

**Design, Analysis and Implementation
of Bulk-Synchronous Parallel
Algorithms, Data structures and Techniques**



Constantinos J. Siniolakis
St. Peter's College, Oxford University

Programming Research Group, Oxford University,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Supervisor: Prof. W. F. McColl

Trinity 1998

*Thesis submitted for the degree of Doctor of Philosophy
at the University of Oxford*



Abstract

The objective of this thesis is the unified investigation of a wide range of fundamental parallel methods that are transportable amongst pragmatic parallel machines having different number of processors, different periodicity of global synchronization and different bandwidth of inter-processor communication. The computational model adopted is the bulk-synchronous parallel (BSP) model, which abstracts the characteristics of parallel machines into three numerical parameters p , L and g , that quantify, respectively, processors, periodicity and bandwidth – the model differentiates memory that is local to a processor from memory that is non-local, yet, for the sake of universality, does not differentiate network proximity. The BSP parameters p , L and g , together with the problem size n , are employed to measure the performance, and consequently, the transportability of parallel methods across machines having different values of these parameters.

We show that optimality to within small multiplicative constant factors close to *one* can be achieved for a multiplicity of fundamental computational problems by transportable algorithms and data structures that can be applied for a wide range of values of the BSP parameters. While these algorithms and data structures are fairly simple themselves, description of their performance in terms of these parameters is somewhat complicated. The main reward for quantifying these complications, is that it enables software to be written once and for all that can be migrated efficiently amongst a variety of parallel machines. The methods considered in this thesis – both theoretically and experimentally – embody deterministic and randomized techniques for the efficient realization of fundamental algorithms (broadcasting, computing parallel-prefixes, load-balancing, list-contracting, merging, sorting, integer-sorting, selecting, searching and hashing), data structures (heaps, search trees and hash tables) and applications (computational geometry, parallel model simulations and structured query language primitives).

Acknowledgments

Many people have contributed substantially to the creation of this thesis. On the technical side, I am most indebted to my supervisor Prof. Bill McColl. Bill and I spent many hours discussing on my work and he is directly responsible for improving the quality of the manuscript. In addition to suggesting simpler explanations for several results and recommending further references, Bill also helped provide motivation for completing the thesis on time.

Of course, many other people provided technical assistance with this work. I am particularly thankful to Dr. Alexandros Gerbessiotis for countless helpful discussions, suggestions and pointers. Many of the results and explanations presented in this work are due to Alexandros or were improved as a result of his influence.

I would also like to thank the Bodossaki Foundation for their gracious financial support during the past few years. This support has been very helpful to my research and to the preparation of this text. I am also grateful to my colleagues and instructors in the Oxford University Computing Laboratory for their support and for providing such a stimulating and congenial environment within which to work.

Special recognition also goes to my examiners Prof. Martin Dyer and Dr. Jonathan Hill for their invaluable comments and suggestions.

Finally, I would like to thank my family for all their support through the years and Marilena for her love through all the early mornings and late nights that were spent with “The Thesis.”

Contents

1	Introduction	1
1.1	Parallel Computing	1
1.2	A Bridging Model: The BSP Approach	2
1.2.1	The Theoretical Model	3
1.2.2	The Practical Model	5
1.2.3	The Cost Model	7
1.2.4	Augmentations to the Standard BSP Model	8
1.3	Preliminaries and Notation	8
1.3.1	Basic Computational Operations	9
1.3.2	Randomization	9
1.3.3	Asymptotic Analysis	10
1.3.4	Optimality	10
1.3.4.1	Specifying the Performance of BSP Algorithms	10
1.3.5	Speedup and Efficiency	11
1.3.6	Related Work and New Results	12
1.4	A BSP Algorithmic Library	12
1.4.1	Experimental Results	13
2	Fundamental Algorithms	16
2.1	Primitive Operations	16
2.1.1	Broadcasting and Parallel-Prefix	16
2.1.1.1	A Broadcasting Algorithm	16
2.1.1.2	Implementation and Experimental Results	18
2.1.1.3	A Parallel-Prefix Algorithm	19
2.1.1.4	A Segmented Parallel-Prefix Algorithm	21
2.1.1.5	Implementation and Experimental Results	21
2.1.2	Load-Balancing	22
2.1.2.1	A Load-Balancing Algorithm	22
2.1.2.2	Implementation and Experimental Results	22
2.2	List-Contraction	23
2.2.1	Preliminaries and Notation	23
2.2.2	Cost Model	23
2.2.3	Related Work and New Results	23

2.2.4	An Optimal Randomized List-Contraction Algorithm	24
2.2.4.1	Implementation and Experimental Results	27
2.3	Merging	29
2.3.1	Preliminaries and Notation	29
2.3.2	Cost Model	29
2.3.3	Related Work and New Results	30
2.3.4	A Sub-Optimal Merging Algorithm	30
2.3.4.1	Implementation and Experimental Results	32
2.3.5	An Optimal Merging Algorithm	33
2.4	Integer-Sorting	37
2.4.1	Preliminaries and Notation	37
2.4.2	Cost Model	37
2.4.3	Related Work and New Results	37
2.4.4	An Optimal Integer-Sorting Algorithm	38
2.4.4.1	Implementation and Experimental Results	40
2.5	Probabilistic Integer-Sorting	42
2.5.1	Preliminaries and Notation	43
2.5.2	Cost Model	43
2.5.3	Related Work and New Results	43
2.5.4	A Combinatorial Lemma	44
2.5.5	An Optimal Probabilistic Integer Sorting Algorithm	46
2.5.6	Yet Another Optimal Probabilistic Integer-Sorting Algorithm	49
2.5.6.1	Implementation and Experimental Results	51
2.6	Sorting	52
2.6.1	Preliminaries and Notation	53
2.6.2	Cost Model	54
2.6.3	Related Work and New Results	54
2.6.4	A Sub-Optimal Sorting Algorithm	56
2.6.5	A Sorting Algorithm with n^2 Processors	57
2.6.6	An Optimal Deterministic Sorting Algorithm	58
2.6.6.1	Improving Deterministic Sorting	64
2.6.6.2	Implementation and Experimental Results	65
2.6.7	An Optimal Randomized Sorting Algorithm	67
2.6.7.1	Improving Randomized Sorting	76
2.6.7.2	Implementation and Experimental Results	77
2.6.8	A Fully Scaleable Optimal Sorting Algorithm	79
2.6.8.1	The CREW-BSP and BCAST-BSP Variants	79
2.6.8.2	The BSP Variant	86
2.6.8.3	A Lower Bound	95
2.6.8.4	Applications of Sorting in Computational Geometry Algorithms	103
2.6.9	Duplicate Item Handling	104
2.7	Converting Sorting Algorithms into Fast Randomized Ones through Searching	104
2.7.1	Preliminaries and Notation	104

2.7.2	Cost Model	105
2.7.3	An Algorithmic Transformation	105
2.7.3.1	Implementation and Experimental Results	107
2.8	Selection	108
2.8.1	Preliminaries and Notation	109
2.8.2	Cost Model	110
2.8.3	Related Work and New Results	110
2.8.4	A Randomized Sorting Algorithm with $n^{1+\epsilon}$ Processors	111
2.8.5	An Optimal Randomized Selection Algorithm	113
2.8.5.1	Improving Randomized Selection	118
2.8.5.2	Implementation and Experimental Results	119
2.9	Searching	120
2.9.1	Preliminaries and Notation	121
2.9.2	Cost Model	121
2.9.3	Related Work and New Results	121
2.9.4	Ordered h -level Graphs	122
2.9.4.1	Preliminaries and Notation	123
2.9.5	Multi-Way Searching on Ordered h -Level Graphs	124
2.9.6	An Optimal Multi-Way Searching Algorithm on Ordered h -level Graphs	125
2.9.6.1	The Case $m \leq n$ through Load-Balancing	130
2.9.6.2	The Case $m > n$ through Randomization	131
2.9.6.3	Improving Multi-Way Searching	134
2.9.6.4	One-Optimal Applications of Multi-Way Searching in Computational Geometry Algorithms	142
2.9.6.5	Further Applications of Multi-Way Searching in Computational Geometry Algorithms	147
2.10	Applications of Fundamental Algorithms to Computational Geometry Algorithms	148
2.10.1	Preliminaries and Notation	148
2.10.2	Cost Model	149
2.10.3	Related Work and New Results	149
2.10.4	An Optimal Convex-Hull Construction Algorithm	150
2.10.4.1	Implementation and Experimental Results	153
2.10.5	Optimal Geometric Searching and Dominance-Counting Algorithms	155
2.10.5.1	Applications of Dominance Counting in Rectangle Inclusion	158
2.10.6	Optimal Visibility and Lower-Envelope Construction Algorithms	159
2.10.7	Improving Computational Geometry Algorithms	162
2.10.7.1	Improving Convex-Hull Construction I	162
2.10.7.2	Improving Convex-Hull Construction II	164
2.10.8	Other Optimal Computational Geometry Algorithms	165
2.10.8.1	Intersection of Horizontal and Vertical Line Segments	166
2.10.8.2	Three-Dimensional Maxima	166
2.10.8.3	All Nearest Neighbors	166
2.10.8.4	All Furthest Neighbors	166

2.10.8.5	Minimum-Area Enclosing Rectangle	167
3	Fundamental Data Structures	168
3.1	Priority Queues	168
3.1.1	Preliminaries and Notation	168
3.1.2	Cost Model	169
3.1.3	Related Work and New Results	169
3.1.4	(n, d) -Bandwidth Concurrent Heaps	171
3.1.4.1	Preliminaries and Notation	171
3.1.5	Optimal Randomized Inserting, Deleting-Min and Finding-Min Algorithms on (n, d) - Bandwidth Concurrent Heaps	172
3.1.5.1	Improving Deleting-Min and Finding-Min	177
3.1.5.2	Introducing Concurrency	180
3.2	Dictionaries	183
3.2.1	Preliminaries and Notation	184
3.2.2	Cost Model	184
3.2.3	Related Work and New Results	185
3.2.4	(d, α) - B_{2-3-4}^{+-} -Trees	186
3.2.4.1	Preliminaries and Notation	187
3.2.5	Optimal Randomized Inserting, Deleting and Finding Algorithms on (d, α) - B_{2-3-4}^{+-} - Trees	188
3.2.6	Hash Tables	190
3.2.6.1	Preliminaries and Notation	190
3.2.6.2	Hashing	190
3.2.6.3	(δ, s) -Quasi-Random Functions	193
3.2.7	Optimal Randomized Inserting, Deleting and Finding Algorithms on Parallel Hash Tables	195
3.2.7.1	Static Hash-Table Algorithms	195
3.2.7.2	Dynamic Hash-Table Algorithms	201
3.2.8	Duplicate Item Handling	204
3.3	Priority Queues and Dictionaries Revisited	205
3.3.1	Parallel Priority Queues	206
3.3.1.1	Preliminaries and Notation	206
3.3.2	Optimal Randomized Inserting, Deleting-Min and Finding-Min Algorithms on Par- allel Priority Queues	206
3.3.2.1	Implementation and Experimental Results	213
3.3.3	Parallel Dictionaries	215
3.3.3.1	Preliminaries and Notation	216
3.3.4	Optimal Randomized Inserting, Deleting and Finding Algorithms on Parallel Dic- tionaries	216
3.3.4.1	Implementation and Experimental Results	222
4	Conclusion	224
4.1	Epilogue	224

A Probabilistic Results	226
A.1 Tail Inequalities	226
A.1.1 Applications of Azuma's Inequality	227
A.2 Randomized Results	230
A.2.1 Random Sampling Applied to Sorting	230
A.2.2 Random Sampling Applied to Selection	233
A.2.3 Hashing	237
B A Sample Program	245
B.1 Source Code	245
B.1.1 BSP Deterministic Sorting	245

Chapter 1

Introduction

1.1 Parallel Computing

The explosion of parallel computing we are witnessing is arousing extraordinary interest at every level of society. As the parallel computing hardware is continuing unabated its current steep performance gradient, computations once infeasible become routine. Another factor, however, has had an even more momentous effect in extending the frontiers of feasible computation: the use of efficient parallel *algorithms*, *data structures* and *techniques*.¹

The Concise Oxford Dictionary defines the term *algorithm* – the subject matter of this work – as “*Math.* a process or set of rules used for calculation or problem-solving, esp. with a computer”, that is, the execution of an algorithm does not encompass any subjective decisions, nor does it require the use of intuition or creativity [41]. The theme of this work is the *architecture independent* study (*design*, *analysis* and *implementation*) of a multiplicity of fundamental parallel methods (*algorithms*, *data structures* and *techniques*).²

This work is neither a comprehensive account of theoretical results on architecture independent parallel algorithms and data structures nor a thorough investigation of all available techniques. Still less is it a programming manual containing a long catalogue of programs ready to be used directly on parallel machines to solve certain specific problems, but giving at best a concise abstraction of the ideas involved in their design and analysis.³ On the contrary, the aim of this work is to give the reader some basic understanding of the principles involved in the architecture independent design and analysis of certain

¹In the remainder of this work, we sporadically employ the term algorithm(s) to collectively refer to algorithm(s), data structure(s) and technique(s).

²A functional division of parallel algorithms is the division between those that exploit data parallelism (a single thread of control that acts on many homogenous data items in parallel) and those that exploit control parallelism (a multiplicity of threads of control that act on possibly heterogeneous data items). Although the distinction is not always sharp, it does indeed capture an important aspect of parallel computing. Undoubtedly, both classes of parallel algorithms are consequential and research in both areas is emphatically required. In this work, however, we focus on data parallel algorithms.

In order for a problem to be amenable to a data parallel algorithm, it should pertain to a massive homogenous data set – such a set allows the underlying problem to be efficiently partitioned into sub-problems that can be subsequently processed in parallel. Fortunately, a vast number of important and demanding problems in the realm of science involve massive homogenous data sets.

³Even though most of the algorithms and data structures have been actually implemented and experimental results reported, in this work we refrain from presenting the low-level details involved in the implementations, but rather, we focus on the underlying high-level principles – with the exception of one or two purely illustrative examples.

fundamental parallel algorithms and data structures. Although our approach is rigorous and theoretical, we do not neglect the needs of practitioners: besides illustrating the design and analysis of the underlying methods, most of the methods also have real-life applications and implementations.

Thus we concentrate on the techniques used to design, analyze and subsequently implement efficient algorithms and data structures. These architecture independent techniques include but are not limited to: *balanced trees* (e.g., broadcasting [section 2.1] and parallel-prefix [section 2.1]), *balancing* (e.g., load-balancing [section 2.1]), *pointer jumping* (e.g., list contraction [section 2.2]), *symmetry breaking* (e.g., list contraction [section 2.2]), *over-sampling* (e.g., merging [section 2.3], sorting, [section 2.6] and selection [section 2.8]), *partitioning* (e.g., merging [section 2.3], sorting [section 2.6]), *divide-and-conquer* (e.g., sorting [section 2.6] and computational geometry [section 2.10]), *cascading divide-and-conquer* (e.g., sorting [section 2.6]), *pipelining* (e.g., heaps [section 3.1] and search trees [section 3.2]) and *hashing* (e.g., hash tables [section 3.2]). We note, however, that these techniques are not presented in their full generality, but rather reflected through concrete examples of fundamental algorithms and data structures.

In the last few paragraphs we have recurrently practiced the term *architecture independent* with reference to algorithms, data structures and techniques. The rationale behind emphasizing the aforementioned term derives from the recent developments in the field of parallel computing: indubitably, parallelism has been established as a viable approach to high-performance computing, however, a consensus has not yet emerged on how parallelism should best be exploited – a multiplicity of radically different system architectures and programming frameworks have been proposed, praised and criticized by different schools of thought. We argue – the subject matter of the following section – that if the parallel computing industry is to prosper, then a unifying, or indeed, a *bridging* model of parallel computation has to be established on a substantial base. In the following section, we briefly evaluate the underlying principles of such a bridging model and claim that the model has to be architecture independent in order to serve its purpose.

1.2 A Bridging Model: The BSP Approach

Over the last three decades or so a major challenge for the parallel computing community has been to develop and establish, on a substantial base, a model of parallel computation that could serve as a basis, or indeed, as a kind of unifying platform for the *design*, *analysis* and *implementation* of high-performance parallel *algorithms* and *data structures* that could be implemented efficiently and effectively on a wide range of current and future parallel machines.⁴ Unfortunately, despite the enormous research that has been undertaken on this front in recent years, no single model of parallel computing has yet come to dominate advancements in the way that the von Neumann model has dominated sequential computing over the last fifty years [149, 150].

A close investigation of the body of parallel algorithms and data structures – developed under past and current parallel models – reveals that the vast majority are impractical because they exploit artificial factors not present in any realistic current or near-future machine, such as zero communication delay or infinite bandwidth [55]. Others are overly and unduly specialized because they are tailored to the individual idiosyncrasy of a given parallel machine, such as a particular interconnection topology [55].⁵

⁴The field of parallel computing is developing so rapidly nowadays that in some sense making any statement about the present – much less the future – is a risky endeavor. Nonetheless, some clear directions can be delineated.

⁵As noted in [153, 154], on most of the early commercial parallel machines, scaleable parallel performance could only be achieved by tailoring algorithms and data structures to the particular details of the underlying parallel machine. Besides being extremely tedious and resource consuming in most cases, this approach usually resulted in parallel software that can not

To remedy the situation, the parallel community needs to design parallel algorithms and data structures using a *bridging* model that is *realistic* enough to capture the most important performance factors of current or near-future machines, *abstract* enough to ensure universality, and yet *simple* enough to promote the design, analysis and implementation of tractable parallel methods.⁶

The *bulk-synchronous parallel* (BSP) model has been proposed as a viable candidate for the role of bridging model [204, 205, 206].⁷

1.2.1 The Theoretical Model

The BSP model of computation has been proposed by L. G. Valiant [206], as a unified platform for the design, analysis and implementation of general purpose parallel machines. It offers the prospects of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework that allows the performance of parallel programs to be analyzed and predicted in a precise way. The term *bulk-synchronous* reflects the underlying position of the model between the two extremes, (i) entirely synchronous systems (a global clock is utilized to achieve synchronization on every instruction), and (ii) fully asynchronous systems (synchronization is realized pair-wise whenever communication is required). Processing units in the BSP model advance jointly through a program, with the required communication occurring between *supersteps*: segments of computation delimited by a barrier synchronization.

A BSP machine as originally introduced in [204, 205, 206] and further addressed in [75, 111, 149, 150, 151, 152, 153, 154, 156, 200, 207] consists of the following three components.

- (1) A *collection* of p *processor-memory components*.
- (2) A *mechanism* for efficient *communication* of data point-to-point amongst the processor-memory components.
- (3) A *mechanism* for efficient *barrier synchronization* of all or a subset of the processor-memory components.

In the remainder of this work we use the terms “processor-memory component” and “processor” interchangeably. The performance of any BSP machine can be characterized by the following four parameters.

- (1) s – the performance of each processor, measured in basic computational operations per time unit.⁸
- (2) p – the number of processors.
- (3) L – the minimum elapsed time between successive synchronization steps, measured in basic computational operations.

be transported on other parallel machines. In a computing environment of rapidly developing diverse parallel architectures, this machine dependence of parallel software proved to be a major weakness and serious inhibitor to the growth of the field.

⁶For a variety of economic and technological reasons, over the last decade or so we have evidenced a high-level architectural convergence of models for parallel scalable computing – the economic advantages of using standard commodity components has been the major factor in this convergence; other influential factors have been the need to efficiently support a shared address space on distributed-memory machines (for easy of programming) and the need to replace bus-based shared-memory machines by network-based distributed-memory machines. As a consequence, there is currently a growing consensus that over the next few years we shall evidence steady evolution towards a “standard” architectural model for scalable parallel computing [153, 154] – the high-level architectural model is likely to consist of a collection of processor-memory pairs connected by a sparse communication network.

⁷The BSP model is essentially the “standard” model described above.

⁸The term *basic computational operation* is formally defined in subsequent section 1.3.1.

- (4) g – the ratio of the total throughput of all p processors in terms of basic computational operations to the total throughput of the communication mechanism in terms of words of information delivered. More formally, if h denotes maximum number of incoming or outgoing words over all processors in a given superstep (such a communication pattern is called an h -relation), then g is related to the time required to realize an h -relation in $g \cdot h$ time, for h larger than or equal to some h_{min} [206], measured in basic computational operations.

In order to compare the parallel performance of different BSP machines, we assume that the parameters L and g are normalized with respect to the processor performance s , and consequently, all time measurements are expressed in terms of basic computational operations (e.g., *flops*). Consequently, in the remainder of this work we ignore parameter s and concentrate on the remaining three BSP parameters p , L and g .

The values of L and g for any BSP machine are likely to be monotonically non-decreasing functions of the number of processors p [149, 200]. In particular, the BSP machine parameters L and $1/g$ are proportional to the latency and bisection bandwidth of the underlying communication network, but not equivalent.⁹ For example, g depends on other factors such as: the protocols used to interface with and within the communication network, the buffer management by both the processors and the communication network, the routing strategy used within the communication network and the BSP run-time system [149, 200]. Similarly, L depends on other factors such as: the software or hardware mechanism available on the communication network for synchronizing and the way interrupts are handled by processors [200]. We note that a lower bound on L is imposed on the basis that the theoretical bound on g may be realizable only for large enough L , so that the communication network can achieve the stated capacity – i.e., L should satisfy $L \geq g \cdot h_{min}$, so that every h -relation, for h less than h_{min} , is charged as an h_{min} -relation. Table 1.1 summarizes the typical values of s , p , L and g for several commercial parallel machines [200].

Computation on the BSP model proceeds in a succession of *supersteps*; a superstep may be conceived as a segment of a computation during which each processing unit performs a given thread using data already available there locally before the start of the superstep. The thread may include (i) computation steps on data held locally at the start of the superstep, (ii) data transmissions, and (iii) data receipts. At the end of each superstep, all processing units wait in a global *barrier synchronization* until the non-local data exchanges are completed before proceeding to the next superstep (for a graphical illustration refer to figure 1.1). It should be noted that, although the model stresses global barrier mode synchronization, pairs of processing units may synchronize pair-wise by sending messages to and from an agreed memory location. However, such message exchanges should respect the superstep rules [75].

The model as described in [206] has two variants depending on how synchronization is handled, that affect performance analysis to small multiplicative constant factors. In the first variant, processing units are checked every L time units to determine whether the threads that are synchronized currently have all terminated and only then proceed to the next superstep. The second variant, permits completion of a superstep at any point after the first L time units. Along the lines of [205, 206] and for the sake of simplicity, in the remainder of this work we adopt the latter variant.

⁹Even though there is currently a growing consensus on a “standard” high-level architectural model for scaleable parallel computing – a collection of processor-memory pairs connected by a sparse communication network – there appears to be no consensus emerging on the underlying low-level interconnection topology [55] – the communication networks of current commercial parallel machines are typically different from their predecessors and different from one another. The high-level convergence of parallel architectures is captured in the BSP model by addressing significant common issues, such as latency and communication throughput, whilst suppressing low-level considerations, such as interconnection topology and routing algorithms.

Machine	$\lfloor s \rfloor$	$\lceil s \rceil$	s	p	L		g	
	$Mflops$				$flops$	μs	$flops/word$	$\mu s/word$
Cray T3D	5.0	19.0	12.0	2	164	13.50	0.70	0.06
				4	168	13.90	0.70	0.06
				8	175	14.40	0.80	0.07
				16	181	14.90	0.90	0.07
				32	201	16.60	1.10	0.09
				64	148	12.30	1.00	0.09
				128	301	24.90	1.10	0.09
				256	387	32.10	1.20	0.11
Cray T3E	4.3	89.2	46.7	2	269	5.70	0.87	0.02
				4	357	7.60	0.87	0.02
				8	506	10.80	0.81	0.02
				16	751	16.00	1.04	0.02
Digital 8400	3.6	44.2	23.9	2	286	11.80	2.50	0.11
				3	362	15.20	3.70	0.16
IBM SP2	25.0	27.0	26.0	2	1903	73.20	6.30	0.24
				4	3583	137.80	6.40	0.25
				8	5412	208.20	6.90	0.27
SGI Origin 2000	36.6	164.7	100.7	2	804	7.90	7.00	0.07
				4	1789	17.90	9.10	0.09
				8	3867	39.80	13.40	0.14
SGI Power Challenge	53.0	94.0	74.0	2	1132	15.30	9.80	0.13
				3	1496	20.20	8.90	0.12
				4	1902	25.70	9.80	0.13

Table 1.1: BSP parameters – s , p , L and g . (1) Three values are given for s : (i) $\lfloor s \rfloor$ is the $Mflop$ rating for a dot-product on two 1,048,570 item single precision floating point arrays, (ii) $\lceil s \rceil$ is the $Mflop$ rating for a dense matrix-matrix multiplication on two 200×200 single-precision floating point matrices, and (iii) s is the average of $\lfloor s \rfloor$ and $\lceil s \rceil$; (2) all values for g are for the communication of 32-bit words; (3) all benchmarks are performed at the -O3 optimization level; (4) the Cray T3D, SGI Power Challenge and IBM SP2 benchmarks use native implementations of the underlying BSP library.

1.2.2 The Practical Model

Although, the BSP model, as previously described, may be regarded as an architectural model, it can be readily employed as a programming platform or, indeed, as a kind of programming framework [152, 156]. The underlying concept in the BSP model is the notion of the superstep and the abstraction that non-local communication associated with a superstep takes place between supersteps as a global operation. Thus, a BSP program may be viewed as a succession of supersteps, with the required non-local communication occurring at the end of each superstep. Viewed in this way, the BSP model can be employed as a unifying programming framework that can be realized on a variety of realistic parallel machines.

To this front, a considerable research effort has been recently invested in forming a world-wide standard BSP library [112] by synthesizing several BSP programming approaches that have been pursued over the last few years [130, 151, 152, 153, 155, 156, 162, 163]. This library, called the Oxford BSP toolset, *BSPlib*, provides a parallel computation and communication platform based around a single program multiple

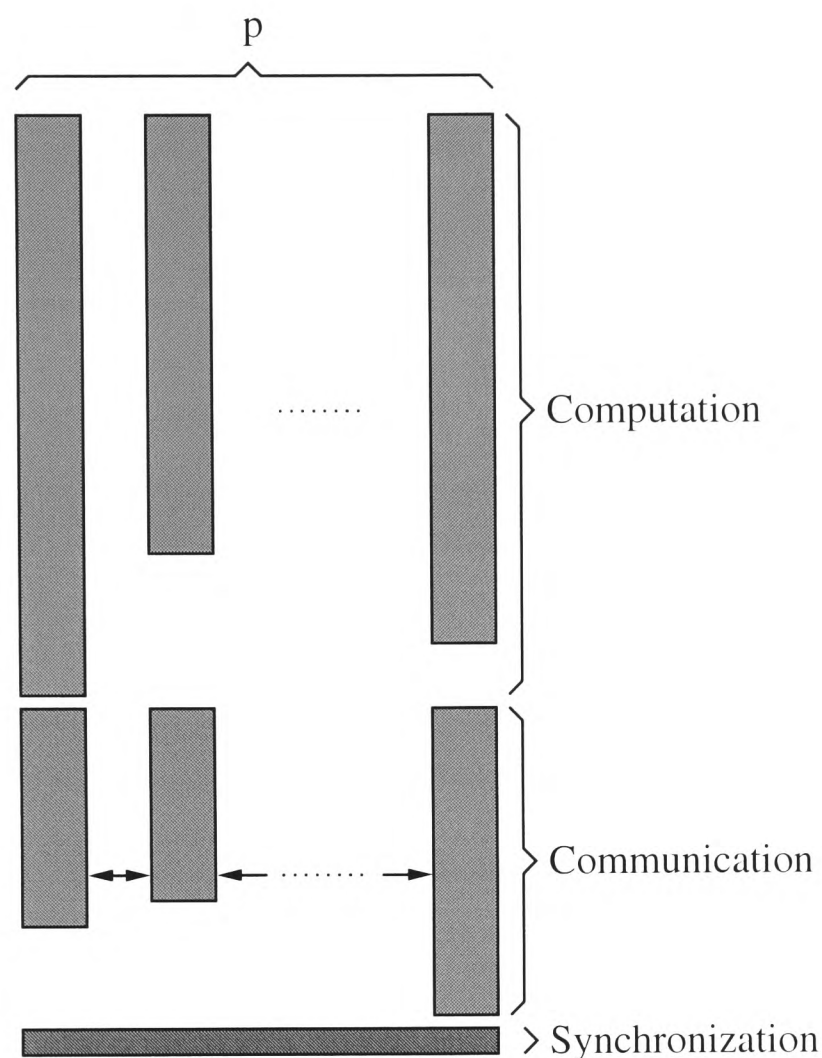


Figure 1.1: A BSP superstep.

data (SPMD) model of computation. The main constituent parts of *BSPlib* are:

- routines to spawn a number of processes to be run in a SPMD manner,
- barrier-synchronization facilities,
- direct remote memory access (DRMA) communication facilities that allow processors to manipulate the address space of remote processors without the active participation of the remote processors, and
- bulk-synchronous message passing (BSMP) communication facilities that allow processors to realize non-blocking send operations.

The essence of the library consists of a handful of routines embodying the three major components of a BSP abstract machine: processors, synchronization and communication. Table 1.2 summarizes the major *BSPlib* primitives. The first three routines handle process management, the next implements the synchronization mechanism and the last four implement the communication mechanism.

The library routines are callable from standard imperative languages (e.g., C, C++ and Fortran). Highly efficient, scaleable, native implementations of the library have been developed by *Oxford Parallel*¹⁰ for various parallel and distributed commercial parallel machines: Convex Exemplar, Cray T3D, Cray T3E, Digital 8400, Digital Alpha Farm, Hitachi SR2001, IBM SP2, Multiprocessor Sun, SGI Origin 2000, SGI Power Challenge and Parsytec GC.

Operation	Routine	Semantics
Initialization/Conclusion	<code>bsp_init</code>	simulate dynamic processes
	<code>bsp_begin</code>	start SPMD code
	<code>bsp_end</code>	end SPMD code
Synchronization	<code>bsp_sync</code>	barrier synchronize
DRMA Communication	<code>bsp_put</code>	write to remote memory
	<code>bsp_get</code>	read from remote memory
BSMP Communication	<code>bsp_send</code>	send to remote queue
	<code>bsp_move</code>	fetch from local queue

Table 1.2: *BSPlib* fundamental primitives.

1.2.3 The Cost Model

The time complexity of a BSP program is merely the sum of the time complexities of its constituent supersteps; in turn, for each superstep, the time complexity can be decomposed into:

- (1) Local computation – referred to as *computation time*.
- (2) Barrier synchronization and non-local communication – collectively referred to as *communication time*.

To ensure that time complexity analysis can be performed in an architecture independent way, cost formulas are based on the architecture dependent BSP parameters p , L and g . In particular, the computation and communication time C_k and M_k respectively of the k -th superstep is captured by the cost formula [206]:

$$C_k = \max_i \{w_i\} \quad \text{and} \quad M_k = L + g \cdot \max_i \{h_i\},$$

where (i) k ranges over the supersteps, (ii) i ranges over processors, (iii) w_i is an architecture independent time measure that reflects the maximum number of basic computational operations executed by processor i in the local computation phase of the k -th superstep, and (iv) h_i is an architecture independent data size measure that reflect the largest accumulated number of incoming or outgoing words at processor i in the non-local communication phase of the k -th superstep. Often the maxima are assumed and the time is expressed in the form w and $L + g \cdot h$, for computation and communication respectively. The overall time of the underlying program is simply the sum of all the individual superstep times $\sum_k (C_k + M_k)$. We call this the standard cost model.¹¹

¹¹Other cost formulas have been proposed, incorporating finer detail [206]. For example, computation and communication can be conveniently overlapped, thus giving the following cost formula:

$$\max \left\{ \max_i \{w_i\}, L + g \cdot \max_i \{h_i\} \right\}.$$

This optimization, however, results in a higher value of g – on most current parallel machines – and therefore, the benefit obtained from overlapping computation and communication is entirely subsumed by the higher value of g [111, 173]. This and other cost model variations [200] alter time analysis results by no more than small multiplicative constant factors, so, in the remainder of this work, we use the standard cost model in the interests of simplicity and clarity.

Remark As noted in [200], the existence of a cost model that is both simple and tractable makes it feasible to truly design and implement efficient parallel algorithms and data structures – that is to consciously and justifiably make choices amongst alternative implementations of a specification. Additionally, the existence of a cost model that is analytic and accurate makes it possible to analyze (and accurately predict) performance across target parallel machines. The number of supersteps and the values of $\max_i \{w_i\}$ and $\max_i \{h_i\}$ for each superstep can be determined by inspection of the program code, subject to the usual limits on predicting the performance of sequential code. The values of L and g can then be incorporated into the standard cost model to accurately predict execution time.

1.2.4 Augmentations to the Standard BSP Model

The standard BSP model, as it has been described, does not support any specialized combining, broadcasting or parallel prefix facilities, although these can be effectively realized in hardware or software where required [75, 205, 206]. This realization has led to the proposal of a multiplicity of alternatives to the standard BSP model that do incorporate such facilities.

In this work we refer to two such variants (augmentations) of the BSP model, namely the *concurrent-read exclusive-write bulk-synchronous parallel* (CREW-BSP) model and the *broadcast bulk-synchronous parallel* (BCAST-BSP) model.

- (1) A CREW-BSP machine is a BSP machine that supports concurrent read operations but excludes concurrent write operations. If h denotes the maximum number of incoming or outgoing words over all processors in a given superstep – in case of read operations, h relates to the number of outgoing words from *distinct* memory cells – then $g \cdot h$ time is charged, for h larger than or equal to some h_{min} [206], measured in basic computational operations. Informally, concurrent read operations (not write operations) are charged as the standard BSP model would charge a single read operation.
- (2) A BCAST-BSP machine is a BSP machine that supports broadcasting from any processor to any subset of processors, provided that the destination of any single broadcast can be described in a constant number of words. If h denotes the maximum number of incoming or outgoing words over all processors in a given superstep – in case of broadcast operations h relates to the number of outgoing words from the *initiating* processor – then $g \cdot h$ time is charged, for h larger than or equal to some h_{min} [206], measured in basic computational operations. Informally, broadcast operations are charged as the standard BSP model would charge a single write operation.

In the bulk of this work, we have almost completely dispensed of employing such artificially augmented models. Nevertheless, for the sake of simplicity, we do sporadically employ such augmented models. We do however present the corresponding standard BSP algorithms at full detail once the simplified high-level analysis of the augmented model algorithms has been established.

1.3 Preliminaries and Notation

For the past few decades the analysis, design and implementation of algorithms and data structures has been one of the most absorbing subjects in computer science. The fundamental work of [131, 132, 133]

has originated an effort to bring order and uniformity to a rich compilation of otherwise isolated results. Subsequent work has further strengthened the theoretical foundations. The scope of this section is merely to introduce the adopted concepts and notation.

1.3.1 Basic Computational Operations

The term *basic computational operation* refers to the time – within a multiplicative constant factor depending solely on the particular implementation used (machine, programming language, and so on) – required to perform one of the following elementary operations.

- The fundamental *arithmetic* operations $(+, -, \times, /, \text{mod})$.
- The common *comparison* operations $(<, \leq, \geq, >, =, \neq)$.
- The flow control *branch* operation *if ... then*.¹²
- The local memory *assignment* operations *read* and *write*.

Throughout this work, the term basic computational operation is employed to characterize such $\Theta(1)$ time primitive operations. Additionally, in order to simplify the exposition, the following functions are assumed to be basic computational operations – this assumption can be removed without affecting the asymptotic analysis of the subsequent sections.

- $\lg(\cdot)$, $\log(\cdot)$, $\log_t(\cdot)$ and $\exp(\cdot)$, where $\lg(x)$, $\log(x)$ and $\log_t(x)$ denote the logarithms of x to base 2, e and t respectively and $\exp x$ denotes the inverse of $\log x$.
- $\sqrt{(\cdot)}$ and $\sqrt[3]{(\cdot)}$.
- $\lfloor(\cdot)\rfloor$ and $\lceil(\cdot)\rceil$.
- $\text{rnd}(\cdot)$, where $\text{rnd}(x)$ denotes a random integer in the range $[0, 1, \dots, \lceil x \rceil - 1]$.

1.3.2 Randomization

A *randomized* algorithm (whether sequential or parallel) is one that bases some of its decisions on random choices and terminates within a pre-specified running time with a given probability. The running time of such an algorithm is said to be *probabilistic*. A *deterministic* algorithm, on the other hand, is one that terminates within a pre-specified time with certainty. In this work we refer to an algorithm as being deterministic in those circumstances where it is to be contrasted with a randomized algorithm. When no qualifier is used explicitly, it is implied that the algorithm in question is deterministic. With reference to randomized algorithms, the term *high probability* is employed in this work to denote probability $P \geq 1 - n^{-\rho}$, for any constant $\rho > 0$, where n is the problem size. In subsequent sections, randomized algorithms are employed successfully to obtain efficient BSP algorithms for a multiplicity of fundamental computational problems.

¹²In general, if S is any statement that can be true or false, the bracketed notation $[S]$ evaluates to 1 if S is true, and 0 otherwise.

1.3.3 Asymptotic Analysis

The work of [134] has manifested a notational methodology, usually referred to as *asymptotic analysis*, that differentiates properly between upper and lower bounds. Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals. Then,

- $O(f(n))$ – denotes the set of all functions $g(n)$ such that there exist two (unspecified) positive constants c and N_0 with $g(n) < c \cdot f(n)$ for all $n \geq N_0$.
- $\Omega(f(n))$ – denotes the set of all functions $g(n)$ such that there exist two (unspecified) positive constants c and N_0 with $g(n) > c \cdot f(n)$ for all $n \geq N_0$.
- $\Theta(f(n))$ – denotes the set of all functions $g(n)$ with $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$.
- $o(f(n))$ – denotes the set of all functions $g(n)$ such that for *all* positive constants c there exists a positive constant N_0 with $g(n) < c \cdot f(n)$ for all $n \geq N_0$.
- $\omega(f(n))$ – denotes the set of all functions $g(n)$ such that for *all* positive constants c there exists a positive constant N_0 with $g(n) > c \cdot f(n)$ for all $n \geq N_0$.

The $O(\cdot)$, $o(\cdot)$ and $\Omega(\cdot)$, $\omega(\cdot)$ notations are used, respectively, to express *upper* and *lower* bounds on the computational complexity of problems.

Remark Obviously, the “order of” notation conceals multiplicative constants, and therefore, a complexity result secures its full validity only for sufficiently large problem sizes. Hence, it is not uncommon that for small problem sizes the most efficient algorithm is not necessarily the asymptotically best algorithm. Practice has demonstrated that, for small problem sizes, many asymptotically inferior algorithms behave better than the corresponding asymptotically superior algorithms. Furthermore, a multiplicity of algorithms perform considerably better on the *average* than predicted by the *worst-case* analysis.

1.3.4 Optimality

In this work, the theoretical performance of algorithms is measured in terms of worst-case analysis – unless otherwise indicated. In particular, we specify the performance of BSP algorithms by comparing their analytic run-time with that of (their corresponding) sequential algorithms. In order to achieve this, we measure run-times on pairs of theoretical models – BSP and RAM [52] – that have the same set of local instructions.

Let Q be a problem of size n for which there exists a p -processor BSP algorithm that requires time $T_Q(n, p)$. The product $W = p \cdot T_Q(n, p)$ represents the *work* performed by the BSP algorithm. The algorithm is called *work optimal* (in short *optimal*), if the work W performed satisfies $W = \Theta(T_Q(n))$, where $T_Q(n)$ denotes the time complexity of the fastest known sequential algorithm to solve Q – the term $T_Q(n)$ appearing in the above definition is not necessarily the value of $T_Q(n, p)$ for $p = 1$.¹³

1.3.4.1 Specifying the Performance of BSP Algorithms

In this work we concentrate on optimal BSP algorithms; in fact, in the spirit of [75], for a multiplicity of problems we adopt the notion of c -optimal BSP algorithms. In particular, we specify two ratios π and μ .

¹³The work W would be more naturally compared to the fastest sequential algorithm (rather than just the fastest known), but in general this quantity is difficult to determine, and therefore, impractical to use [102].

The former, π , is the ratio between the computation time $C_Q(n, p)$ of the p -processor BSP algorithm, over the time $T_Q(n)$ of the comparing sequential algorithm, divided by p , that is $\pi = (p \cdot C_Q(n, p))/T_Q(n)$. The latter, μ , is the ratio between the communication time $M_Q(n, p)$ of the p -processor BSP algorithm, over the time $T_Q(n)$ of the comparing sequential algorithm, divided by p , that is $\mu = (p \cdot M_Q(n, p))/T_Q(n)$. Then, a BSP algorithm is c -optimal, for some constant $c \geq 1$, if $\pi = 1 + o(1)$ and $\mu = o(1)$ as the problem size $n \rightarrow \infty$. We note that the c -optimality notion does not require the specification of a low level instruction set in order to detail how the local operations of a BSP algorithm are charged, as compared to the local operations of some reference sequential algorithm. In particular, since the local operation mechanisms of the BSP model are similar to those of the RAM model [52] and π , μ represent ratios between these two, it is possible to be specific at a high level instruction set – the particulars of such an instruction set are specified with reference to a given algorithm and/or data structure and are diverged to later sections (under the heading “Cost Model”). Obviously, insistence on c -optimality leads to algorithms that only work in restricted ranges of the BSP parameters p , L and g .

Remark The notion of c -optimality is considerably stronger than the (overly simplified) standard optimality notion discussed at the introduction of this section. The major advantage of c -optimality over standard optimality is that the c -optimality notion takes under explicit consideration multiplicative constant factors, and therefore, avoids the caveats of the “order of” notation that is inherent in the standard optimality notion.

We have not yet factored the running time $T_Q(n, p)$ of the BSP algorithm into our notion of optimality. An optimal parallel algorithm is *work-time optimal* or *optimally fully scaleable* if it can be shown that $T_Q(n, p)$ cannot be improved – within multiplicative constant factors – by any other optimal BSP algorithm. Hence, the running time of a work-time optimal algorithm represents the ultimate speed that can be achieved without sacrificing in the total number of operations.

1.3.5 Speedup and Efficiency

In this work, the practical performance of algorithms is measured in terms of time measurements – unless otherwise indicated. In particular, we specify the performance of BSP algorithms by comparing their experimental run-time with that of (their corresponding) sequential algorithms. In order to achieve this, we measure run-times on pairs of practical models – BSP and RAM [52] – under the same clock mechanism.

Let Q be a problem of size n for which there exists a p -processor BSP algorithm implementation that requires time $T_Q(n, p)$. Additionally, let $T_Q(n)$ denote the time required by the fastest known sequential algorithm implementation to solve Q .¹⁴ Then, the ratio

$$S = \frac{T_Q(n)}{T_Q(n, p)}$$

represents the *speedup* achieved by the BSP algorithm implementation. Obviously, S measures the speedup factor obtained by the BSP algorithm when p processors are available, and therefore, ideally, since $S \leq p$, the design and implementation processes should aim at algorithms that achieve $S \approx p$. Similarly, the ratio

$$E = \frac{T_Q(n)}{p \cdot T_Q(n, p)}$$

¹⁴For the sake of simplicity and uniformity, it is not uncommon practice in the experimental part of this work to replace $T_Q(n)$ by $T_Q(n, 1)$ whenever $T_Q(n)$ is not known or difficult to realize.

represents the *efficiency* achieved by the BSP algorithm. Obviously, E measures the effective utilization of the p available processors relative to the given BSP algorithm implementation, and therefore, ideally, since $E \leq 1$, the design and implementation processes should aim at algorithms that achieve $E \approx 1$.

Unfortunately, there are several factors that introduce inefficiencies and inhibit the ultimate aim of achieving $S \approx p$, or equivalently, $E \approx 1$ (e.g., insufficient concurrency in the computation, communication and synchronization overheads). A major goal in this work is the design, analysis and implementation of algorithms that achieve the best possible speedup and efficiency.

1.3.6 Related Work and New Results

Throughout this work (in sections under the heading “Related Work and New Results”) we compare the theoretical performance of our BSP algorithms with well-known parallel algorithms on other models. In order to simplify the exposition, we present these comparison results on an abstract parallel model with no concurrent capabilities – unless otherwise indicated. In general, the performance of these algorithms is summarized with respect to the following statistics: *computation time*, *communication time*, *multiplicative constant factors*, *implementation effort* and amount of *parallel slackness* (in terms of *maximum number of processors*) required to achieve optimality. We note that the multiplicative constant factors and implementation effort statistics are presented with the following graphical format:

- (1) ■ – no (hidden) multiplicative constant factors.
- (2) ■ – low (hidden) multiplicative constant factors.
- (3) ■■ – moderate (hidden) multiplicative constant factors.
- (4) ■■■ – high (hidden) multiplicative constant factors.
- (5) ■■■■ – very high (hidden) multiplicative constant factors.
- (1) ■ – very low implementation effort.
- (2) ■ – low implementation effort.
- (3) ■■ – moderate implementation effort.
- (4) ■■■ – high implementation effort.
- (5) ■■■■ – very high implementation effort.

Obviously, the aforementioned graphical format lacks preciseness, yet, possesses conciseness and clarity.

1.4 A BSP Algorithmic Library

This work is conducted in conjunction with a major project which is currently underway at Oxford Parallel, that of developing a comprehensive and integrated library of BSP algorithms for computationally intensive problems. The objective is to specify, design, analyze and implement a core of BSP algorithms in several scientific areas. On this premise we settled on developing a library of efficient BSP fundamental algorithms. The main features of the library are summarized as follows:

- A sizable collection of algorithms and data structures in a form that allows them to be used by non-experts.

- A precise and readable specification for each of the algorithms and data structures.
- An efficient implementation for each of the algorithms and data structures.

We hope that this library will narrow the gap between algorithmic research, implementation and teaching.

The constituent parts of the library are analyzed in detail at the relevant sections. The discussion in the following section lays the foundations for experimentally assessing the practical viability (in terms of performance) of the algorithmic library.

1.4.1 Experimental Results

Throughout this work we employ a set of data set benchmarks to compare the different algorithms, data structures and techniques, and to examine their performance on commercial parallel machines with respect to a multiplicity of varying parameters (in sections under the heading “Implementation and Experimental Results”).¹⁵

As noted in [186], to decide on a suitable benchmark a combination of contradicting objectives has to be considered and evaluated:

- Too many computational results might tire the reader.
- Too few results might not exhibit enough insight.
- Not all test instances are suitable for every problem.
- Problems from different sources might not be comparable in a fair way.

To overcome these limitations, in the remainder of this work, we adopt the following framework.

- A set of data set benchmarks (of the same type) has been selected that can always be treated by our methods.
- Considerable effort has been invested in conforming to previously published data set benchmarks [20, 108, 109] by including them in our experimental study. The six predominant benchmarks are defined as follows, where n denotes the problem size, p the number of available processors and $m = 2^{31}$ the maximum number of non-negative integer values accommodated in a 32-bit signed arithmetic data type. To simplify the exposition, we assume that p is an integral power of two.

- (1) *Uniform* (U) – a uniformly distributed random input in the range 0 to $m - 1$.¹⁶
- (2) *Gaussian* (G) – a Gaussian distributed random input in the range 0 to $m - 1$.¹⁷
- (3) *Bucket Sorted* (B) – an input that is generated as follows: the input items are obtained by setting the first $\lceil n/p^2 \rceil$ items at each processor to uniformly distributed random values in the range 0 to $\lceil m/p \rceil - 1$, the next $\lceil n/p^2 \rceil$ items at each processor

¹⁵In all of our algorithms we employed the BSP model in both the design and implementation stages of the algorithm development; that is, we based our design and implementation choices purely on the BSP cost model (e.g., the algorithms take into explicit consideration the BSP parameters p , g and L).

¹⁶Uniformly distributed random values are approximated by calling a pseudo random number generator: the C standard library function `rand48`, which returns an integer in the range $[0, 1, \dots, m - 1]$.

¹⁷Gaussian distributed random values are approximated by adding four uniformly distributed random values and dividing the sum by four.

to uniformly distributed random values in the range $\lceil m/p \rceil$ to $\lceil (2 \cdot m)/p \rceil - 1$, and so forth.

- (4) *k-Group* (k-G) – an input that is generated as follows: the processors are initially divided into p/k groups each consisting of k processors, where k is any integer that partitions p evenly, and within the j -th group, $1 \leq j \leq p/k$, the input items are obtained by setting the first $\lceil n/(k \cdot p) \rceil$ items at each processor to uniformly distributed random values in the range $((j-1) \cdot k + p/2 - 1) \bmod p + 1 \cdot \lceil m/p \rceil$ to $((j-1) \cdot k + p/2) \bmod p + 1 \cdot \lceil m/p \rceil - 1$, the next $\lceil n/(k \cdot p) \rceil$ items at each processor to uniformly distributed random values in the range $((j-1) \cdot k + p/2) \bmod p + 1 \cdot \lceil m/p \rceil$ to $((j-1) \cdot k + p/2 + 1) \bmod p + 1 \cdot \lceil m/p \rceil - 1$, and so forth.
- (5) *Staggered* (S) – an input that is generated as follows: if processor index $0 \leq i \leq p/2$, then the input items are obtained by setting the items at processor p_i to uniformly distributed random values in the range $(2 \cdot i - 1) \cdot \lceil m/p \rceil$ to $(2 \cdot i) \cdot \lceil m/p \rceil - 1$; otherwise, the input items are obtained by setting the items at processor p_i to uniformly distributed random values in the range $(2 \cdot i - p - 2) \cdot \lceil m/p \rceil$ to $(2 \cdot i - p - 1) \cdot \lceil m/p \rceil - 1$.
- (6) *Deterministic Duplicates* (DD) – an input of duplicates that is generated as follows: if processor index $0 \leq i \leq p/2$, then the input items are obtained by setting the items at processor p_i to $\lceil \lg n \rceil$; if processor index $p/2 < i \leq 3 \cdot (p/4)$, then the input items are obtained by setting the items at processor p_i to $\lceil \lg n \rceil - 1$, and so forth; if processor index $i = p - 1$, then the input items are obtained by setting the first $\lceil m/(2 \cdot p) \rceil$ items at processor p_i to $\lceil \lg(m/p) \rceil$, the next $\lceil m/(4 \cdot p) \rceil$ to $\lceil \lg(m/p) \rceil - 1$, and so forth.

For a discussion of the practicality and suitability of this data set benchmarks refer to [108].

- Considerable effort has also been invested in dispensing of data set benchmarks that would invariantly favor our implementations and no algorithm has been optimized with respect to these benchmarks.

Concerning the CPU times that are either explicitly given or presented in a graphical exhibit, the following remarks apply.

- All CPU times are given in seconds on two machines:
 - (1) A *Silicon Graphics* (SGI) *Power Challenge* shared memory system, consisting of four 75MHz TFP 64-bit RISC processors, for a total peak capacity of 1.2Gflops with 512Mbytes main memory.
 - (2) A *Cray T3D* distributed memory system, consisting of 512 150MHz DECchip 21064 Alpha processors, for a total peak capacity of 76.8Gflops with 64Mbytes main memory (per processor).¹⁸
- All software (unless otherwise indicated) has been written in C – by the author – and has been linked with the Oxford BSP toolset, *BSPlib* version 0.81 α . The source code has been compiled

¹⁸For our experimental results we were provided with 32 processors in on-line mode and 128 processors in off-line mode.

using `cc` with option `-O3` under the operating systems *IRIX 6.0.1* and *UNICOS 8.0.4* on the SGI Power Challenge and Cray T3D systems respectively.

- The *BSPlib* function `bsp_time` has been used for measuring running times.

For each test problem instance, a series of experiments has been performed, and the running times indicated reflect the worst-case performance of these experiments.

Chapter 2

Fundamental Algorithms

2.1 Primitive Operations

The design of a complex algorithm relies on a set of primitive operations and the instruments required to compile these operations into an algorithm. In this section, we examine some of these basic primitive operations and present algorithms that are suitable for the BSP model. In particular, we consider algorithms for the following primitive operations.

- (1) n -word broadcasting.
- (2) n -item parallel-prefix.
- (3) n -item segmented parallel-prefix.
- (4) Load-balancing.

The material of this section is based on the following papers:

- (1) “Portable and scaleable algorithm design on the IBM SP2: The bulk-synchronous parallel paradigm.” In *Proceedings of the Sup-Eur’96 Conference*, Krakow, Poland, September 1996 [11].
- (2) “Primitive operations on the BSP model.” Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996 [82].

2.1.1 Broadcasting and Parallel-Prefix

In this section we design and analyze BSP algorithms for two fundamental operations, namely *broadcasting* and *parallel-prefix* (along with segmented parallel-prefix). These primitive operations are auxiliary routines for the more complicated algorithms described in later sections.

2.1.1.1 A Broadcasting Algorithm

The *broadcast* operation is a fundamental primitive in many parallel applications. In this section we derive an efficient BSP algorithm for a generalization of the single-item broadcast operation. In particular, if X_i is a vector of size n , such that X_i resides on processor p_{i-1} and $X_i = (x_1, \dots, x_n)$, $1 \leq i \leq p$, then the broadcast operation is defined to give the sequence $\langle X_1, X_2, \dots, X_p \rangle$, where $X_j = X_i$, for all $1 \leq j \leq p$.

Proposition 2.1.1 *There exists a p -processor BSP algorithm for broadcasting an n -word vector, that for any integer t such that $2 \leq t \leq p$, requires time at most¹*

$$M_{bcast}^n(p) = \left(L \cdot 2 + g \cdot \left(2 \cdot (p-1) \cdot \left\lceil \frac{n}{p} \right\rceil \right) \right) \cdot [n \geq p] + \left(L \cdot (2 \cdot h - 1) + g \cdot \left((2 \cdot h - 1) \cdot t \cdot \left\lceil \frac{n}{h} \right\rceil \right) \right) \cdot [n < p],$$

for communication, where $h = \lceil \log_t ((t-1) \cdot p + 1) \rceil - 1$. The overall time complexity of the algorithm is denoted by $T_{bcast}^n(p) = M_{bcast}^n(p)$.

Proof Consider an n -word vector $X = (x_1, \dots, x_n)$. Without loss of generality we assume that X resides in the local memory of processor p_0 . The procedure to manage the broadcast operation follows. We distinguish two cases.

Case 1 If $n \geq p$, then processor p_0 sends $\lceil n/p \rceil$ -word vector $(x_{(j-1)\lceil n/p \rceil+1}, \dots, x_{j\lceil n/p \rceil})$ to processor p_{j-1} , for all $1 \leq j < p$. Subsequently, each processor p_{j-1} , sends the vector it received in the previous step to all other processors. This process requires $L \cdot 2 + g \cdot (2 \cdot (p-1) \cdot \lceil n/p \rceil)$ communication time.

Case 2 If $n < p$, then processor p_0 employs a pipelined t -ary tree, consisting of p nodes, for some appropriate t . The depth of the tree is $h = \lceil \log_t ((t-1) \cdot p + 1) \rceil - 1$. In phase l , processor p_{j-1} , for all $1 \leq j \leq t^{l-1} + 1$, sends t copies of the $(l - \lceil (j-1)/t \rceil)$ -th component of size $\lceil n/h \rceil$ of X , that is vector $(x_{(l-\lceil (j-1)/t \rceil) \cdot \lceil n/h \rceil+1}, \dots, x_{(l-\lceil (j-1)/t \rceil+1) \cdot \lceil n/h \rceil})$, to processors $p_{(j-1) \cdot t+1}, \dots, p_{j \cdot t}$. This process completes within $h + \lceil n/\lceil n/h \rceil \rceil - 1$ phases and each such phase requires $L + g \cdot \lceil n/h \rceil$ communication time.

The proposition follows. $\square$

The following result involves a rather more complicated algorithm and improves upon the previous result for the case $1 < n < p$. A similar result has been obtained independently and subsequently to this work in [122].

Proposition 2.1.2 *There exists a p -processor BSP algorithm for broadcasting an n -word vector, that for any integer t such that $2 \leq t \leq p$, requires time at most*

$$M_{bcast}^n(p) = \left(L \cdot 2 + g \cdot \left(2 \cdot (p-1) \cdot \left\lceil \frac{n}{p} \right\rceil \right) \right) \cdot [n > 1] + (L \cdot h + g \cdot (2 \cdot h \cdot t)) \cdot [n < p],$$

for communication, where $h = \lceil \log_t ((t-1) \cdot \lceil p/n \rceil + 1) \rceil - 1$. The overall time complexity of the algorithm is denoted by $T_{bcast}^n(p) = M_{bcast}^n(p)$.

Proof Consider an n -word vector $X = (x_1, \dots, x_n)$. We number processors in a mesh-like fashion, so that we can refer to processor at row i and column j , as $p_{i,j}$, $1 \leq i \leq R_r$ and $1 \leq j \leq R_c$, where $R_r = \lceil p/n \rceil$ and $R_c = \lceil p/R_r \rceil$. Without loss of generality we assume that X resides in the local memory of processor $p_{1,1}$. The procedure to manage the broadcast operation follows.

Procedure BROADCAST

begin

¹In general, if S is any statement that can be true or false, the bracketed notation $[S]$ evaluates to 1 if S is true, and 0 otherwise.

[Stage 1] Processor $p_{1,1}$ communicates $\lceil n/R_c \rceil$ -word vector $(x_{(j-1)\lceil n/R_c \rceil+1}, \dots, x_{j\lceil n/R_c \rceil})$ to processor $p_{1,j}$, for all $1 \leq j \leq R_c$.

[Analysis of Stage 1] This stage requires at most $L + g \cdot ((R_c - 1) \cdot \lceil n/R_c \rceil)$ communication time.

[Stage 2] Each processor $p_{1,j}$, for all for all $1 \leq j \leq R_c$, communicates the $\lceil n/R_c \rceil$ -word vector it received in the previous stage to processor $p_{i,j}$, for all $1 \leq i \leq R_r$.

[Analysis of Stage 2] This stage is realized by utilizing a t -ary tree, rooted at $p_{1,j}$ and consisting of R_r nodes, for some appropriate t . In the first phase, processor $p_{1,j}$, for all $1 \leq j \leq R_c$, sends t copies of the $\lceil n/R_c \rceil$ -word vector it received in stage 1 to processors $p_{2,j}, \dots, p_{t+1,j}$. Subsequently, at phase $l > 1$ processors $p_{i,j}$, for all $(t^{l-1} - 1)/(t - 1) + 1 \leq i \leq (t^l - 1)/(t - 1)$, sends t copies of the $\lceil n/R_c \rceil$ -word vector it received in the previous phase to processors $p_{(i-1)t+2,j}, \dots, p_{it+1,j}$. This process completes within $h = \lceil \log_t ((t - 1) \cdot R_r + 1) \rceil - 1$ phases, and therefore, takes at most $L \cdot h + g \cdot (h \cdot t \cdot \lceil n/R_c \rceil)$ communication time.

[Stage 3] Each processor $p_{i,j}$, for all $1 \leq i \leq R_r$ and $1 \leq j \leq R_c$, sends the $\lceil n/R_c \rceil$ -word vector it received in stage 2 to processors $p_{i,k}$, for all $1 \leq k \leq R_c$.

[Analysis of Stage 3] This stage requires at most $L + g \cdot ((R_c - 1) \cdot \lceil n/R_c \rceil)$ communication time.

end

Analysis The correctness of the algorithm follows from its description and is omitted. To derive the time complexity of procedure BROADCAST we distinguish three cases.

Case 1 If $n \geq p$, then $R_r = 1$ and $R_c = p$, and stage 2 is redundant. Thus, the time complexity of algorithm BROADCAST for this case is bounded above by $L \cdot 2 + g \cdot (2 \cdot (p - 1) \cdot \lceil n/p \rceil)$.

Case 2 If $n = 1$, then $R_r = p$ and $R_c = 1$, and stages 1 and 3 are redundant. Thus, the time complexity of algorithm BROADCAST for this case is bounded above by $L \cdot h + g \cdot (2 \cdot h \cdot t)$.

Case 3 If $1 < n < p$, then $R_r = \lceil p/n \rceil$ and $R_c = \lceil p/R_r \rceil$, and therefore, the time complexity of algorithm BROADCAST for this case is bounded above by the combined time complexities of cases 1 and 2.

The proposition follows. $\square$

2.1.1.2 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the broadcasting algorithm (BROADCAST) of the previous section.

Implementation The prototype adopted for the *BSPlib* broadcasting procedure is outlined in figure 2.1.

The aim of function (procedure) `BSP_broadcast` is to provide a data-type independent high-performance *BSPlib* broadcasting algorithm implementation.

```

void BSP_broadcast (byte *src, byte **trg, int n, int sb) ;
    src    - input array.
    trg    - output array.
    n      - input size.
    sb     - item size.

```

Figure 2.1: Procedure for BSP broadcasting.

2.1.1.3 A Parallel-Prefix Algorithm

The *parallel-prefix* operation [140, 145] is a powerful primitive operation with many parallel applications [38]. In this section we derive an efficient BSP algorithm for a generalization of the prefix operation – n distinct prefixes each consisting of p items. In particular, if $\langle X_1, X_2, \dots, X_p \rangle$ is a sequence of p vectors each of size n , such that X_i resides on processor p_{i-1} and $X_i = (x_{1,i}, \dots, x_{n,i})$, $1 \leq i \leq p$, then the prefix operation on sequence $\langle X_1, X_2, \dots, X_p \rangle$, over an associative operator $\otimes$, is defined to give the sequence $\langle X_1, X_1 \otimes X_2, \dots, X_1 \otimes X_2 \otimes \dots \otimes X_p \rangle$, where $X_i \otimes X_{i+1} = (x_{1,i} \otimes x_{1,i+1}, \dots, x_{n,i} \otimes x_{n,i+1})$, $1 \leq i < p$.

Proposition 2.1.3 *There exists a p -processor BSP algorithm for realizing n disjoint parallel-prefix operations, each of size p , over an associative operator $\otimes$, that for any integer t such that $2 \leq t \leq p$, requires time at most*

$$C_{ppf}^n(p, \otimes) = \left(c_{\otimes} \cdot (p-1) \cdot \left\lceil \frac{n}{p} \right\rceil \right) \cdot [n \geq p] + \left((4 \cdot h - 2) \cdot c_{\otimes} \cdot t \cdot \left\lceil \frac{n}{h} \right\rceil \right) \cdot [n < p]$$

and

$$M_{ppf}^n(p, \otimes) = \left(L \cdot 2 + g \cdot O \left(2 \cdot (p-1) \cdot \left\lceil \frac{n}{p} \right\rceil \right) \right) \cdot [n \geq p] + \left(L \cdot (4 \cdot h - 2) + g \cdot O \left((4 \cdot h - 2) \cdot t \cdot \left\lceil \frac{n}{h} \right\rceil \right) \right) \cdot [n < p],$$

for computation and communication respectively, where $h = \lceil \log_t p \rceil$ and $c_{\otimes}$ is the time required to evaluate $\otimes$. The overall time complexity of the algorithm is denoted by $T_{ppf}^n(p, \otimes) = C_{ppf}^n(p, \otimes) + M_{ppf}^n(p, \otimes)$.

Proof The desired result follows by way of arguments similar to that of proposition 2.1.2; the difference being that the t -ary tree utilized in the analysis of case 2 consists of p leaf nodes and is of depth $h = \lceil \log_t p \rceil$. $\square$

The following result involves a rather more complicated algorithm and improves upon the previous result for the case $1 < n < p$. A similar result has been obtained independently and subsequently to this work in [122].

Proposition 2.1.4 *There exists a p -processor BSP algorithm for realizing n disjoint parallel-prefix operations, each of size p , over an associative operator $\otimes$, that for any integer t such that $2 \leq t \leq p$, requires time at most*

$$C_{ppf}^n(p, \otimes) = \left(2 \cdot c_{\otimes} \cdot (p-1) \cdot \left\lceil \frac{n}{p} \right\rceil \right) \cdot [n > 1] + (4 \cdot c_{\otimes} \cdot h \cdot (t-1)) \cdot [n < p],$$

and

$$M_{ppf}^n(p, \otimes) = \left(L \cdot 2 + g \cdot O \left((p-1) \cdot \left\lceil \frac{n}{p} \right\rceil \right) \right) \cdot [n > 1] + (L \cdot (2 \cdot h) + g \cdot O(h \cdot t)) \cdot [n < p],$$

for computation and communication respectively, where $h = \lceil \log_t \lceil p/n \rceil \rceil$ and $c_\otimes$ is the time required to evaluate $\otimes$. The overall time complexity of the algorithm is denoted by $T_{ppf}^n(p, \otimes) = C_{ppf}^n(p, \otimes) + M_{ppf}^n(p, \otimes)$.

Proof Consider an n -word vector $X_{i,j} = (x_{1,i,j}, \dots, x_{n,i,j})$. We number processors in a mesh-like fashion, so that we can refer to processor at row i and column j , as $p_{i,j}$, $1 \leq i \leq R_r$ and $1 \leq j \leq R_c$, where $R_r = \lceil p/n \rceil$ and $R_c = \lceil p/R_r \rceil$. Without loss of generality we assume that vector $X_{i,j}$ resides in the local memory of processor $p_{i,j}$. Our goal is to compute the parallel-prefix of the $X_{i,j}$'s in row-major direction – i.e., the parallel prefix of $\langle Y_1, \dots, Y_p \rangle$, where $Y_{(i-1)R_c+j} = X_{i,j}$. The procedure to manage the parallel-prefix operation follows.

Procedure PREFIX

begin

[**Stage 1**] Each processor $p_{i,j}$ communicates vector $(x_{(k-1)\lceil n/R_c \rceil+1,i,j}, \dots, x_{k\lceil n/R_c \rceil,i,j})$ to processors $p_{i,k}$, for all $1 \leq k \leq R_c$.

[**Analysis of Stage 1**] This stage requires $L + g \cdot O((R_c - 1) \cdot \lceil n/R_c \rceil)$ communication time. Note that each processor receives R_c disjoint $\lceil n/R_c \rceil$ -word vectors, and consequently, processor $p_{i,j}$, for all $1 \leq i \leq R_r$, receives the vectors corresponding to $\lceil n/R_c \rceil$ disjoint parallel-prefix operations.

[**Stage 2**] Each processor $p_{i,j}$, for all $1 \leq i \leq R_r$ and $1 \leq j \leq R_c$, computes locally the $\lceil n/R_c \rceil$ disjoint prefixes of size R_c it received in stage 1.

[**Analysis of Stage 2**] This stage takes time at most $c_\otimes \cdot (R_c - 1) \cdot \lceil n/R_c \rceil$.

[**Stage 3**] For each $1 \leq j \leq R_c$, processor $p_{i,j}$, for all $1 \leq i \leq R_r$, realizes $\lceil n/R_c \rceil$ disjoint parallel-prefix operations on the results of stage 2.

[**Analysis of Stage 3**] This stage is realized by processors $p_{i,j}$, for all $1 \leq i \leq R_r$, by utilizing a t -ary tree, rooted at $p_{1,j}$ and consisting of R_r leaf nodes, for some appropriate t , and proceeding in two phases, first moving from the leaves to the root, and then back to the leaves, performing $t - 1$ computations at each node [140]. The depth of the t -ary tree is $h = \lceil \log_t R_r \rceil$, and therefore, this stage requires time $2 \cdot c_\otimes \cdot h \cdot (t - 1) \cdot \lceil n/R_c \rceil$ and $L \cdot (2 \cdot h) + g \cdot O(2 \cdot h \cdot t \cdot \lceil n/R_c \rceil)$, for computation and communication respectively.

[**Stage 4**] Each processor $p_{i,j}$, for all $1 \leq i \leq R_r$ and $1 \leq j \leq R_c$, uses the results of stage 3 to complete the computation of the $\lceil n/R_c \rceil$ disjoint prefixes of size R_c it received in stage 1.

[**Analysis of Stage 4**] This stage is complementary to stage 2, and thus, takes time $c_\otimes \cdot (R_c - 1) \cdot \lceil n/R_c \rceil$.

[**Stage 5**] Each processor $p_{i,j}$, for all $1 \leq i \leq R_r$ and $1 \leq j \leq R_c$, communicates the results of the parallel-prefix operation to their target (original) position.

[**Analysis of Stage 5**] This stage is analogous to stage 1, and thus, takes $L + g \cdot O((R_c - 1) \cdot \lceil n/R_c \rceil)$ communication time.

end

Analysis The correctness of the algorithm follows from its description. The time complexity analysis of procedure PREFIX is along the lines of procedure BROADCAST (refer to proposition 2.1.1.1) and is omitted.

The proposition follows. $\square$

Remark If $c_{\otimes} = O(1)$, then we write $T_{ppf}^n(p)$ for $T_{ppf}^n(p, \otimes)$. We also write $T_{ppf}(p, \otimes)$ for $T_{ppf}^1(p, \otimes)$. The same notational convention applies to all pipelined operations.

2.1.1.4 A Segmented Parallel-Prefix Algorithm

The *segmented parallel-prefix* operation [145] consists of a sequence of prefix computations over the same associative operator $\otimes$ on k disjoint sets of total size p . As shown in [145], to solve the segmented prefix problem – along the lines of the standard prefix problem – the k disjoint input sets are first concatenated into a single set, inserting a *barrier* at the start of each set of data. Next, an associative operator $\bar{\otimes}$ is defined that respects the barriers, but otherwise performs identically to the original operator $\otimes$. This augmented parallel-prefix operation is then realized as the standard parallel-prefix operation. We therefore obtain that the time required to realize n disjoint segmented parallel-prefix operations, each on k disjoint sets of total size p , over an augmented associative operator $\bar{\otimes}$ is at most

$$T_{seg-ppf}^n(p, \bar{\otimes}) = T_{ppf}^n(p, \bar{\otimes}).$$

2.1.1.5 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the parallel-prefix algorithm (PREFIX) of the previous section.

Implementation The prototype adopted for the *BSPlib* parallel-prefix procedure is outlined in figure 2.2.

```
void BSP_prefix (byte *src, byte **trg, int n, int sb, void (*operator)()) ;
    src          - input array.
    trg          - output array.
    n            - input size.
    sb           - item size.
    operator     - binary operator.
```

Figure 2.2: Procedure for BSP parallel-prefix.

The aim of function (procedure) `BSP_prefix` is to provide a data-type independent high-performance *BSPlib* parallel-prefix algorithm implementation.

2.1.2 Load-Balancing

In this section we design and analyze a BSP algorithm for a fundamental data-redistribution operation, namely *load-balancing*. This primitive operation is an auxiliary routine for the more complicated algorithms described in later sections.

2.1.2.1 A Load-Balancing Algorithm

Let multi-set $X = \{x_1, \dots, x_n\}$ be arbitrarily distributed amongst p processors such that no processor possesses more than $m \leq n$ items of X . We say that the *maximum load* of multi-set X is m . In this section we consider the problem of redistributing (*load-balancing*) the items of X amongst the p processors, so that after the redistribution operation no processor possesses more than $\lceil n/p \rceil$ items of X .

Proposition 2.1.5 *There exists a p -processor BSP algorithm for load-balancing a multi-set of size n and of maximum load m , that for all integers n and p such that $n \geq p$, requires time at most*

$$C_{lbal}(n, m, p) = O\left(\frac{m \cdot p}{n}\right) + C_{ppf}(p)$$

and

$$M_{lbal}(n, m, p) = L + g \cdot O(m) + M_{ppf}(p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{lbal}(n, m, p) = C_{lbal}(n, m, p) + M_{lbal}(n, m, p)$.

Proof Consider a multi-set of size n initially distributed amongst p processors such that no processor possesses more than $m \leq n$ items. Let n_i , $0 \leq i < p$, denote the number of items residing in processor p_i . Initially, a parallel-prefix sum operation is realized on the n_i 's. This step takes time $T_{ppf}(p)$. Each processor p_i , using the information of the previous step, routes the items in its possession to their final destination (no more than $\lceil n/p \rceil$ per processor). This step can be easily shown to require time $O((m \cdot p)/n)$ and $L + g \cdot m$, for computation and communication respectively. The proposition follows. $\square$

2.1.2.2 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the load-balancing algorithm of the previous section.

Implementation The prototype adopted for the *BSPlib* load-balancing procedure is outlined in figure 2.3.

The aim of function (procedure) `BSP_load_balance` is to provide a data-type independent (generic) high-performance *BSPlib* load-balancing algorithm implementation.

```

void BSP_load_balance (byte *src, int **trg, int n, int *m, int sb) ;
  src    - input array.
  trg    - output array.
  n      - input size.
  m      - output size.
  sb     - item size.

```

Figure 2.3: Procedure for BSP load-balancing.

2.2 List-Contraction

In this section we design, analyze and implement a BSP algorithm for a fundamental list operation, namely *list-contraction*.² Parallel list-contraction [212] is a frequently recurring operation in computer science and is fundamental for many parallel applications, such as computing the prefix-sum for some associative operator over a linked list, determining the pre-order ordering of nodes on trees and evaluating expressions on trees [119, 146, 182]. The material of this section is based on the following paper:

- (1) “Parallel priority queues and list contraction: The BSP approach.” In *Proceedings of Euro-Par’97 Parallel Processing, Lecture Notes in Computer Science*, Springer-Verlag, August 1997 [88].

2.2.1 Preliminaries and Notation

Let $X = \langle x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n \rightarrow \perp \rangle$ be a linked list. We consider, in this section, the problem of *list-contracting* list X ; that is, making all its items point to $\perp$ by means of pointer-jumping [212]. By pointer-jumping we imply the modification of the successor pointer $\rightarrow x_{i+1}$ of some list item x_i to $\rightarrow x_{i+2}$.³ We note that the solution we propose in the remainder of this section to the list-contraction problem trivially gives a solution to the well-known list-ranking problem [212].

2.2.2 Cost Model

The performance of the BSP list-contraction algorithm of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations. Furthermore, constant time $O(1)$ is charged for each pointer-jumping operation in the modification process of the successor pointer.

Simple optimal $\Theta(n)$ time sequential list-contraction algorithms are straightforward to design and implement. In the following sections, we design, analyze and implement a simple optimal BSP list-contraction algorithm – optimality is realized for an ample range of the BSP parameters.

2.2.3 Related Work and New Results

There has been a significant amount of previous work – mainly on the PRAM model – on parallel list-contraction (e.g., [13, 46, 48, 135, 160, 212] on the PRAM model). The exhibit in table 2.1 summarizes the

²The list-contraction and list-ranking operations are rather similar, and therefore, list-contraction algorithms can (in general) be trivially modified to list-ranking algorithms.

³In the discussion of this section we assume that $x_i = x_i$ if $i \leq n$, and $x_i = \perp$ otherwise.

aforementioned results with respect to the following statistics: computation time, communication time, number of pointer-jumping phases,⁴ multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality.

Refs	Computation	Communication	Phases	Constants	Implem.	Procs
[212]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg n\right)$	$\lceil \lg n \rceil$	■	■	$O(n)$
[135]	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	$O(\lg p \cdot \frac{\lg n}{\lg(n/p)})$	■ ■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[46]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	$O(\lg n \cdot \lg^* n)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg n \cdot \lg^* n}\right)$
[48]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	$O(\lg n)$	■ ■ ■ ■	■ ■ ■ ■	$O\left(\frac{n}{\lg n}\right)$
[160]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	$O(\lg n)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[13]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	$O(\lg n)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg n}\right)$
Sec 2.2.4	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	$\lceil \lg n \rceil + o(\lg n)$	■ ■	■ ■	$O\left(\frac{n}{\lg n}\right)$

Table 2.1: Performance comparison of parallel list-contraction algorithms – the cyan color is associated with randomized approaches.

With reference to table 2.1 we note that the optimal deterministic PRAM algorithm of [48] employs expander graphs, and therefore, its practical viability is rather limited. Furthermore, the randomized PRAM algorithms of [13, 160] do not migrate efficiently to the BSP model as a result of their inherent requirement for a uniformly accessible global shared memory space (PRAM style memory) – as opposed to a non-uniformly accessible distributed memory space (BSP style memory).

The optimal BSP list-contraction algorithm of section 2.2.4 improves upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [135, 212],
- (2) the asymptotic communication time requirements of [135, 212],
- (3) the number of pointer-jumping phases of [13, 46, 48, 135, 160],
- (4) the multiplicative constant factor requirements of [13, 46, 48, 160],
- (5) the implementation effort requirements of [48], and
- (6) the parallel slackness requirements (to achieve optimality) of [46, 135, 212].

The principal contribution, however, of the list-contraction algorithm of section 2.2.4 is attributed to the *simplicity* of the underlying BSP algorithm (refer to case 5) and the *small number of pointer-jumping phases* (refer to case 3) that ultimately results in low synchronization cost.

2.2.4 An Optimal Randomized List-Contraction Algorithm

In this section we present a randomized BSP algorithm for the problem of contracting a list of n items. The algorithm is based on *pointer-jumping* techniques [212] and employs a deterministic “local” load-balancing scheme to achieve efficiency. The algorithm achieves optimality for an ample range of the parameters n , p , L and g .

⁴The number of pointer-jumping phases gives a rather accurate indication of the number of supersteps in the underlying algorithm.

Informal Description of Algorithm The optimal randomized list contraction algorithm is based on pointer-jumping techniques and is iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. In each iteration of the algorithm, processors assign a random sex, either male or female, to each non-contracted item in their possession and contract, by means of pointer-jumping techniques, the items whose sex is male and their successor's sex is female. Subsequently, the processors are partitioned into an appropriate number of groups and the non-contracted items are load-balanced within each group. The aforementioned steps secure that a constant proportion of the non-contracted items of each processor is contracted per iteration of the algorithm. Finally, after a pre-specified number of iterations the iterative process is halted and the remaining non-contracted items are contracted, by means of a sub-optimal list contraction algorithm.

Our main result is summarized in the following theorem.

Theorem 2.2.1 *For any constant $\rho > 0$, there exists a p -processor BSP algorithm for contracting a list of size n , that for all integers n and p such that $p = O(n/\lg n)$, requires time [with probability $1 - O(n^{-\rho})$] $O(n/p)$ and $L \cdot (\lceil \lg n \rceil + O(\lg \lg^2 n)) + g \cdot O(n/p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the following proposition. Then, theorem 2.2.1 follows easily as a consequence of this result.

Proposition 2.2.1 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for contracting a list of size n , that for all integers n and p such that $p = O(n/\lg n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$C_{lcntr}(n, p) = O\left(\frac{n}{p}\right) + \sum_{j=1}^{\log_t \lg n} C_{lbal}\left(O\left(\frac{n}{p}\right), O\left(\frac{1}{t^j} \cdot \frac{n}{p}\right), t^j\right)$$

and

$$M_{lcntr}(n, p) = L \cdot (\lceil \lg n \rceil + O(\log_t \lg n)) + g \cdot O\left(\frac{n}{p}\right) + \sum_{j=1}^{\log_t \lg n} M_{lbal}\left(O\left(\frac{n}{p}\right), O\left(\frac{1}{t^j} \cdot \frac{n}{p}\right), t^j\right),$$

for computation and communication respectively, where $t = (8/7)/(1 + o(1))$. The overall time complexity of the algorithm is denoted by $T_{lcntr}(n, p) = C_{lcntr}(n, p) + M_{lcntr}(n, p)$.

Proof The algorithm is iterative in nature and each iterative call is divided into a sequence of stages (in the subsequent analysis the j -th iterative call is denoted by the subscript notation $(_j)$). We assume that the input is evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm. Moreover, we initially assume that all items are marked as *active* and $j = 0$.

Procedure LIST_CONTRACT

begin

[Stage 1] If $j \geq \lg \lg n / \lg((8/7)/(1 + \varepsilon))$, for some $0 < \varepsilon < 1/7$ to be specified later, then the remaining active items are contracted by a sub-optimal parallel list-contraction algorithm, and subsequently, the algorithm terminates; otherwise $j = j + 1$.

[**Analysis of Stage 1**] Let $k_{(j)}$ denote the maximum number of active items per processor, over all processors, in the j -th iteration of the algorithm. The sub-optimal list-contraction process is realized by way of a straightforward BSP adaptation of the list-contraction algorithm of [212]. Then, it is trivial to verify that this stage requires overall time $O(k_{(\lceil \lg \lg n / \lg((8/7)/(1+\varepsilon)) \rceil)} \cdot \lg n$ and $L \cdot \lceil \lg n \rceil + g \cdot O(k_{(\lceil \lg \lg n / \lg((8/7)/(1+\varepsilon)) \rceil)} \cdot \lg n)$, for computation and communication respectively.

[**Stage 2**] Each processor assigns uniformly at random a sex, either *male* or *female*, to each active item in its possession.

[**Analysis of Stage 2**] This stage requires (per iteration) time $O(k_{(j)})$.

[**Stage 3**] Each processor deactivates the items in its possession whose sex is male and their successor's sex is female – no two adjacent items can be deactivated in this stage.

[**Analysis of Stage 3**] This stage requires (per iteration) time $O(k_{(j)})$ and $L + g \cdot O(k_{(j)})$, for computation and communication respectively.

[**Stage 4**] The active items are “locally” load-balanced amongst the processors – the term “locally” is justified in the forthcoming analysis.

[**Analysis of Stage 4**] The load-balancing computation in j -th iteration is scheduled as follows. The processors are partitioned into $\lceil ((8/7)/(1+\varepsilon))^{\log p - j} \rceil$ groups, each consisting of $q_{(j)} \leq ((8/7)/(1+\varepsilon))^j$ consecutively numbered processors. The active items within each group – and thus the term “locally” – are load-balanced. This stage requires (per iteration) time at most $T_{bal}(k_{(j)} \cdot q_{(j)}, k_{(j)}, q_{(j)})$.

end

Analysis The correctness of the algorithm follows from its description. We proceed by establishing an upper bound on the computation and communication requirements of procedure LIST_CONTRACT. We start by showing the following bound on the total number of active items $n_{(j)}$ in the j -th iterative call of the algorithm.

Claim 2.2.1 *Let m be the depth of the iteration of procedure LIST_CONTRACT. Then, for any $0 < \varepsilon < 1/7$,*

$$\Pr \left[n_{(j)} \leq (1 + \varepsilon) \cdot \frac{n_{(j-1)}}{(8/7)} \right] \geq 1 - e^{-(1/3) \cdot \varepsilon^2 \cdot n_{(j-1)}}, \quad \text{for all } 1 \leq j \leq m.$$

Proof Let $X_{(j)}$ denote the set of active items at (the beginning of) the j -th iteration of procedure LIST_CONTRACT. Obviously, $|X_{(j)}| = n_{(j)}$. Let x be an item in X_j . Then x is deactivated in the $j+1$ -th iteration if its sex is male and the sex of its successor is female. Hence, the probability that $x \in X_j$ is deactivated is equal to $1/4$. The items, however, are not stochastically independent of each other. For example, if an item is deactivated, then, with certainty, its successor is not deactivated in the same iteration. Consider set $\bar{X}_j \subseteq X_j$ consisting of every other item of X_j . Then, the distance between any two items of $\bar{X}_j$ is at least 2, and therefore, given two items in $\bar{X}_j$, the event that any of these two items is deactivated is independent of the event that another item is deactivated. We conclude that the probability that $x \in X_j$ is deactivated is at least $1/8$ (independently of the other items). Then, the desired probability bound follows by way of the inequality in theorem A.1.3. $\square$

Similarly, we obtain the following bound on the maximum number of active items $k_{(j)}$ per processor in the j -th iterative call of the algorithm.

Claim 2.2.2 *Let m be the depth of the iteration of procedure LIST_CONTRACT and consider integers n and p such that $p = O(n/\lg n)$. Then, for any constant $\rho > 0$,*

$$\Pr[k_{(j)} = O(\lceil n_{(j)}/p \rceil)] = 1 - O(n^{-\rho}), \text{ for all } 1 \leq j \leq m.$$

Proof The proof follows a line of arguments similar to that of claim 2.2.1, and derives by way of the load-balancing operation of stage 3 of procedure LIST_CONTRACT and the inequality in theorem A.1.2. $\square$

By combining claims 2.2.1 and 2.2.2 and summing up the computation and communication requirements of procedure LIST_CONTRACT, for $\varepsilon = n^{1/2-\zeta}$ (for arbitrarily small constant $0 < \zeta < 1/2$), we obtain the desired time and probability bounds.

Remark For the special case that $p = O(n/\lg^2 n)$, it is straightforward to verify that the time and probability bounds follow even at the absence of the load-balancing operation of stage 3 of procedure LIST_CONTRACT.

The proposition follows. $\square$

Proof (of theorem 2.2.1) The theorem follows easily from proposition 2.2.1 by substituting for T_{lbal} and T_{ppf} the corresponding expressions in propositions 2.1.5 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.2.4.1 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the list-contraction algorithm (LIST_CONTRACT) of the previous section.

Implementation The prototype adopted for the *BSPlib* list-contraction procedure is outlined in figure 2.4.

```
void BSP_list_contract (byte *src, int **trg, int n, int sb, void (*next)()) ;
    src      - input list.
    trg      - table of ranks.
    n        - input size.
    sb       - item size.
    next     - generic pointer-jumping function.
```

Figure 2.4: Procedure for BSP list-contraction.

The aim of function (procedure) `BSP_list_contract` is to provide a data-type independent *BSPlib* list-contraction algorithm implementation.⁵

⁵The terms generic and data-type independent imply that procedure `BSP_list_contract` can be employed to handle *any* data type. To this end, a next function `next` is used to facilitate the pointer jumping process on two elements e_1 and e_2 . In particular, the `next` function returns to e_1 the successor of e_2 .

Experimental Results The theoretical performance of the list-contraction algorithm is summarized in theorem 2.2.1. To determine the practical significance of the aforementioned result we benchmarked our implementation (`BSP_list_contract`) on two different machines with a varying number of processors and input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). Figures 2.5 and 2.7 present the execution time of procedure `BSP_list_contract` on the two machines. Similarly, figures 2.6 and 2.8 expose the efficiency of the corresponding timing results.

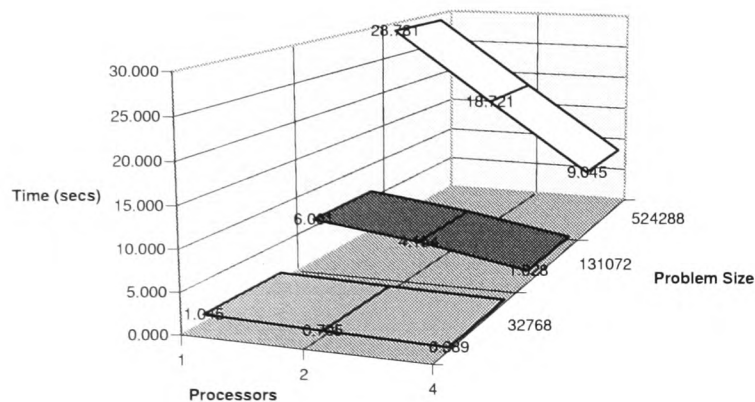


Figure 2.5: Execution time on a SGI Power Challenge for randomized list-contraction – uniform distribution (U).

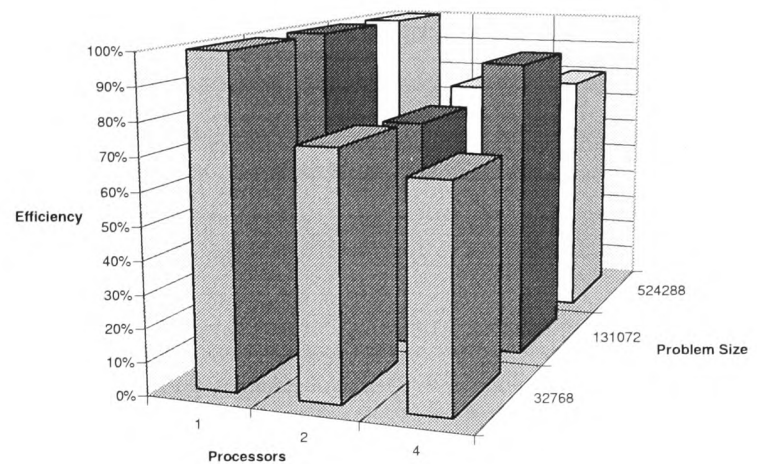


Figure 2.6: Efficiency on a SGI Power Challenge for randomized list-contraction – uniform distribution (U).

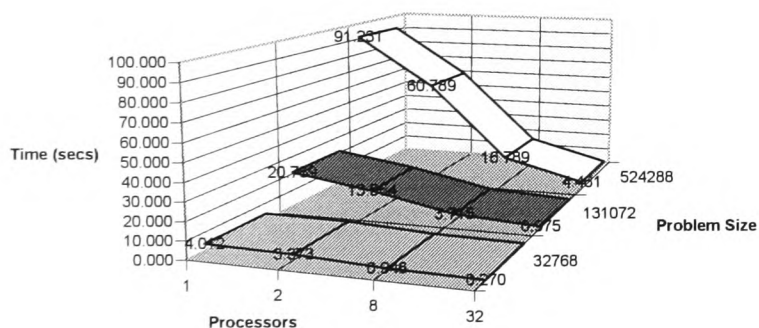


Figure 2.7: Execution time on a Cray T3D for randomized list-contraction - uniform distribution (U).

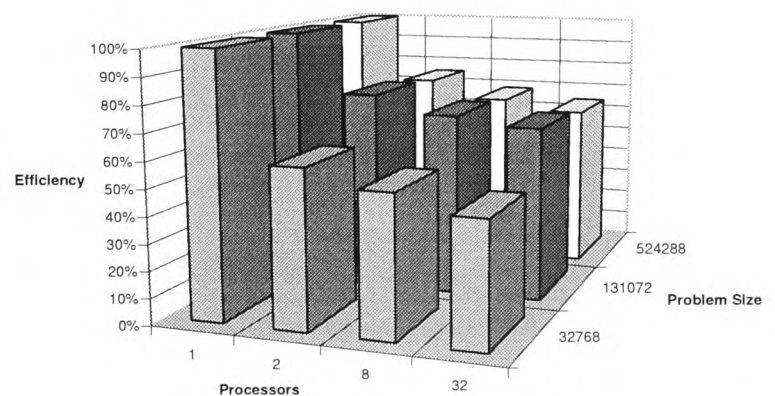


Figure 2.8: Efficiency on a Cray T3D for randomized list-contraction – uniform distribution (U).

As shown in figures 2.5 and 2.7, the execution time of procedure `BSP_list_contract` using a fixed number of processors scales with the input size (approximately as n). Accordingly, the execution time of procedure `BSP_list_contract` for a fixed input size scales inversely with the number of processors (approximately as $1/p$); nevertheless, the multiplicative constant factors introduced by the parallelization of the list-contraction algorithm are quite large, and therefore, linear in p speedup is not achievable. Both experimental results are in accordance with the theoretical analysis of section 2.2.4. In terms of efficiency, procedure `BSP_list_contract` utilizes approximately (within a 5%):

- 50%–65% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 75%–80% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),

- 40%–50% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 60%–75% of the available processor resources on the Cray T3D machine (for large problem sizes).

These results should be interpreted with caution on a twofold basis: (i) the one processor execution time and efficiency correspond to the one processor parallel version of the algorithm (we note that a highly optimized sequential list-contraction algorithm is expected to perform 10–100 times faster than the one processor parallel version of the algorithm; this slowdown is mainly attributed to the overheads introduced by communication and synchronization, along with the high multiplicative constant factors introduced by the parallelization of the list-contraction algorithm), and (ii) the list-contraction problem is communication bound, and therefore, efficiency close to one is not possible without the introduction of artificial conditions and premises.

2.3 Merging

In this section we design, analyze and implement BSP algorithms for a fundamental combinatorial operation, namely *merging*. Parallel merging [133] is a fundamental operation on ordered sequences and of great importance for many parallel applications, such as implementing database primitives, sorting and computational geometry algorithms [54, 133, 198]. The material of this section is based on the following paper:

- (1) “Primitive operations on the BSP model.” Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996 [82].

2.3.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_k \rangle$ be two non-decreasing sequences whose items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments. Without loss of generality, we assume that all the items appearing in sequences X and Y are distinct – if the items are not distinct we can make them so by appending to them their index. We consider, in this section, the problem of *merging* sequences X and Y into a non-decreasing sequence – say, $\langle z_1, z_2, \dots, z_{m+k} \rangle = \text{non-decreasing}(X \cup Y)$.

The following notation and definitions are assumed in the remainder of this section. Let X and Y be non-decreasing sequences, and let x_i be an item in X , $1 \leq i \leq m$. The *position* of item x_i in sequence X , denoted by $\text{pos}(x_i : X)$, is i . The *rank* of item x_i in sequence Y , denoted by $\text{rank}(x_i : Y)$, is the number of items of Y that are less than or equal to x_i . *Ranking* X in Y , amounts to computing the integer array $\text{rank}(X : Y) = (r_1, r_2, \dots, r_m)$, where $r_i = \text{rank}(x_i : Y)$, $1 \leq i \leq m$.

Let c be a positive integer. Sequence $\bar{X}$ is a c -sample of sequence X , denoted by $\text{smp}_c(X)$, if $\bar{X}$ consists of every c -th item of X .

2.3.2 Cost Model

The performance of the BSP merging algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator

in parallel-prefix computations. Furthermore, for some constant c_{mrg} , time $c_{mrg} \cdot n$ is charged for merging two non-decreasing sequences of combined size n – within the context of this section we denote c_{mrg} by c_* .

Optimal $\Theta(n)$ time sequential merging algorithms are well-known [54, 133]. In the following sections, we design, analyze and implement simple sub-optimal and rather more complicated optimal BSP merging algorithms – optimality is realized for an ample range of the BSP parameters.

2.3.3 Related Work and New Results

There has been a notable amount of previous work – mainly on the PRAM model – on parallel merging (e.g., [8, 60, 72, 105] on the PRAM model, [22] on the comparison network model). The exhibit in table 2.2 summarizes the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality.

Refs	Computation	Communication	Constants	Implem.	Procs
[22]	$O\left(\frac{n}{p} \cdot \lg p\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■	■	$O(n)$
[8]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg^2 n}\right)$
[72]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\sqrt{\frac{n}{\lg n}}\right)$
[105]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[60]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg^2 n}\right)$
Sec 2.3.5	$c_* \cdot \frac{n}{p} + o\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$

Table 2.2: Performance comparison of parallel merging algorithms.

With reference to table 2.2 we note that both the algorithm of [105] and the algorithm of section 2.3.5 derive (independently) from the work of [195], and therefore, utilize similar concepts and ideas.

The optimal BSP merging algorithm of section 2.3.5 improves upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [22],
- (2) the asymptotic communication time requirements of [22],
- (3) the multiplicative constant factor requirements of [8, 22, 60, 72, 105], and
- (4) the parallel slackness requirements (to achieve optimality) of [8, 22, 60, 72].

The principal contribution, however, of the merging algorithm of section 2.3.5 is attributed to the *small constant multiplicative factors* (refer to case 3), and the *low amount of parallel slackness* required to achieve optimality (refer to case 4). Additionally, our techniques provide *one-optimality* in computation for an ample range of the BSP parameters.

2.3.4 A Sub-Optimal Merging Algorithm

In this section we present a BSP adaptation of Batcher’s bitonic-sort algorithm [22]. Let $V = \langle v_1, \dots, v_n \rangle$ be a sequence of items drawn from a linearly ordered set. The sequence V is bitonic if, for some positive

integer $i \leq n$, we have $v_1 \geq \dots \geq v_i \leq v_{i+1} \leq \dots \leq v_n$. In [22] it is shown that there exists a sorting network of depth $\lceil \lg n \rceil$ and consisting of $n \cdot \lceil \lg n \rceil / 2$ comparators that correctly sorts V . The following result generalizes the construction of [22] by providing an efficient BSP implementation of Batcher's bitonic-sort algorithm.

Proposition 2.3.1 *There exists a p -processor BSP algorithm for merging two non-decreasing sequences of sizes m and k that for all integers n , p and t , such that $n = m + k$, $p = O(n)$ and $2 \leq t \leq p$, requires time at most*

$$C_{btn}(n, p) = \lceil \log_t p + 1 \rceil \cdot \left(c_* \cdot 2^{\lceil \lg t \rceil} \cdot \left\lceil \frac{n}{p} \right\rceil \right)$$

and

$$M_{btn}(n, p) = L \cdot \lceil \log_t p + 1 \rceil + g \cdot O \left(\lceil \log_t p + 1 \rceil \cdot 2^{\lceil \lg t \rceil - 1} \left\lceil \frac{n}{p} \right\rceil \right),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{btn}(n, p) = C_{btn}(n, p) + M_{btn}(n, p)$.

Proof Let X and Y be respectively the two input sequences. We assume that X and Y are evenly and orderly distributed among the p processors – item $x_i \in X$, for all $1 \leq i \leq m$, resides in the local memory of processor $p_{\lfloor i/q \rfloor}$ at position $i \bmod q$, and item $y_j \in Y$, for all $1 \leq j \leq k$, resides in the local memory of processor $p_{\lfloor (m+k-j+1)/q \rfloor}$ at position $(m+k-j+1) \bmod q$, where $q = \lceil (m+k)/p \rceil$. This distribution of items to processors results in an ordered sequence for each processor except (possibly) for processor $p_{\lfloor m/q \rfloor}$ that results in a bitonic sequence.

Let $W = \langle \langle w_{1,1}, \dots, w_{1,l} \rangle, \dots, \langle w_{i,1}, \dots, w_{i,l} \rangle, \dots, \langle w_{p,1}, \dots, w_{p,l} \rangle \rangle$ be a sequence of items such that $w_{i,1} \leq \dots \leq w_{i,l}$, for all $1 \leq i \leq p$, and sequence $\bar{W} = \langle w_{1,l}, \dots, w_{p,l} \rangle$ is bitonic. We next outline how to modify Batcher's bitonic-sort algorithm to obtain a p -processor BSP algorithm for sorting W . Each input to the bitonic-sort network is replaced by a non-decreasing sequence $w_{i,1} \leq \dots \leq w_{i,l}$ of size l , $1 \leq i \leq p$. Additionally, each comparator is replaced by a sequential merging algorithm that receives as input two non-decreasing sequences each of size l , merges them, and outputs separately the lower and upper halves of the merged sequences. As shown in [23, 133] the concatenation of the merged sequences at the output of the bitonic-sort network yields the desired sorted sequence. The performance of the aforementioned algorithm can be improved [122] by combining $\lceil \lg t \rceil$ consecutive levels of comparators of the bitonic-sort network into a single level, for some integer $2 \leq t \leq p$. The so constructed network has a total of at most $\lceil \lceil \lg p \rceil / \lceil \lg t \rceil \rceil$ levels.

To simulate this sorting network on a BSP machine, comparators at each level of the network are replaced by a pairs of processors. Each processor, therefore, communicates a total of at most $2^{\lceil \lg t \rceil - 1} \cdot l$ messages per level. Since the total number of levels is at most $\lceil \log_t p + 1 \rceil$ we obtain that the time complexity of the BSP algorithm is bounded above by $\lceil \log_t p + 1 \rceil \cdot (c_* \cdot 2^{\lceil \lg t \rceil} \cdot l)$ and $L \cdot \lceil \log_t p + 1 \rceil + g \cdot O(\lceil \log_t p + 1 \rceil \cdot 2^{\lceil \lg t \rceil - 1} \cdot l)$, for computation and communication respectively. By substituting in the above analysis $l = \lceil n/p \rceil$ we obtain the desired result. $\square$

2.3.4.1 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the sub-optimal merging algorithm of section 2.3.4.

Implementation The prototype adopted for the *BSPlib* merging procedure is outlined in figure 2.9.

```
void BSP_btn_merge (byte *src, byte **trg, int n1, int n2, int sb, void (*merge)(), int (*cmp)()) ;
    src1, src2    - input array.
    trg          - output array.
    n1, n2       - input sizes.
    sb           - item size.
    merge        - generic merge function.
    cmp          - generic compare function.
```

Figure 2.9: Procedure for BSP merging.

The aim of function (procedure) `BSP_btn_merge` is to provide a data-type independent high-performance *BSPlib* merging algorithm implementation.⁶

Experimental Results The theoretical performance of the merging algorithm is summarized in proposition 2.3.1. To determine the practical significance of the aforementioned result we benchmarked our implementation (`BSP_btn_merge`) on two different machines with a varying number of processors and input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). In fact, we employed procedure `BSP_btn_merge` to sort a sequence of integers and we benchmarked this sorting operation. Figures 2.10 and 2.12 present the execution time of procedure `BSP_btn_merge` (in the context of sorting) on the two machines. Similarly, figures 2.11 and 2.13 expose the efficiency of the corresponding timing results.

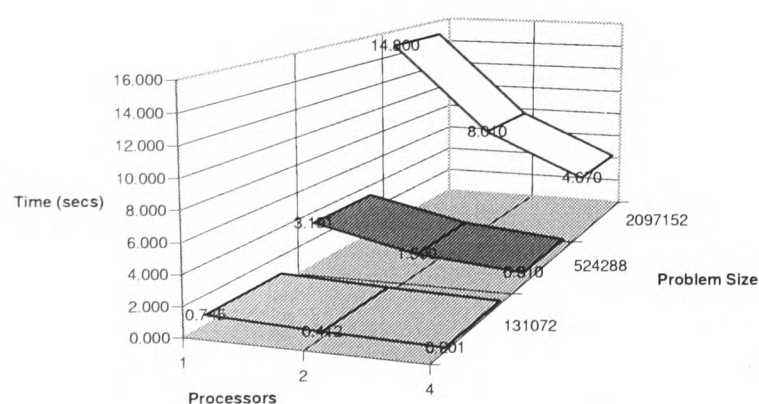


Figure 2.10: Execution time on a SGI Power Challenge for bitonic-sort sorting – uniform distribution (U).

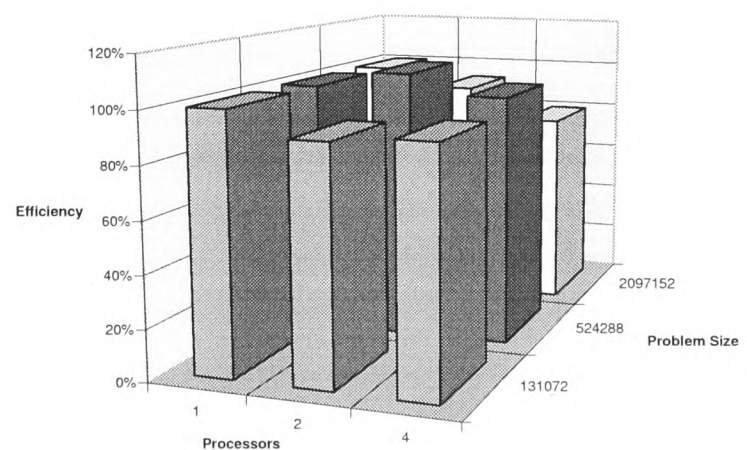


Figure 2.11: Efficiency on a SGI Power Challenge for bitonic-sort sorting – uniform distribution (U).

As shown in figures 2.10 and 2.12, the execution time of procedure `BSP_btn_merge` (in the context of sorting) using a fixed number of processors scales with the input size (approximately as $n \cdot \lg n$).

⁶The terms generic and data-type independent imply that procedure `BSP_btn_merge` can be employed to handle *any* data type. To this end, a comparison function `cmp` is used to facilitate the comparison of two elements e_1 and e_2 . In particular, the `cmp` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2 .

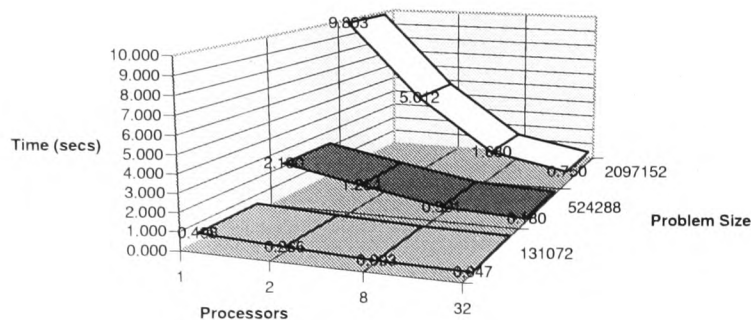


Figure 2.12: Execution time on a Cray T3D for bitonic-sort sorting – uniform distribution (U).

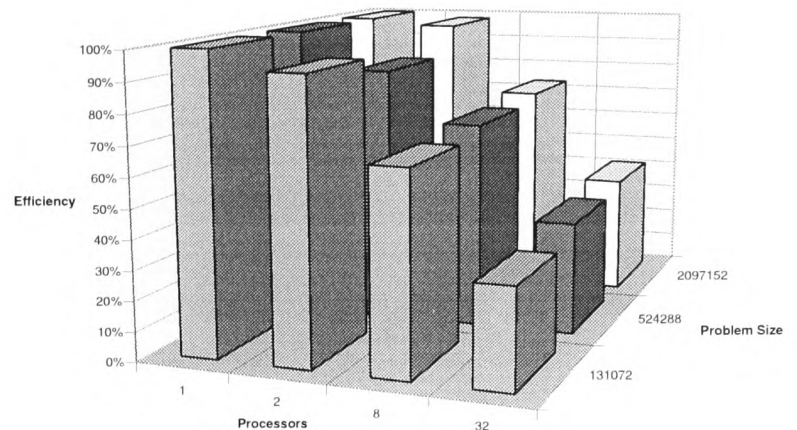


Figure 2.13: Efficiency on a Cray T3D for bitonic-sort sorting – uniform distribution (U).

Accordingly, the execution time of procedure `BSP_btn_merge` for a fixed input size scales inversely with the number of processors (approximately as $\lg p/p$). Both experimental results are in accordance with the theoretical analysis of section 2.3.4. In terms of efficiency, procedure `BSP_btn_merge` utilizes approximately (within a 5%):

- 85%–100% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 80%–90% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),
- 25%–85% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 40%–100% of the available processor resources on the Cray T3D machine (for large problem sizes).

These results should be interpreted with caution on the basis that the merging algorithm of section 2.3.4 is sub-optimal, and therefore, efficiency close to one is not possible for large number of processors without the introduction of artificial conditions and premises.

In order to test the obliviousness of the merging algorithm to the input data distribution we benchmarked our implementation (`BSP_btn_merge`) on a machine with a varying number of processors and input sets (consisting of 2^{21} integer valued items – unless otherwise indicated). Figure 2.14 presents the execution time of procedure `BSP_btn_merge` (in the context of sorting). Similarly, figure 2.15 exposes the efficiency of the corresponding timing result.

Obviously, the performance of the algorithm is quite consistent across the different input data distributions with the exception of the deterministic-duplicates distribution (we note that the observable inconsistency in procedure `BSP_btn_merge`, with respect to the deterministic-duplicates distribution, derives from the sequential merging implementation (`merge`) employed in the main body of `BSP_btn_merge` and is irrelevant to the otherwise structure of the procedure).

2.3.5 An Optimal Merging Algorithm

In this section, we provide an optimal BSP solution to the problem of merging two sequences that is based on *over-sampling* techniques [77, 92]. The basic algorithmic idea derives from a merging algorithm

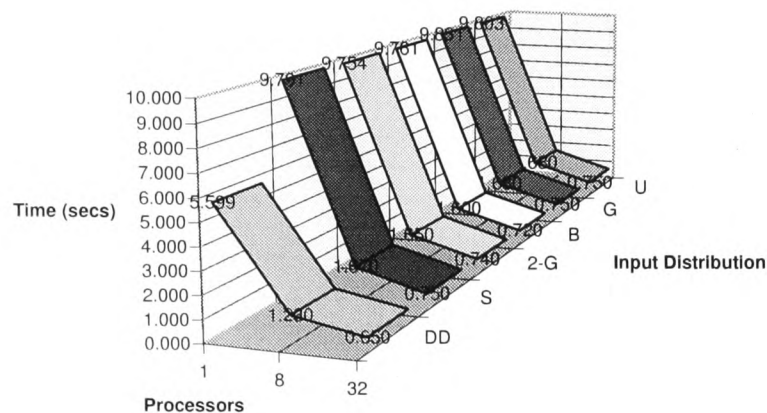


Figure 2.14: Execution time on a Cray T3D for bitonic-sort sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

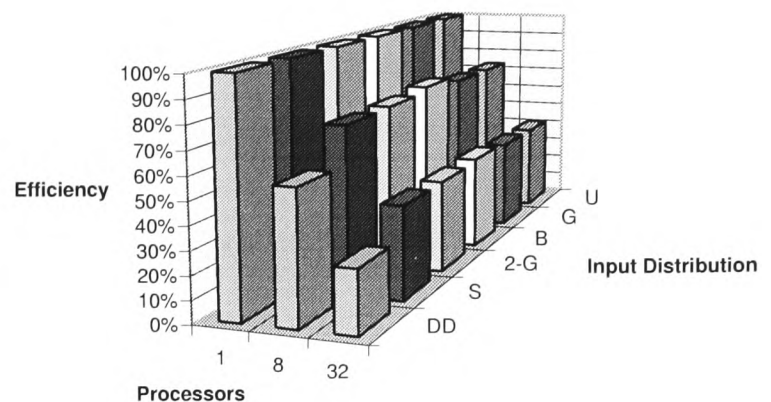


Figure 2.15: Efficiency on a Cray T3D for bitonic-sort sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

of [195]. The algorithm achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic merging algorithm is based on over-sampling techniques and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, two sub-sequences are constructed that are appropriate over-samples of the two input sequences. The two sub-sequences are merged, by means of the sub-optimal bitonic-sort algorithm of section 2.3.4, and a sub-sequence of size $p - 1$ is constructed that is an appropriate sample of the resulting merged sequence. Finally, this sub-sequence is employed to partition the two input sequences into p pairs of overlapping partitions that are sequentially and independently merged at each processor (the aforementioned partitioning operation is realized by means of an optimal binary search method that is based on a composite parallel-prefix scheme in order to avoid the inherent need for concurrent read).

Our main result is summarized in the following theorem.

Theorem 2.3.1 *There exists a p -processor BSP algorithm for merging two non-decreasing sequences of combined size n , that for all integers n and p such that $p = o(n/\lg n)$, requires time $(c_* + o(1)) \cdot (n/p)$ and $L \cdot O(\lg p) + g \cdot O(n/p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 2.3.1 follows easily as a consequence of this result.

Optimality Claim 2.3.1 *Theorem 2.3.1 implies that there exists a p -processor BSP algorithm for merging two non-decreasing sequences of combined size n , that for all integers n , p , L and g such that $p = o(n/\lg n)$ and $L = o(n/(p \cdot \lg p))$, has $\pi = 1 + o(1)$ and $\mu = O(g)$.*

Proposition 2.3.2 *Let ω_n be any function of n such that $\omega_n = \Omega(1)$ and $\omega_n = O(n/p)$. Then, there exists a p -processor BSP algorithm for merging two non-decreasing sequences of sizes m and k that for all integers n and p , such that $n = m + k$ and $p = O(n/\lg n)$, requires time*

$$C_{mrg}(n, p) = c_* \cdot \left(1 + O\left(\frac{1}{\omega_n}\right)\right) \cdot \frac{n}{p} + O\left(\omega_n + \lg \frac{n}{p \cdot \omega_n}\right) + C_{btn}(p \cdot \lceil \omega_n \rceil, p) + 2 \cdot C_{ppf}(p)$$

and

$$M_{mrg}(n, p) = L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + M_{btn}(p \cdot \lceil \omega_n \rceil, p) + 2 \cdot M_{ppf}(p) + M_{bcast}^{O(n/p)}(p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{mrg}(n, p) = C_{mrg}(n, p) + M_{mrg}(n, p)$.

Proof Let X and Y be respectively the two input sequences. We assume that X and Y are evenly and orderly distributed among the p processors – item $x_i \in X$, for all $1 \leq i \leq m$, resides in the local memory of processor $p_{\lfloor i/q \rfloor}$ at position $i \bmod q$, and item $y_j \in Y$, for all $1 \leq j \leq k$, resides in the local memory of processor $p_{\lfloor (m+j)/q \rfloor}$ at position $(m+j) \bmod q$, where $q = \lceil (m+k)/p \rceil$. We also assume that the items are distinct since we can always make them so by, for example, appending to them the code for their memory location. This assumption can be removed without affecting the asymptotic performance of our algorithm.

Procedure MERGE

begin

[**Stage 1**] Sub-sequences $\bar{X}$ and $\bar{Y}$ are constructed such that $\bar{X}$ is a $\lceil n/(p \cdot \omega_n) \rceil$ -sample of X and, similarly, $\bar{Y}$ is a $\lceil n/(p \cdot \omega_n) \rceil$ -sample of Y . For each item in $\bar{X}$ and $\bar{Y}$ information about whether it belongs to sequence X or Y along with its rank in that sequence is also maintained.

[**Analysis of Stage 1**] Each processor selects at most $\lceil \omega_n \rceil$ items from its local memory, and therefore, this stage takes time $O(\omega_n)$.

[**Stage 2**] Sub-sequences $\bar{X}$ and $\bar{Y}$ are merged into sub-sequence $\bar{Z}$. For each item in $\bar{Z}$ information about whether it belongs to sequence X or Y along with its rank in that sequence is maintained.

[**Analysis of Stage 2**] The merging operation is realized by employing the BSP bitonic-sort algorithm implied in proposition 2.3.1, for a total time of $T_{btn}(|\bar{X} \cup \bar{Y}|, p)$, where $|\bar{X} \cup \bar{Y}| \leq p \cdot \lceil \omega_n \rceil$.

[**Stage 3**] Sub-sequence $\hat{Z} = \langle \hat{z}_1, \dots, \hat{z}_{|\hat{Z}|} \rangle$ is constructed such that $\hat{Z}$ is a $\lceil \omega_n \rceil$ -sample of $\bar{Z} - \{\bar{z}_{|\bar{Z}|}\}$.

[**Analysis of Stage 3**] Since $|\bar{Z}| \leq p \cdot \lceil \omega_n \rceil$ we have that $|\hat{Z}| < p$, and therefore, each processor selects at most one item from its local memory. This stage requires time $O(1)$.

[**Stage 4**] Let $\hat{X} = \bar{X} \cap \hat{Z}$ and $\hat{Y} = \bar{Y} \cap \hat{Z}$. Integer arrays $R_{\hat{X}:X} = \text{rank}(\hat{X} : X)$, $R_{\hat{X}:Y} = \text{rank}(\hat{X} : Y)$, $R_{\hat{Y}:X} = \text{rank}(\hat{Y} : X)$ and $R_{\hat{Y}:Y} = \text{rank}(\hat{Y} : Y)$ are constructed. As a consequence, integer arrays $R_{\hat{Z}:X} = \text{rank}(\hat{Z} : X)$ and $R_{\hat{Z}:Y} = \text{rank}(\hat{Z} : Y)$ are obtained.

[**Analysis of Stage 4**] Integer arrays $R_{\hat{X}:X}$ and $R_{\hat{Y}:Y}$ can be constructed in $O(1)$ time since $\hat{X} \subset \bar{X} \subset X$ and $\hat{Y} \subset \bar{Y} \subset Y$. The construction of integer arrays $R_{\hat{X}:Y}$ and $R_{\hat{Y}:X}$ is rather more complicated. In the remainder of the analysis of this stage we outline how to generate $R_{\hat{X}:Y}$. Integer array $R_{\hat{Y}:X}$ is obtained analogously. To simplify the exposition, we assume that sequences $\hat{X}$ and $\hat{Y}$ are padded with

items $-\infty$ and $+\infty$. This assumption can be removed without affecting the asymptotic performance of our algorithm.

Let $\hat{x}$ be an item in $\hat{X}$, and let $\bar{y}_l(\hat{x})$ and $\bar{y}_r(\hat{x})$ be the two consecutive items in $\bar{Y}$, such that $\bar{y}_l(\hat{x}) \leq \hat{x} \leq \bar{y}_r(\hat{x})$. In order to determine $\text{rank}(\hat{x} : Y)$ it suffices to perform a binary search in sub-sequence $\langle y_{\text{rank}(\bar{y}_l(\hat{x}):Y)}, \dots, y_{\text{rank}(\bar{y}_r(\hat{x}):Y)} \rangle$. Recall that $\hat{X} \subset \bar{X} \subset X$, and similarly, $\hat{Y} \subset \bar{Y} \subset Y$.

A naive implementation of this step would require a CREW-BSP model to achieve optimality. Concurrency is eliminated by providing each processor with a private copy of the required sequence (on which it subsequently performs the search). To this end, for each item $\bar{y} \in \bar{Y}$ the two consecutive items $\hat{x}_l(\bar{y})$ and $\hat{x}_r(\bar{y})$ in $\hat{X}$ are computed, such that $\hat{x}_l(\bar{y}) \leq \bar{y} \leq \hat{x}_r(\bar{y})$. The computation of these two items, for each item $\bar{y}$ in $\bar{Y}$, is realized by two distinct parallel-prefix operations on sub-sequence $\bar{Z}$. The first parallel-prefix operation is from left to right of sequence $\bar{Z}$ and computes for each item $\bar{y}$ in $\bar{Y}$ item $\hat{x}_l(\bar{y})$. Analogously, the second parallel-prefix operation is from right to left of sequence $\bar{Z}$ and computes for each item $\bar{y}$ in $\bar{Y}$ item $\hat{x}_r(\bar{y})$. Thus, this step requires time $O(\omega_n) + 2 \cdot C_{ppf}(p)$ and $2 \cdot M_{ppf}(p)$, for computation and communication respectively.

For each pair $(\bar{y}_i, \bar{y}_j)$, $i < j$, of consecutive items in $\bar{Y}$, sub-sequence $\langle y_{\text{rank}(\bar{y}_i:Y)}, \dots, y_{\text{rank}(\bar{y}_j:Y)} \rangle$ is broadcast to the consecutive block of processors maintaining items $\hat{x}_r(\bar{y}_i)$ through $\hat{x}_l(\bar{y}_j)$. By way of stages 1–3, no processor broadcasts more than $\lceil n/p \rceil$ items, and no processor receives more than $\lceil n/(p \cdot \lceil \omega_n \rceil) \rceil$ items. Therefore, since different broadcast operations can be realized on disjoint sets of processors, this step takes time $T_{bcast}^{O(n/p)}(p)$.

Processors generate integer array $R_{\hat{X}:Y}$ by performing a binary search on the sub-sequence of size at most $\lceil n/(p \cdot \omega_n) \rceil$ they received in the previous step. This step takes time $O(\lg(n/(p \cdot \omega_n)))$. Integer array $R_{\hat{Y}:X}$ is computed similarly. We note that integer arrays $R_{\hat{X}:Y}$ and $R_{\hat{Y}:X}$ can be generated simultaneously. Therefore, this stage requires overall time $O(\omega_n + \lg(n/(p \cdot \omega_n))) + 2 \cdot C_{ppf}(p)$ and $2 \cdot M_{ppf}(p) + M_{bcast}^{O(n/p)}(p)$, for computation and communication respectively.

[Stage 5] Each processor p_i , for all $0 \leq i \leq |\hat{Z}|$, fetches into its local memory sub-sequences

$$X_i = \begin{cases} \langle x_1, \dots, x_{\text{rank}(\hat{z}_{i+1}:X)} \rangle & \text{if } i = 0, \\ \langle x_{\text{rank}(\hat{z}_i:X)+1}, \dots, x_{\text{rank}(\hat{z}_{i+1}:X)} \rangle & \text{if } 1 \leq i < |\hat{Z}|, \\ \langle x_{\text{rank}(\hat{z}_i:X)+1}, \dots, x_m \rangle & \text{if } i = |\hat{Z}|, \end{cases}$$

and

$$Y_i = \begin{cases} \langle y_1, \dots, y_{\text{rank}(\hat{z}_{i+1}:Y)} \rangle & \text{if } i = 0, \\ \langle y_{\text{rank}(\hat{z}_i:Y)+1}, \dots, y_{\text{rank}(\hat{z}_{i+1}:Y)} \rangle & \text{if } 1 \leq i < |\hat{Z}|, \\ \langle y_{\text{rank}(\hat{z}_i:Y)+1}, \dots, y_k \rangle & \text{if } i = |\hat{Z}|, \end{cases}$$

and subsequently merges them into sub-sequence Z_i by employing some optimal deterministic merging algorithm [54, 133]. The desired sequence is the concatenation of sequences Z_i , for all $0 \leq i \leq |\hat{Z}|$, in increasing order of i . Note that by way of stage 3, $|\hat{Z}| < p$.

[Analysis of Stage 5] Since integer arrays $R_{\hat{Z}:X}$ and $R_{\hat{Z}:Y}$ are available, the determination of $\text{rank}(\hat{z}_i : X)$ and $\text{rank}(\hat{z}_i : Y)$, for all $1 \leq i \leq |\hat{Z}|$, can be performed, by realizing a constant number

of local operations and a constant number of message exchanges, in time $O(1)$ and $L + g \cdot O(1)$, for computation and communication respectively. By way of stages 1–3, the size of $X_i \cup Y_i$, for all $0 \leq i \leq |\hat{Z}|$, is bounded above by $\lceil \omega_n \rceil \cdot \lceil n/(p \cdot \omega_n) \rceil = n/p + O(n/(p \cdot \omega_n))$. Thus, this stage takes time $c_* \cdot (1 + O(1/\omega_n)) \cdot (n/p)$ and $L + g \cdot O(n/p)$, for computation and communication respectively.

end

Analysis The correctness of the algorithm follows from its description. By summing up the computation and communication requirements of procedure MERGE we obtain the desired time bound.

The proposition follows. $\square$

Proof (of theorem 2.3.1) Let $\omega_n = \omega(1)$ and $\omega_n = o(n/(p \lg n))$. Then, the theorem follows easily from proposition 2.3.2 by substituting for T_{btn} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.3.1, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.4 Integer-Sorting

In this section we design, analyze and implement a BSP algorithm for a fundamental computational problem, namely *integer-sorting*. Parallel integer-sorting [133] is an important special case of sorting with a significant number of parallel applications, such as implementing parallel model simulations and improving the performance of computational geometry algorithms [9, 133, 206]. The material of this section is based on the following paper:

- (1) “Primitive operations on the BSP model.” Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996 [82].

2.4.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a sequence whose items belong to a finite universe $U = [0, 1, \dots, R - 1]$. We consider, in this section, the problem of *integer-sorting* sequence X into a non-decreasing sequence – say $\langle x_{\pi(1)}, \dots, x_{\pi(n)} \rangle = \text{non-decreasing}(X)$.

2.4.2 Cost Model

The performance of the BSP integer-sorting algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations.

Optimal $\Theta(n)$ time sequential integer-sorting algorithms are well-known [54, 133]. In the following sections, we design, analyze and implement simple optimal BSP integer-sorting algorithms – optimality is realized for an ample range of the BSP parameters.

2.4.3 Related Work and New Results

There has been a notable amount of previous work – mainly on the PRAM model – on parallel integer-sorting (e.g., [135, 174, 206] on the PRAM model, [20] on an abstract parallel communication model).

The exhibit in table 2.3 summarizes the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel (in terms of maximum number of processors) required to achieve optimality.

Refs	Computation	Communication	Constants	Implem.	Procs
[135]	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p)}\right)$	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p)}\right)$	■ ■	■	$O(n)$
[174]	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p)}\right)$	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■	$O(n)$
[206]	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p)}\right)$	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■	$O(n)$
[20]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O(\sqrt[3]{n})$
Sec 2.4.4	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p + \lg p)}\right)$	$O\left(\left(\frac{n}{p} + \lg p\right) \cdot \frac{\lg n}{\lg(n/p + \lg p)}\right)$	■	■	$O(n)$

Table 2.3: Performance comparison of parallel integer-sorting algorithms – range $[0, 1, \dots, n^{O(1)}]$.

With reference to table 2.3 we note that all algorithms derive from radix-sort techniques, and therefore, utilize similar concepts and ideas.

The optimal BSP integer-sorting algorithm of section 2.4.4 improves upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [135, 174, 206] (if $p = n / \lg^{o(1)} n$),
- (2) the asymptotic communication time requirements of [135, 174, 206] (if $p = n / \lg^{o(1)} n$),
- (3) the multiplicative constant factor requirements of [135], and
- (4) the parallel slackness requirements (to achieve optimality) of [20, 135, 174, 206].

The main contribution, however, of the integer-sorting algorithm of section 2.4.4 is attributed to the *low amount of parallel slackness* required to achieve optimality (refer to case 4).

2.4.4 An Optimal Integer-Sorting Algorithm

In this section, we provide an optimal BSP solution to the problem of integer-sorting a sequence that is based on radix-sort techniques and improves an idea of [174]. The algorithm is realized by employing exclusively parallel-prefix, regular routing and local operations, and achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic integer-sorting algorithm is based on parallel-prefix techniques and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Additionally, the n input items are assumed to belong to the range $R = [0, 1, \dots, n/p - 1]$ (to accommodate the case $R = [0, 1, \dots, (n/p)^{O(1)} - 1]$ standard radix-sort techniques may be applied). Initially, each processor computes the number of input items in its possession that fall into each of the n intervals induced by the range R . Subsequently, the sum and prefix sum of the number of items per interval is computed, by means of a parallel-prefix scheme, and communicated to each processor. Finally, the items are routed to their final position (the aforementioned routing operation is realized by employing the sum and prefix sum of the number of items per interval).

Our main result is summarized in the following theorem.

Theorem 2.4.1 *There exists a p -processor BSP algorithm for sorting a multi-set consisting of n integers in the range $[0, 1, \dots, n^{O(1)}]$, that for all integers n and p such that $p = O(n)$, requires time $O((n/p + \lg p) \cdot (\lg n / \lg(n/p + \lg p)))$ and $L \cdot O(\lg p \cdot (\lg n / \lg(n/p + \lg p))) + g \cdot O((n/p + \lg p) \cdot (\lg n / \lg(n/p + \lg p)))$, for computation and communication respectively. The algorithm requires $O(n/p + \lg p)$ space.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 2.4.1 follows easily as a consequence of this result.

Proposition 2.4.1 *There exists a p -processor BSP algorithm for sorting a multi-set consisting of n integers in the range $[0, 1, \dots, R - 1]$, that for all integers n , p and t , such that $p = O(n)$ and $2 \leq t \leq p$, requires time*

$$C_{rdx_srt}(n, R, p) = r \cdot \left(O \left(\max \left\{ \left\lceil \frac{n}{p} \right\rceil, t \cdot \lceil \log_t p \rceil \right\} \right) + C_{ppf}(p) \right),$$

and

$$M_{rdx_srt}(n, R, p) = r \cdot \left(L \cdot O(1) + g \cdot O \left(2 \cdot \max \left\{ \left\lceil \frac{n}{p} \right\rceil, t \cdot \lceil \log_t p \rceil \right\} + \left\lceil \frac{n}{p} \right\rceil \right) + M_{ppf}(p) \right),$$

for computation and communication respectively, where $r = \lceil \lg R / \lg(\max \{ \lceil n/p \rceil, t \cdot \lceil \log_t p \rceil \}) \rceil$. The overall time complexity of the algorithm is denoted by $T_{rdx_srt}(n, R, p) = C_{rdx_srt}(n, R, p) + M_{rdx_srt}(n, R, p)$. The algorithm requires $O(\max \{ \lceil n/p \rceil, t \cdot \lceil \log_t p \rceil \})$ space.

Proof We assume that the input is evenly but otherwise arbitrarily distributed among the p processors. We next present a procedure for sorting n integer items in the range $[0, 1, \dots, \bar{R} - 1]$, where $\bar{R} = \max \{ \lceil n/p \rceil, t \cdot \lceil \log_t p \rceil \}$. We then apply radix-sort techniques [133] to establish a procedure for sorting n integer items in the range $[0, \dots, R - 1]$, within $r = \lceil \lg R / \lg \bar{R} \rceil$ iterations of the basic procedure.

Procedure RADIX_SORT

begin

[**Stage 1**] Every processor counts the number of items in its possession that fall into the $\bar{R}$ intervals induced by $[0, 1, \dots, \bar{R} - 1]$. Let $C_{k,i}$ denote the number of integer items with value k in $[0, 1, \dots, \bar{R} - 1]$ on processor p_i , $0 \leq i < p$.

[**Analysis of Stage 1**] When this step is executed, each processor holds at most $\lceil n/p \rceil$ items. Thus, this stage requires $O(\bar{R})$ time.

[**Stage 2**] Each processor p_j , $0 \leq j < p$, computes

$$S_{m,j} = 1 + \sum_{k,i} (C_{k,i} \mid (k < m) \text{ or } (k = m \text{ and } i < j)), \text{ for all } m \text{ in } [0, 1, \dots, \bar{R} - 1].$$

[**Analysis of Stage 2**] The position in the sorted array of the first integer item with value m that is assigned to processor p_j is given by $S_{m,j}$. Hence, if each processor p_j computes the value of $S_{m,j}$

(over all m), then it can store, in sorted order, all of its items in the appropriate output position. To realize this computation it is sufficient to order the $C_{k,i}$'s in the direction of increasing k , perform a parallel-prefix sum operation, and finally restore the original order of the $C_{k,i}$'s. Thus, this stage requires the realization of two $\bar{R}$ -relations and a parallel-prefix sum operation, for a total time of $O(\bar{R}) + C_{ppf}(p)$ and $L \cdot 2 + g \cdot O(\bar{R}) + M_{ppf}(p)$, for computation and communication respectively.

[**Stage 3**] Each processor knows where to route its integer items, and performs the routing.

[**Analysis of Stage 3**] This stage requires time $L + g \cdot O(\lceil n/p \rceil)$.

end

Analysis The correctness of the algorithm follows from its description and is omitted. By applying radix-sort techniques [133] we obtain that n items in the range $[0, 1, \dots, R - 1]$, can be sorted within $r = \lceil \lg R / \lg(\max\{\lceil n/p \rceil, t \cdot \lceil \log_t p \rceil\}) \rceil$ iterations of procedure RADIX_SORT. Thus, by summing up the computation and communication requirements of procedure RADIX_SORT we obtain the desired time bound.

The proposition follows. $\square$

Remark The algorithm as described requires $O(\max\{\lceil n/p \rceil, t \cdot \lceil \log_t p \rceil\})$ space per processor. We can trivially verify that, by replacing in the above result $\lceil n/p \rceil$ for $\max\{\lceil n/p \rceil, t \cdot \lceil \log_t p \rceil\}$, the space can be reduced to $O(n/p)$ per processor.

Proof (of theorem 2.4.1) The theorem follows easily from proposition 2.4.1 by substituting for T_{ppf} the corresponding expression in proposition 2.1.4, and letting $t = 2$ in this expression. $\square$

2.4.4.1 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a variant of the integer-sorting algorithm of section 2.4.4.

Implementation The prototype adopted for the *BSPlib* integer-sorting procedure is outlined in figure 2.16.

```
void BSP_int_sort (byte *src, byte **trg, int n, int sb) ;
    src    - input array.
    trg    - output array.
    n      - input size.
    sb     - item size.
```

Figure 2.16: Procedure for BSP integer-sorting.

The aim of function (procedure) `BSP_int_sort` is to provide a high-performance *BSPlib* integer-sorting algorithm implementation.

Experimental Results The theoretical performance of the algorithm is summarized in theorem 2.4.1. To determine the practical significance of the aforementioned result we benchmarked our implementation (BSP_int_sort) on two different machines with a varying number of processors and input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). Figures 2.17 and 2.19 present the execution time of procedure BSP_int_sort on the two machines. Similarly, figures 2.18 and 2.20 expose the efficiency of the corresponding timing results.

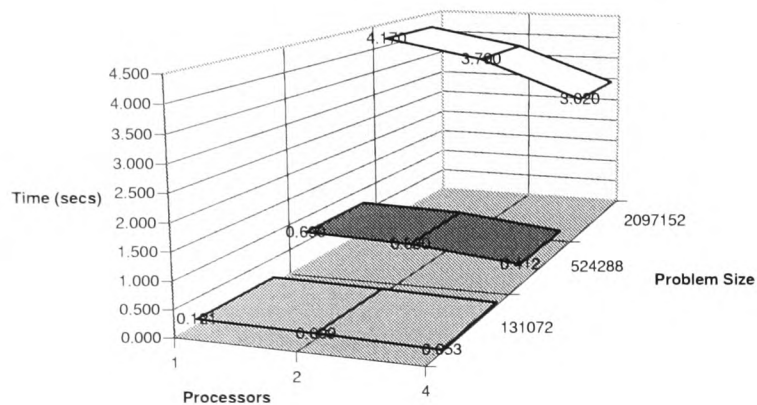


Figure 2.17: Execution time on a SGI Power Challenge for integer-sorting – uniform distribution (U).

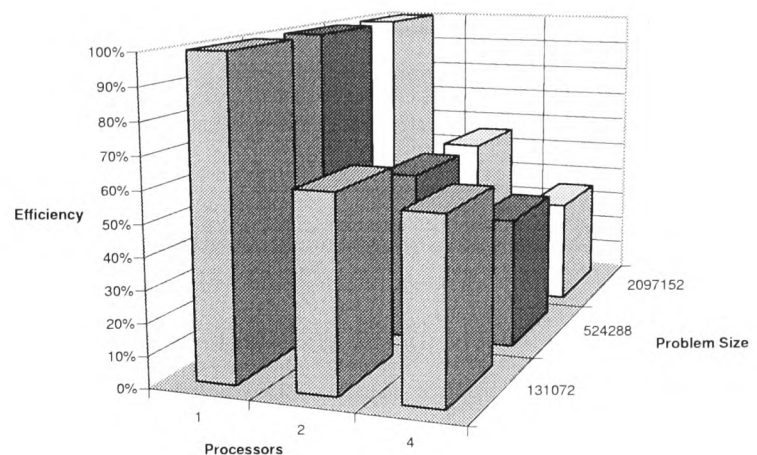


Figure 2.18: Efficiency on a SGI Power Challenge for integer-sorting – uniform distribution (U).

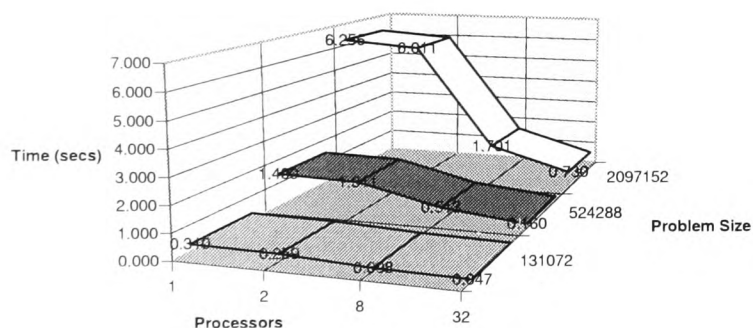


Figure 2.19: Execution time on a Cray T3D for integer-sorting – uniform distribution (U).

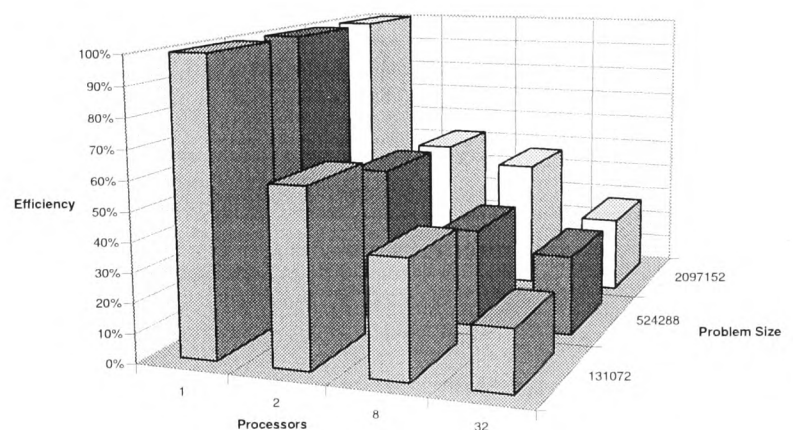


Figure 2.20: Efficiency on a Cray T3D for integer-sorting – uniform distribution (U).

As shown in figures 2.17 and 2.19, the execution time of procedure BSP_int_sort using a fixed number of processors scales with the input size (approximately as n). Accordingly, the execution time of procedure BSP_int_sort for a fixed input size scales inversely with the number of processors (approximately as $1/p$); nevertheless, the multiplicative constant factors introduced by the parallelization of the integer-sorting algorithm are quite large, and therefore, linear in p speedup is not achievable. Both experimental results are in accordance with the theoretical analysis of section 2.4.4. In terms of efficiency, procedure BSP_int_sort utilizes approximately (within a 5%):

- 45%–55% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 35%–55% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),

- 20%–55% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 25%–50% of the available processor resources on the Cray T3D machine (for large problem sizes).

These results should be interpreted with caution on a twofold basis: (i) the integer-sorting problem is communication bound, and therefore, efficiency close to one is not possible without the introduction of artificial conditions and premises, and (ii) the communication pattern of procedure `BSP_int_sort` is fine-grained (refer to stage 3 of procedure `RADIX_SORT` of section 2.4.4), and consequently, a considerable slowdown is incurred due to the inability of the current *BSPlib* version to cope with such a situation (particularly for large problem sizes and number processors).

In order to test the obliviousness of the integer-sorting algorithm to the input data distribution we benchmarked our implementation (`BSP_int_sort`) on a machine with a varying number of processors and input sets (consisting of 2^{21} integer valued items – unless otherwise indicated). Figure 2.21 presents the execution time of procedure `BSP_int_sort`. Similarly, figure 2.22 exposes the efficiency of the corresponding timing result.

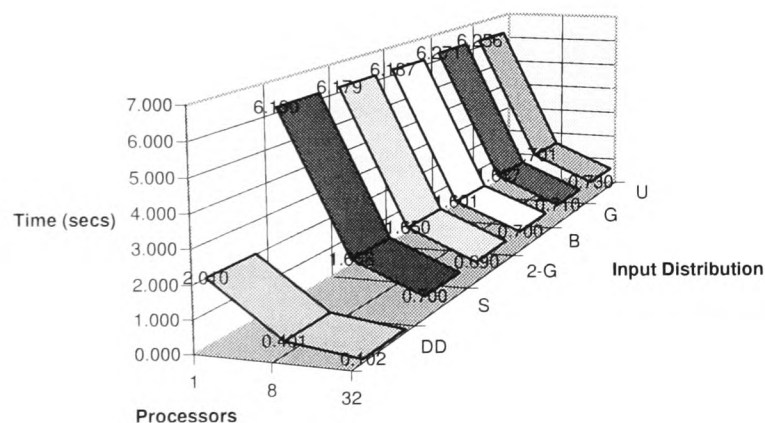


Figure 2.21: Execution time on a Cray T3D for integer-sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

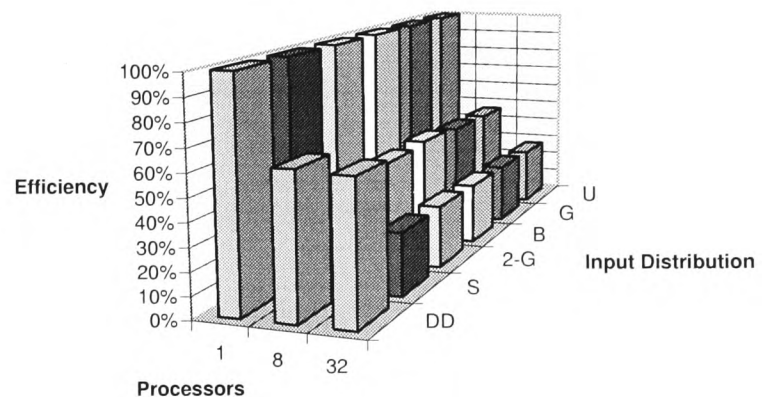


Figure 2.22: Efficiency on a Cray T3D for integer-sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

Obviously, the performance of the algorithm is quite consistent across the different input data distributions with the exception of the deterministic-duplicates distribution (we note that the observable inconsistency in procedure `BSP_int_sort`, with respect to the deterministic-duplicates distribution, derives from the underlying sequential integer-sorting-like process employed in the main body of `BSP_int_sort` and is irrelevant to the otherwise structure of the procedure).

2.5 Probabilistic Integer-Sorting

In this section we design, analyze and implement a BSP algorithm for a fundamental computational problem, namely *integer-sorting of uniformly at random distributed items*. Parallel integer-sorting of uniformly at random distributed items [133] is an important special case of integer-sorting with a number of

parallel applications, such as improving the performance of computational geometry problems [9, 133, 148]. The material of this section is based on the following papers:

- (1) “A note on probabilistic integer-sorting.” Technical Report PRG-TR-14-97, Computing Laboratory, Oxford University, March 1997 [86].
- (2) “On probabilistic integer sorting.” In *Proceedings of Workshop on Randomized Algorithms in Sequential, Parallel and Distributed Computing (RALCOM'97)*, October 1997 [89].

2.5.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a sequence of *uniformly at random distributed* items whose items belong to a finite universe $U = [0, 1, \dots, R - 1]$. We consider, in this section, the problem of *integer-sorting* sequence X into a non-decreasing sequence – say $\langle x_{\pi(1)}, \dots, x_{\pi(n)} \rangle = \text{non-decreasing}(X)$.

2.5.2 Cost Model

The performance of the BSP integer-sorting algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel prefix computations.

Optimal $\Theta(n)$ time sequential integer-sorting algorithms are well-known [54, 133]. In the following sections, we design, analyze and implement simple and rather more complicated optimal probabilistic BSP integer-sorting algorithms – optimality is realized for an ample range of the BSP parameters.

2.5.3 Related Work and New Results

There has been a notable amount of previous work – mainly on the PRAM model – on parallel probabilistic integer-sorting (e.g., [44, 104, 176] on the PRAM model). The exhibit in table 2.4 summarizes the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort, probability of success and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality. Approaches designed to exploit the CRCW capabilities of parallel models are identified with a † symbol.

Refs	Computation	Communication	Constants	Implem.	Probability	Procs
[44] †	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■ ■ ■	$1 - O\left(2^{-\rho \cdot n / \lg^3 n}\right)$	$O\left(\frac{n}{\lg n}\right)$
[104] †	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■ ■ ■	$1 - O\left(2^{-\rho \cdot \sqrt{n}}\right)$	$O\left(\frac{n}{\lg n}\right)$
[176] †	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■ ■	$1 - O\left(2^{-\rho \cdot \lg n}\right)$	$O\left(\frac{n}{\lg n}\right)$
Sec 2.5.5	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■ ■	■	$1 - O\left(2^{-\rho \cdot n}\right)$	$O\left(\frac{n}{\lg n}\right)$
Sec 2.5.6	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■	$1 - O\left(2^{-\rho \cdot \lg n}\right)$	$O\left(\frac{n}{\lg n}\right)$

Table 2.4: Performance comparison of parallel probabilistic integer-sorting algorithms (for any constant $\rho > 0$) – range $[0, 1, \dots, n^{O(1)}]$, – the cyan color is associated with randomized approaches.

With reference to table 2.4 we note that the algorithms of [44, 104, 176] are not stable as opposed to the algorithms of sections 2.5.5 and 2.5.6. We also note that the algorithms of [44, 104, 176] are designed to fully utilize the CRCW capabilities of parallel models, and therefore, compare unfairly with our algorithms

that are designed for parallel models with EREW capabilities. In fact, the performance of the algorithm of section 2.5.5 matches the performance of the best known deterministic EREW integer-sorting algorithm; yet, under realistic assumptions, the algorithm can be employed to optimally integer-sort uniformly at random items in an *arbitrary range* as opposed the best known deterministic EREW integer-sorting algorithm that can only be employed to optimally integer-sort items in the range $[0, 1, \dots, n^{O(1)}]$.

The optimal BSP integer-sorting algorithms of sections 2.5.5 and 2.5.6 improve upon previous approaches on the following particulars:

- (1) the implementation effort requirements of [44, 104, 176], and
- (2) the probability of success of [44, 104, 176] (algorithm of section 2.5.5).

The main contribution, however, of the probabilistic integer-sorting algorithms of section 2.5 is attributed to the *stability* and *simplicity* of the underlying BSP algorithm (refer to case 1), the *very high probability of success* (refer to case 2), and the *EREW* nature of the algorithm.

The optimal probabilistic BSP algorithm implied in the previous discussion can be trivially transformed into a highly efficient sequential algorithm. The sequential algorithm matches the performance of previous probabilistic integer-sorting approaches [44, 104, 148, 159, 176], yet, improves upon the probability of success of all these approaches. The exhibit in table 2.5 summarizes the aforementioned results with respect to the following statistics: computation time, multiplicative constant factors, implementation effort and probability of success.

Refs	Computation	Constants	Implem.	Probability
[148]	$O(n)$	■ ■	■	expected time
[159]	$O(n)$	■ ■	■	expected time
[44]	$O(n)$	■ ■	■	$1 - O\left(2^{-\rho \cdot n / \lg^3 n}\right)$
[104]	$O(n)$	■ ■	■	$1 - O\left(2^{-\rho \cdot \sqrt{n}}\right)$
[176]	$O(n)$	■ ■	■	$1 - O\left(2^{-\rho \cdot n / (\lg n \cdot \lg \lg n)}\right)$
Sec 2.5.5	$O(n)$	■ ■	■	$1 - O\left(2^{-\rho \cdot n}\right)$

Table 2.5: Performance comparison of sequential probabilistic integer-sorting algorithms (for any constant $\rho > 0$) – range $[0, 1, \dots, n^{O(1)}]$, – the cyan color is associated with randomized approaches.

2.5.4 A Combinatorial Lemma

This section is devoted to the proof of the following combinatorial lemma that is employed later in the analysis of the integer-sorting algorithm of section 2.5.5.

Lemma 2.5.1 *For any positive constants d and β , if k balls are independently placed into $k^{1+\beta}$ bins with probability at most $d/k^{1+\beta}$ of a ball being placed in a fixed bin, then there are at most $B \cdot (k/\lg k)$ balls into bins of size greater than or equal to D with probability at least $1 - 2^{-c \cdot k^{1+\beta}}$, for any positive constants B , D and c such that $0 < c < \beta \cdot (B - 2 \cdot B/D - B/(\beta \cdot D))$.*

Remark We assume that k is large enough to ensure that $(e \cdot D/B) \cdot (e \cdot d/D)^D \cdot \log k \leq k^\beta$ and $e \cdot d < k^\beta$.

Proof For any positive constant B , denote by $\mathcal{E}$ the event that there are at least $m = (B/D) \cdot (k/\lg k)$ bins of size greater than or equal to D . By definition, the probability of a ball being placed in a fixed bin is at most $d/k^{1+\beta}$. Then, by a variation of a simple combinatorial argument of [104], we bound $\Pr[\mathcal{E}]$ as follows.

$$\begin{aligned}
 \Pr[\mathcal{E}] &\leq \binom{k^{1+\beta}}{m} \cdot \sum_{a_1 \geq D, \dots, a_m \geq D} \binom{k}{a_1} \cdot \dots \cdot \binom{k - \sum_{j=1}^{m-1} a_j}{a_m} \cdot \left(\frac{d}{k^{1+\beta}}\right)^{a_1 + \dots + a_m} \\
 &\leq \binom{k^{1+\beta}}{m} \cdot \sum_{a_1 \geq D, \dots, a_m \geq D} \binom{k}{a_1} \cdot \dots \cdot \binom{k}{a_m} \cdot \left(\frac{d}{k^{1+\beta}}\right)^{a_1 + \dots + a_m} \\
 &\leq \binom{k^{1+\beta}}{m} \cdot \sum_{a_1 \geq D, \dots, a_m \geq D} \left(\frac{e \cdot k}{D} \cdot \frac{d}{k^{1+\beta}}\right)^{D \cdot m} \\
 &\leq \binom{k^{1+\beta}}{m} \cdot k^m \cdot \left(\frac{e \cdot k}{D} \cdot \frac{d}{k^{1+\beta}}\right)^{D \cdot m} \\
 &\leq \left(\frac{e \cdot D}{B} \cdot \left(\frac{e \cdot k}{D} \cdot \frac{d}{k^{1+\beta}}\right)^D \cdot k^{1+\beta} \cdot \lg k\right)^m \\
 &= \left(\frac{e \cdot D}{B} \cdot \left(\frac{e \cdot d}{D \cdot k^\beta}\right)^D \cdot k^{1+\beta} \cdot \lg k\right)^m.
 \end{aligned}$$

In the aforementioned bound we claimed the following inequality that derives from Stirling's formulae [40]:

$$\binom{x}{y} \leq \left(\frac{e \cdot x}{y}\right)^y.$$

By assumption, k is large enough to ensure that $(e \cdot D/B) \cdot (e \cdot d/D)^D \cdot \log k \leq k^\beta$. Therefore, for any $0 < c_1 < \beta \cdot (B - 2 \cdot B/D - B/(\beta \cdot D))$ the following bound is derived.

$$\begin{aligned}
 \Pr[\mathcal{E}] &\leq \left(\frac{1}{k^{\beta \cdot D}} \cdot k^{1+2 \cdot \beta}\right)^m \\
 &= \left(k^{1+2 \cdot \beta - \beta \cdot D}\right)^m \\
 &= 2^{(1+2 \cdot \beta - \beta \cdot D) \cdot ((B/D) \cdot k)} \\
 &\leq 2^{-c_1 \cdot k}.
 \end{aligned}$$

We have therefore established that the probability that there are at least m bins of size greater than or equal to D is at most $2^{-c_1 \cdot k}$ for some appropriately bounded positive constant c_1 . Denote by $\mathcal{Z}$ the event that the total number of balls falling into these m bins is at least $B \cdot (k/\lg k)$. Consider any fixed set of m bins out of the total $k^{1+\beta}$ bins. By definition, placing a ball into a bin is independent of all other placements. The total number of balls falling into these bins is stochastically dominated by a binomially distributed random variable, and therefore, by employing the probability bound of theorem A.1.2 on the tail of the binomial distribution the following upper bound on $\Pr[\mathcal{Z}]$ is derived.

$$\begin{aligned}
\Pr[Z] &\leq \binom{k^{1+\beta}}{m} \cdot \left(\frac{e \cdot d}{D \cdot k^\beta} \right)^m \\
&\leq \left(\frac{e \cdot D}{B} \cdot \left(\frac{e \cdot d}{D \cdot k^\beta} \right)^D \cdot k^\beta \cdot \lg k \right)^m.
\end{aligned}$$

By assumption, k is large enough to ensure that $(e \cdot D/B) \cdot (e \cdot d/D)^D \cdot \log k \leq k^\beta$. Therefore, the following bound holds for any $0 < c_2 < \beta \cdot (B - 2 \cdot B/D)$.

$$\begin{aligned}
\Pr[Z] &\leq (k^{2 \cdot \beta - \beta \cdot D})^m \\
&= 2^{(2 \cdot \beta - \beta \cdot D) \cdot ((B/D) \cdot k)} \\
&\leq 2^{-c_2 \cdot k}.
\end{aligned}$$

We have therefore established that the probability that at least $B \cdot (k/\lg k)$ balls fall into some fixed set of $(B/D) \cdot (k/\lg k)$ bins is at most $2^{-c_2 \cdot k}$, for some appropriately bounded positive constant c_2 . We have also shown that the probability that there are at least $(B/D) \cdot (k/\lg k)$ bins of size greater than or equal to D is at most $2^{-c_1 \cdot k}$. Therefore, for any $0 < c_1, c_2 < \beta \cdot (B - 2 \cdot B/D - B/(\beta \cdot D))$ and $c = c_1 = c_2$ there are at most $B \cdot (k/\lg k)$ balls into bins of size greater than or equal to D with probability at least $1 - 2^{-c \cdot k+1}$, and the lemma follows. $\square$

2.5.5 An Optimal Probabilistic Integer Sorting Algorithm

In this section, we provide an optimal probabilistic BSP solution to the problem of integer-sorting a sequence consisting of uniformly at random distributed items that is based on radix-sort techniques and extends the (sequential) expected-time radix-plus-sifting sorting algorithm of [148]. The algorithm derives from the ideas of [104, 148, 159, 176], and achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic integer-sorting algorithm is based on probabilistic techniques and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, each processor extracts an appropriate number of most significant bits (MSBs) from each item in its possession and sorts, by means of some optimal radix sort algorithm, the so extracted items. Subsequently, identical items are grouped (with respect to their MSBs) and items belonging to groups of constant pre-specified size are sorted, by means of specially tailored optimal sorters. The remaining items are sorted, by means of the optimal merge-sort algorithm of section 2.3.5. Finally, the so sorted items are merged with the previously sorted items belonging to groups of constant pre-specified size.

Our main result is summarized in the following theorem.

Theorem 2.5.1 *For any constant $\rho > 0$, there exists a p -processor BSP algorithm for sorting a uniformly at random distributed multi-set consisting of n integers in an arbitrary range, that for all integers n and p such that $p = O(n)$, requires time [with probability $1 - O(2^{-\rho \cdot n})$] $O((n/p + \lg p) \cdot (\lg n / \lg(n/p + \lg p)))$ and $L \cdot O(\lg p \cdot (\lg n / \lg(n/p + \lg p))) + g \cdot O((n/p + \lg p) \cdot (\lg n / \lg(n/p + \lg p)))$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.5.1 follows easily as a consequence of these results.⁷

In the remainder of this section we address a slightly more general problem; its solution provides us with a linear-time integer-sorting algorithm. The basic idea behind the algorithm in [148] is to sort the items with respect to some reduced number of Most Significant Bits (MSBs). As noted in [176], this nearly sorts the input. Suppose for example that the n integer items are sorted with respect to their $\lceil \lg n \rceil$ MSBs. Let S_i , for all $0 \leq i < 2^{\lceil \lg n \rceil}$, be the (possibly empty) sequence of items with value i for their $\lceil \lg n \rceil$ MSBs. Then, the original sorting problem reduces to $2^{\lceil \lg n \rceil}$ disjoint sub-problems, namely the problems of sorting $S_0, S_1, \dots, S_{2^{\lceil \lg n \rceil}-1}$. In particular, the required sorted sequence is $S_0^* \circ S_1^* \circ \dots \circ S_{2^{\lceil \lg n \rceil}-1}^*$, where S_i^* is the sequence of items in S_i in sorted order and $\circ$ denotes concatenation. Therefore, the total time required for the algorithm is $O(n) + \sum_i O(|S_i| \cdot \lg |S_i|)$, which is known to be $O(n)$ on the average [148]. The remainder of this section is devoted to a proof of the following proposition.

Proposition 2.5.1 *For some constant $\beta > 0$, let $f : U \rightarrow [0, 1, \dots, n^{1+\beta} - 1]$ be a non-decreasing function that can be evaluated in constant time, and suppose that $X_1, X_2, \dots, X_n$ are random variables such that for some positive constant d ,*

$$\Pr[f(X_i) = k] \leq \frac{d}{n^{1+\beta}}, \quad \text{for all } 1 \leq i \leq n \text{ and } k \text{ in } [0, 1, \dots, n^{1+\beta} - 1].$$

Then, for any constant $\rho > 0$, there exists a p -processor BSP algorithm for sorting the multi-set (of values) $\{X_1, X_2, \dots, X_n\}$, that for all integers n and p , such that $p = O(n)$, requires time [with probability $1 - O(2^{-\rho \cdot n})$]

$$C_{int_srt}(n, p) = O\left(\frac{n}{p}\right) + C_{rdx_srt}(n, n^{1+\beta}, p) + C_{srt}\left(O\left(\frac{n}{\lg n}\right), p\right) + C_{lbal}\left(O\left(\frac{n}{\lg n}\right), \left\lceil \frac{n}{p} \right\rceil, p\right)$$

and

$$M_{int_srt}(n, p) = L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + M_{rdx_srt}(n, n^{1+\beta}, p) + M_{srt}\left(O\left(\frac{n}{\lg n}\right), p\right) + M_{lbal}\left(O\left(\frac{n}{\lg n}\right), \left\lceil \frac{n}{p} \right\rceil, p\right),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{int_srt}(n, p) = C_{int_srt}(n, p) + M_{int_srt}(n, p)$.

Remark Suppose that $X_1, X_2, \dots, X_n$ are independent and uniformly distributed over $[0, 1, \dots, R - 1]$, where $R \geq n^{1+\beta}$. Then, as noted in [104], $f(\cdot)$ can be defined as follows.

$$f(x) = \begin{cases} 0 & \text{if } x < 0, \\ \left\lfloor (x - 1) \cdot \left\lceil \frac{R}{n^{1+\beta}} \right\rceil^{-1} \right\rfloor & \text{if } 0 \leq x < R, \\ R - 1 & \text{if } x \geq R. \end{cases}$$

⁷For the sequential case, the following analogue of theorem 2.5.1 applies.

Theorem 2.5.2 *For any constant $\rho > 0$, there exists a sequential algorithm for sorting a uniformly at random distributed multi-set consisting of n integers in an arbitrary range, that requires time [with probability $1 - O(2^{-\rho \cdot n})$] $O(n)$.*

Then, $\Pr[f(X_j) = k] \leq 2/n^{1+\beta}$ for all $1 \leq j \leq n$ and k in $[0, 1, \dots, n^{1+\beta} - 1]$. Function $f(\cdot)$ can be efficiently implemented by simply “extracting” from the (values of) X_i ’s an appropriate number of MSBs.

Proof (of proposition 2.5.1) Let $X = \{x_1, x_2, \dots, x_n\}$ be the input multi-set. We assume that X is evenly but otherwise arbitrarily distributed among the p processors.

Procedure R-S_SORT

begin

[**Stage 1**] For all $1 \leq j \leq n$, $y_j = f(x_j)$ is computed.

[**Analysis of Stage 1**] Since by definition, $f(\cdot)$ can be evaluated in constant time, this stage requires $O(n/p)$ time.

[**Stage 2**] Sequence $\langle y_1, y_2, \dots, y_n \rangle$ is sorted by way of some stable radix-sort algorithm.

[**Analysis of Stage 2**] Since by definition, y_j in $[0, 1, \dots, n^{1+\beta} - 1]$, for all $1 \leq j \leq n$, this stages requires time $T_{rdx_srt}(n, n^{1+\beta}, p)$.

Remark For all $0 \leq i < n^{1+\beta}$, let S_i be the (possibly empty) sequence $S_i = \langle x_j \mid 1 \leq j \leq n \text{ and } f(x_j) = i \rangle$, and let $S = S_0 \circ S_1 \circ \dots \circ S_{n^{1+\beta}-1} = \langle x_{w_1}, x_{w_2}, \dots, x_{w_n} \rangle$ be the concatenation of the S_i ’s. Then, the original sorting problem reduces to solving $l \leq n$ disjoint and non-empty sub-problems, namely, the problems of sorting $S_{u_1}, S_{u_2}, \dots, S_{u_l}$ such that S_{u_i} is non-empty and u_i in $[0, 1, \dots, n^{1+\beta} - 1]$, for all $1 \leq i \leq l$. In particular, the required sorted sequence is $S_{u_1}^* \circ S_{u_2}^* \circ \dots \circ S_{u_l}^*$, where $S_{u_i}^*$ is the sorted sequence of items in S_{u_i} .

[**Stage 3**] Let D be some positive constant to be specified later. Then, the $l_1 \leq l$ non-empty sub-sequences S_{v_i} satisfying $|S_{v_i}| < D$, with v_i in $[0, 1, \dots, n^{1+\beta} - 1]$, for all $1 \leq i \leq l_1$, are determined and sorted (independently) by way of some deterministic stable sorter. For each $x_j \in \cup_i S_{v(i)}$ the following information is also maintained: $\text{pos}(x_j : S)$ and $\text{pos}(x_j : S_{v_i})$.

[**Analysis of Stage 3**] If D is sufficiently small, then the algorithm employs specially tailored sorters to sort the S_{v_i} ’s; otherwise, the algorithm employs any deterministic $O(n \cdot \lg n)$ time sorter – such sorters can be found in [133]. Hence, since the number of non-empty sequences S_{v_i} is $l_1 \leq l \leq n$ this stage requires time $O((n/p) \cdot \lg D)$ and $L + g \cdot O(D)$, for computation and communication respectively.

[**Stage 4**] Each item $x_j \in \cup_i S_{v_i}$, for all $1 \leq j \leq n$, is placed in the output sequence at position $\text{pos}(x_j : S) + (\text{rank}(x_j : S_{v_i}) - \text{pos}(x_j : S_{v_i}))$.

[**Analysis of Stage 4**] Since sequence S_{v_i} is in sorted order, the rank of x_j in S_{v_i} can be computed in constant time, and therefore, this stage takes time $O(n/p)$ and $L + g \cdot O(n/p)$, for computation and communication respectively.

[**Stage 5**] The $l_2 = l - l_1 \leq l$ non-empty sub-sequences S_{w_i} satisfying $|S_{w_i}| \geq D$, with w_i in $[0, 1, \dots, n^{1+\beta} - 1]$, for all $1 \leq i \leq l_2$, are determined, load-balanced and sorted (globally) by way of some deterministic stable sorter. For each $x_j \in \cup_i S_{w(i)}$ the following information is also maintained: $\text{pos}(x_j : S)$ and $\text{pos}(x_j : \cup_i S_{w_i})$.

[**Analysis of Stage 5**] This stage requires time $O(n/p) + C_{lbal}(|\cup_i S_{w_i}|, \lceil n/p \rceil, p) + C_{srt}(|\cup_i S_{w_i}|, p)$ and $M_{lbal}(|\cup_i S_{w_i}|, \lceil n/p \rceil, p) + M_{srt}(|\cup_i S_{w_i}|, p)$, for computation and communication respectively.

[**Stage 6**] Each item $x_j \in \cup_i S_{w_i}$, for all $1 \leq j \leq n$, is placed in the output sequence at position $\text{pos}(x_j : S) + (\text{rank}(x_j : \cup_i S_{w_i}) - \text{pos}(x_j : \cup_i S_{w_i}))$.

[**Analysis of Stage 6**] Since sequence $\cup_i S_{w_i}$ is in sorted order, the rank of x_j in $\cup_i S_{w_i}$ can be computed in constant time, and therefore, this stage takes time $O(n/p)$ and $L + g \cdot O(n/p)$, for computation and communication respectively.

end

Analysis The correctness of the algorithm follows from its description and is omitted. To establish the time requirements of procedure R-S_SORT it remains to derive an upper bound for $|\cup_i S_{w_i}|$. Substituting however in lemma 2.5.1 for $k = n$ the following is derived for any positive constants B , D and ρ that satisfy the conditions of lemma 2.5.1.

$$\Pr \left[|\cup_i S_{w_i}| \geq B \cdot \frac{n}{\lg n} \right] \leq 1 - O(2^{-\rho \cdot n}).$$

In particular, for $\beta = 3/2$ and $D = 4$ we get the following bound.

$$\Pr \left[|\cup_i S_{w_i}| \geq 2 \cdot \frac{n}{\lg n} \right] \leq 1 - O(2^{-n}).$$

Thus, by summing up the computation and communication requirements of procedure R-S_SORT we obtain the desired time bound.

The proposition follows. $\square$

Proof (of theorem 2.5.1) The theorem follows from proposition 2.5.1 by substituting for T_{rdx_srt} , T_{srt} and T_{lbal} the corresponding expressions in propositions 2.4.1, 2.6.8 and 2.1.5, and letting $t = 2$ in this expressions. $\square$

2.5.6 Yet Another Optimal Probabilistic Integer-Sorting Algorithm

In this section we present a variant of the BSP algorithm of section 2.5.5. The algorithm is based on the sequential version of algorithm R-S_SORT and achieves optimality for an even wider range of the parameters n and p (at the expense of lower probability of success).

Informal Description of Algorithm The optimal deterministic integer-sorting algorithm is based on probabilistic techniques and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, each processor extracts $\lg p$ most significant bits (MSBs) from each item in its possession and routes, with respect to the so extracted MSBs, the items in its possession to the appropriate processor. Finally, the items received by each processor in the previous step are sorted, by means of the sequential version of the optimal integer-sorting algorithm of section 2.5.5.

Our main result is summarized in the following theorem.

Theorem 2.5.3 *For any constant $\rho > 0$, there exists a p -processor BSP algorithm for sorting a uniformly at random distributed multi-set consisting of n integers in an arbitrary range, that for all integers n and p such that $p = O(n)$, requires time [with probability $1 - O(2^{-\rho \cdot (n/p) + \lg p})$] $O(n/p)$ and $L + g \cdot O(n/p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 2.5.3 follows easily as a consequence of this result.

Proposition 2.5.2 *For some constant $\beta > 0$, let $f : U \rightarrow [0, 1, \dots, n^{1+\beta} - 1]$ be a non-decreasing function that can be evaluated in constant time, and suppose that $X_1, X_2, \dots, X_n$ are random variables such that for some positive constant d ,*

$$\Pr[f(X_i) = k] \leq \frac{d}{n^{1+\beta}}, \text{ for all } 1 \leq i \leq n \text{ and } k \text{ in } [0, 1, \dots, n^{1+\beta} - 1].$$

Then, for any constant $\rho > 0$, there exists a p -processor BSP algorithm for sorting the multi-set (of values) $\{X_1, X_2, \dots, X_n\}$, that for all integers n and p , such that $p = O(n)$, requires time [with probability $1 - O(2^{-\rho \cdot (n/p) + \lg p})$]

$$C_{int_srt}(n, p) = O\left(\frac{n}{p}\right)$$

and

$$M_{int_srt}(n, p) = L + g \cdot O\left(\frac{n}{p}\right),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{int_srt}(n, p) = C_{int_srt}(n, p) + M_{int_srt}(n, p)$.

Proof The main inefficiency of algorithm R-S_SORT of section 2.5.5 derives from the inability to optimally radix-sort the input items for $p = n^{1-o(1)}$. We overcome this problem by orderly and evenly distributing the items among the processors – without employing radix sort – and then by applying the sequential version of algorithm R-S_SORT to the so distributed items.

In the remainder of the analysis we sketch the algorithmic details of the modified R-S_SORT algorithm. Stage 1 is identical to that of the original R-S_SORT algorithm, however, the remaining stages are replaced by the following two stages (stages 2 and 3). In stage 2, the input items are mapped into $[0, 1, \dots, 2^{\lfloor \lg p \rfloor} - 1]$, with respect to the $\lfloor \lg p \rfloor$ MSBs of the values of stage 1, and subsequently, using this information the input items are routed to the corresponding processors.

It is not difficult to verify that the probability of a fixed processor receiving a given input item is at most $(2 \cdot d)/p$. Therefore, by employing the probability bound of theorem A.1.2 on the tail of the binomial distribution the following bound on the number of items n_i received by processor p_i is derived. For any constant $\rho > 0$,

$$\Pr\left[\exists i \left(n_i = O\left(\frac{n}{p}\right)\right)\right] = 1 - O\left(2^{-\rho \cdot (n/p) + \lg p}\right), \quad 0 \leq i < p.$$

Hence, this stage requires time $O(n/p)$ and $L+g \cdot O(n/p)$, for computation and communication respectively. In stage 3, each processor sorts the items in its possession by employing the sequential version of algorithm R-S_SORT. Since the number of items per processor is $O(n/p)$, this stage requires time $O(n/p)$, with probability $1 - O(2^{-\rho \cdot (n/p) + \lg p})$, for any constant $\rho > 0$. The proposition follows. $\square$

Proof (of theorem 2.5.3) The theorem follows easily from proposition 2.5.2. $\square$

2.5.6.1 Implementation and Experimental Results

In this section we discuss an implementation – in C – of a variant of the sequential probabilistic integer-sorting algorithm of section 2.5.5.⁸

Implementation The prototype adopted for the sequential probabilistic integer-sorting procedure is outlined in figure 2.23.

```
void prob_int_sort (byte *src, byte **trg, int n, int sb) ;
    src    - input array.
    trg    - output array.
    n      - input size.
    sb     - item size.
```

Figure 2.23: Procedure for sequential probabilistic integer-sorting.

The aim of function (procedure) `prob_int_sort` is to provide a high-performance probabilistic integer-sorting algorithm implementation.

Experimental Results The theoretical performance of the algorithm is summarized in theorem 2.5.2. To determine the practical significance of the aforementioned result we benchmarked our implementation (`prob_int_sort`) on a machine with a varying number of input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). Figures 2.24, 2.25 and 2.26 present the execution time of procedure `prob_int_sort` (radix-plus-sifting sort (R-S) of section 2.5.5)⁹ along with the execution times of a multiplicity of comparing algorithms – radix sort (X) [133], quick sort (Q) [115], and hybrid sort (HB) [104, 159].

As shown in figures 2.24, 2.25 and 2.26, the execution time of procedure `prob_int_sort` scales with the input size (approximately as n). Additionally, the execution time of procedure `prob_int_sort` scales sub-linearly with the input range. Both experimental results are in accordance with the theoretical analysis of section 2.5.5. In all cases of interest (input size $> 2^6$), our algorithm outperforms the comparing algorithms by a wide margin. Figure 2.27 establishes the aforementioned claim for the largest of the benchmarked problem sizes.

⁸We decided to study the sequential version (as opposed to the BSP version) of the probabilistic integer-sorting algorithm on a twofold basis: (i) the performance of the BSP integer-sorting algorithm can be extrapolated from the performance of the sequential integer-sorting algorithm (by independently considering the performance of the BSP implementations of the constituent parts of the algorithm), and (ii) the sequential integer-sorting algorithm can be easily compared to previous approaches on integer-sorting [44, 104, 148, 159, 176] (recall that previous approaches on parallel models [44, 104, 176] exploit the CRCW capabilities of the underlying model, and therefore, are difficult to implement and study on realistic parallel models).

⁹In the implementation of procedure `prob_int_sort` we fixed – in the proof of theorem 2.5.1 – D to 5 and ranged β from 1 to 3.

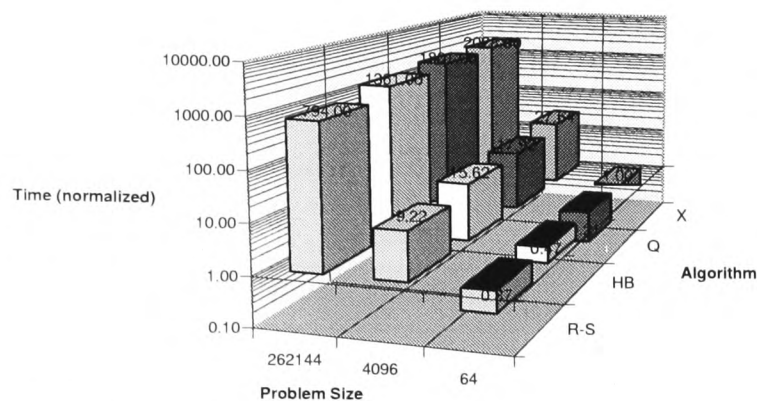


Figure 2.24: Normalized execution time on a SGI Power Challenge for radix sort (X), quick sort (Q), hybrid sort (HB) and radix-plus-sifting sort (R-S) – uniform distribution (U) in $[0, 1, \dots, 2^{80} - 1]$.

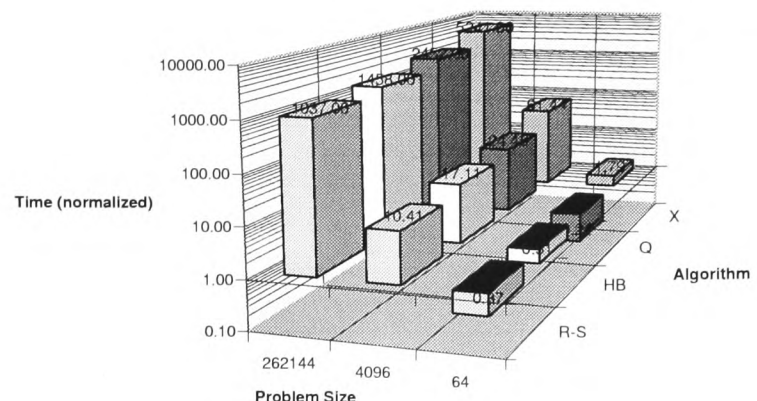


Figure 2.25: Normalized execution time on a SGI Power Challenge for radix sort (X), quick sort (Q), hybrid sort (HB) and radix-plus-sifting sort (R-S) – uniform distribution (U) in $[0, 1, \dots, 2^{160} - 1]$.

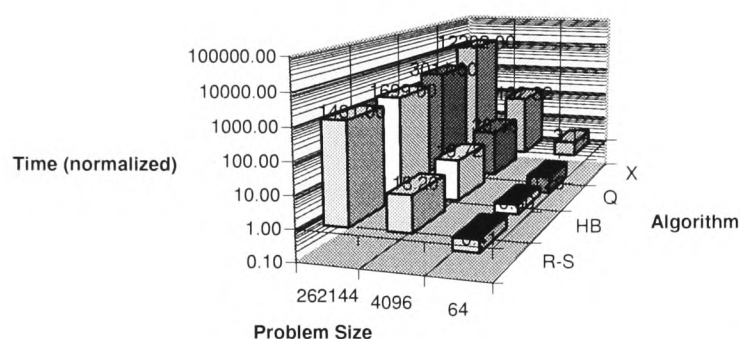


Figure 2.26: Normalized execution time on a SGI Power Challenge for radix sort (X), quick sort (Q), hybrid sort (HB) and radix-plus-sifting sort (R-S) – uniform distribution (U) in $[0, 1, \dots, 2^{320} - 1]$.

Obviously, as indicated in figure 2.27, the execution time of our algorithm is approximately 1.2–1.7 times faster than the fastest of the comparing algorithms.

2.6 Sorting

In this section we design, analyze and implement BSP algorithms for a fundamental combinatorial operation, namely *sorting*. The sorting operation [133] has always been a fruitful research topic both in sequential [31, 54, 115, 133, 192] and parallel computation [4, 5, 7, 22, 47, 101, 133, 144, 145, 183, 196] because of its many important applications and its intrinsic theoretical significance. Parallel sorting [7, 133] is a fundamental operation on ordered sequences and of great importance for many parallel applications, such as implementing database primitives and computational geometry algorithms [7, 133, 198]. The material of this section is based on the following papers:

- (1) “Portable and scaleable algorithm design on the IBM SP2: The bulk-synchronous parallel paradigm.” In *Proceedings of the Sup-Eur’96 Conference*, Krakow, Poland, September 1996 [11].
- (2) “Deterministic sorting and randomized median finding on the BSP model.” In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996

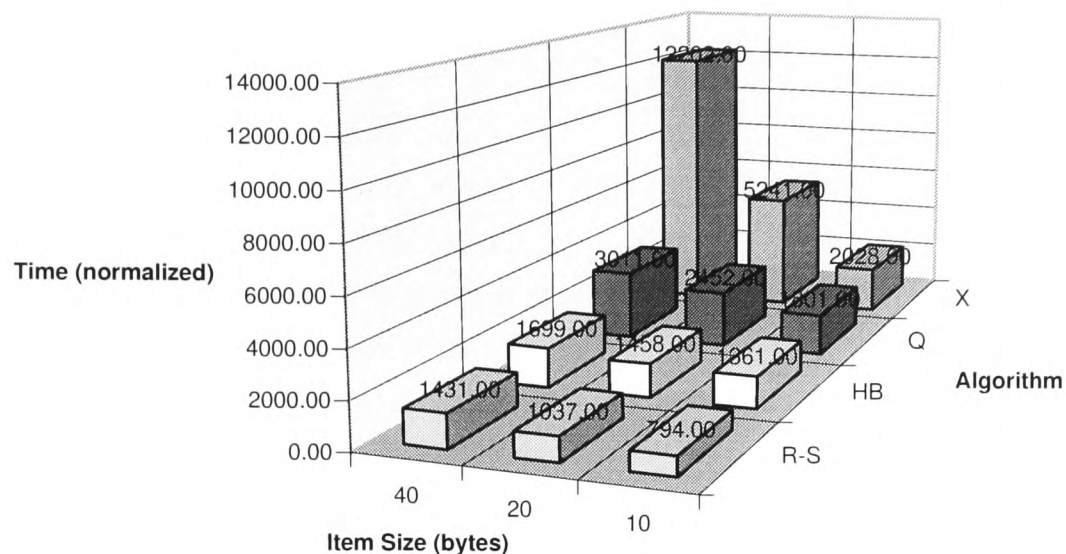


Figure 2.27: Normalized execution time on a SGI Power Challenge for radix sort (X), quick sort (Q), hybrid sort (HB) and radix-plus-sifting sort (R-S) – uniform distribution (U), input size 2^{18} .

[77].

- (3) “Efficient deterministic sorting on the BSP model.” Technical Report PRG-TR-19-96, Oxford University Computing Laboratory, October 1996 [81]. Also to appear in *Parallel Processing Letters*, 1997-8 [92].
- (4) “An experimental study of sorting algorithms on the BSP model: Theory, practice and experience.” Technical Report PRG-TR-06-97, Computing Laboratory, Oxford University, February 1997 [84].
- (5) “A randomized sorting algorithm on the BSP model.” In *Proceedings of the International Parallel Processing Symposium (IPPS’97)*, April 1997 [87].
- (6) “An experimental study of BSP sorting algorithms.” Submitted for publication, 1997 [95].
- (7) “On the complexity of BSP sorting.” Technical Report PRG-TR-09-96, Computing Laboratory, Oxford University, May 1996 [196].

2.6.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a sequence whose items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments. Without loss of generality, we assume that all the items appearing in sequence X are distinct – if the items are not distinct we can make them so by appending to them their index. We consider, in this section, the problem of *sorting* sequence X into a non-decreasing sequence – say $\langle x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)} \rangle = \text{non-decreasing}(X)$.

Let x , y and z be three positive integers, with $x < z$. The following notation and definitions are assumed in the remainder of this section. Integer y is *between* x and z (or equivalently, x and z *straddle* y) if and only if $x < y$ and $y \leq z$. Let $X = \langle x_1, x_2, \dots, x_n \rangle$ and Y be non-decreasing sequences, and let x_i be an item in X , $1 \leq i \leq n$. The *position* of item x_i in sequence X , denoted by $\text{pos}(x_i : X)$, is i . The *rank* of item x_i in sequence Y , denoted by $\text{rank}(x_i : Y)$, is the number of items of Y that are less than or equal to x_i . *Ranking* X in Y , amounts to computing the integer array $\text{rank}(X : Y) = (r_1, r_2, \dots, r_n)$,

where $r_i = \text{rank}(x_i : Y)$, $1 \leq i \leq n$. The *predecessor* of an arbitrary item x_i in X is item x_{i-1} of X (if necessary, we let $x_{i-1} = -\infty$), and the range $(x_{i-1}, x_i]$ is the interval *induced* by item x_i (including the cases $x_{i-1} = -\infty$ and $x_i = +\infty$).

Let c be a positive integer. Sequence X is a c -cover of Y , denoted by $\text{cvr}(Y)$, if each interval induced by an item in X contains at most c items from $\{-\infty, Y, +\infty\}$. More formally, given any two consecutive items x_{i-1} and x_i of $\{-\infty, X, +\infty\}$, the multi-set $\{y \in Y \mid x_{i-1} < y \leq x_i\}$ has at most c items. The items from Y in the range $(x_{i-1}, x_i]$ are *contained* in x_i 's interval.

Let $[x_i, x_j]$, $1 \leq i < j \leq n$ be an interval with $x_i, x_j \in \{-\infty, X, +\infty\}$. The interval $[x_i, x_j]$ *intersects* $\{-\infty, X, +\infty\}$ in c items (or equivalently that there are c items *in common*) if the number of items $z \in \{-\infty, X, +\infty\}$ such that $x_i \leq z \leq x_j$ is equal to c .

2.6.2 Cost Model

The performance of the BSP sorting algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations. Furthermore, for some constants c_{srt} , c_{mrg} and c_{srch} , time $c_{srt} \cdot n \cdot \lg n$, $c_{mrg} \cdot n$, and $c_{srch} \cdot \lg n$ is charged respectively for sorting a sequence of size n , for merging two non-decreasing sequences of combined size n , and for binary searching (for a single item) in a non-decreasing sequence of size n – within the context of this section we denote c_{srt} by c_* .

Optimal $\Theta(n \cdot \lg n)$ time sequential sorting algorithms are well-known [54, 133]. In the following sections, we design, analyze and implement simple sub-optimal and rather more complicated optimal BSP sorting algorithms – optimality is realized for an ample range of the BSP parameters.

2.6.3 Related Work and New Results

There has been a notable amount of previous work – mainly on the PRAM model – on parallel sorting (e.g., [8, 34, 47, 60, 72, 105, 187, 194] on the PRAM model, [108, 109] on an abstract parallel communication model, [75, 101] on the BSP model, [5, 22, 144] on the comparison network model, [183] on the cube-connected-cycles model). The exhibit in table 2.6 summarizes the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality.

With reference to table 2.6 we note that the deterministic comparison network algorithm of [5] employs expander graphs, and therefore, its practical viability is rather limited. Furthermore, the randomized PRAM algorithm of [187] employs a non-standard (stronger) PRAM model, and therefore, compares unfairly with other algorithms. Finally, the randomized cube-connected-cycles algorithm of [183] does not migrate efficiently to the BSP model as a result of its inherent requirement for a non-bulk communication pattern (cube-connected-cycles style communication) – as opposed to a bulk communication pattern (BSP style communication).

The optimal BSP sorting algorithms of sections 2.6.6, 2.6.7 and 2.6.8 improve upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [22, 144],
- (2) the asymptotic communication time requirements of [5, 8, 22, 34, 47, 60, 72, 105, 144, 183, 187],

Refs	Computation	Communication	Implem.	Constant	Procs
[22]	$O\left(\frac{n}{p} \cdot \left(\lg \frac{n}{p} + \lg^2 p\right)\right)$	$O\left(\frac{n}{p} \cdot \lg^2 p\right)$	■	■	$O(n)$
[5]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■ ■ ■	■ ■ ■ ■	$O(n)$
[144]	$O\left(\frac{n}{p} \cdot \frac{\lg^{3.42} n}{\lg^{2.42}(n/p)}\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg^{3.42} n}{\lg^{3.42}(n/p)}\right)$	■ ■	■ ■	$O(n)$
[187]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■ ■	$O(n)$
[8]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■	■ ■	$O\left(\frac{n}{\lg^2 n}\right)$
[183]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■ ■	$O(n)$
[72]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■	■ ■	$O\left(\sqrt{\frac{n}{\lg n}}\right)$
[47]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■ ■	■ ■ ■ ■	$O(n)$
[34]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[105]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[194]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■ ■	$O(\sqrt[3]{n})$
[60]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg^2 n}\right)$
[75]	$c_* \cdot \frac{n}{p} \cdot \lg n + o\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■ ■	$O\left(\frac{n}{\lg^{1+\alpha} n}\right)$
[108]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■ ■	$O\left(\sqrt{\frac{n}{\lg n}}\right)$
[109]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■ ■	$O(\sqrt[3]{n})$
[101] ^a	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■ ■ ■	■ ■ ■ ■	$O(n)$
Sec 2.6.6	$c_* \cdot \left(\frac{\alpha+2}{\alpha}\right) \cdot \frac{n}{p} \cdot \lg n + o\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■ ■	$O\left(\frac{n}{\lg^{2+\alpha} n}\right)$
Sec 2.6.7	$c_* \cdot \frac{n}{p} \cdot \lg n + o\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■	■ ■	$O\left(\frac{n \cdot \lg \lg n}{\lg n}\right)$
Sec 2.6.8	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■ ■ ■	■ ■ ■ ■	$O(n)$

^aThe algorithm implied in the work of [101] is based on an algorithmic scheme similar to that of section 2.6.8, and has been developed independently to this work.

Table 2.6: Performance comparison of parallel sorting algorithms (for any $\alpha = \Omega(1)$) – the cyan color is associated with randomized approaches.

- (3) the multiplicative constant factor requirements of [5, 8, 22, 34, 47, 60, 72, 101, 105, 108, 109, 144, 183, 187, 194] (algorithms of sections 2.6.6 and 2.6.7) and [5] (algorithm of section 2.6.8),
- (4) the implementation effort requirements of [5, 47, 101, 187, 183] (algorithms of sections 2.6.6 and 2.6.7), and
- (5) the parallel slackness requirements (to achieve optimality) of [22, 72, 108, 109, 144, 194] (algorithm of section 2.6.6), [8, 22, 34, 60, 72, 75, 105, 108, 109, 144, 194] (algorithm of section 2.6.7) and [8, 22, 34, 60, 72, 75, 105, 108, 109, 144, 194] (algorithm of section 2.6.8).

The principal contribution, however, of the sorting algorithms of sections 2.6.6 and 2.6.7 is attributed to the *low communication cost* (refer to case 2), the *small constant multiplicative factors* (refer to case 3), and the *low amount of parallel slackness* required to achieve optimality (refer to case 5). Additionally, our techniques provide *tighter probabilistic bounds* than previous randomized methods [75, 108, 183, 187] and *one-optimality* for an ample range of the BSP parameters. Accordingly, the main contribution of the sorting algorithm of section 2.6.8 is attributed to the *low communication cost* (refer to case 2) and the

low amount of parallel slackness required to achieve optimality (refer to case 5).

2.6.4 A Sub-Optimal Sorting Algorithm

In this section, we provide a BSP solution to the problem of sorting a sequence X that is based on the merge-sort technique [54, 133] and derives from algorithm MERGE of section 2.3.5. The algorithm achieves optimality in computation for an ample range of the parameters n and p . Unfortunately, the amount of communication required is sub-optimal – in section 2.6.8 we refine the merge-sort technique of this section to obtain a communication optimal BSP sorting algorithm.

Informal Description of Algorithm The optimal deterministic sorting algorithm is based on merge-sort techniques and is iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, each processor sorts, by means of some optimal sequential sorting algorithm, the items in its possession. In each iterative call, pairs of sorted sub-sequences are merged, by means of the optimal merging algorithm of section 2.3.5, in a tree-like fashion. Finally, after a pre-specified number of iterations (equal to $\lceil \lg p \rceil$) the iterative process results in a single sorted sequence.

The main result of this section is summarized in the following theorem.

Theorem 2.6.1 *There exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p such that $p = o(n/\lg n)$, requires time $c_* \cdot ((n/p) \cdot (\lg(n/p))) + (c_{\text{mrg}} + o(1)) \cdot ((n/p) \cdot \lg p)$ and $L \cdot O(\lg^2 p) + g \cdot O((n/p) \cdot \lg p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 2.6.1 follows easily as a consequence of this result.

Optimality Claim 2.6.1 *Theorem 2.6.1 implies that there exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n , p , L and g such that $p = o(n/\lg n)$ and $L = o(n/(p \cdot \lg p))$, has $\pi = \max\{1, c_{\text{mrg}}/c_*\} + o(1)$ and $\mu = O(g)$.*

Proposition 2.6.1 *There exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p , such that $p = O(n/\lg n)$, requires time at most*

$$C_{\text{mrg_srt}}(n, p) = c_* \cdot \left\lceil \frac{n}{p} \right\rceil \cdot \lg \left\lceil \frac{n}{p} \right\rceil + \sum_{i=1}^{\lceil \lg p \rceil} \left(C_{\text{mrg}} \left(2^i \left\lceil \frac{n}{p} \right\rceil, 2^i \right) \right)$$

and

$$M_{\text{mrg_srt}}(n, p) = \sum_{i=1}^{\lceil \lg p \rceil} \left(M_{\text{mrg}} \left(2^i \left\lceil \frac{n}{p} \right\rceil, 2^i \right) \right),$$

for communication and computation respectively. The overall time complexity of the algorithm is denoted by $T_{\text{mrg_srt}}(n, p) = C_{\text{mrg_srt}}(n, p) + M_{\text{mrg_srt}}(n, p)$.

Proof The sorting algorithm is a BSP adaptation of sequential merge-sort [54, 133]. Initially, each processor, in parallel with all other processors, sorts the items in its possession by some optimal deterministic sorting algorithm [54, 133]. When this step is executed, each processor holds at most $\lceil n/p \rceil$ items, and therefore, this step takes time $c_* \cdot \lceil n/p \rceil \cdot \lg \lceil n/p \rceil$. Subsequently, processors merge the resulting p sequences via $\lceil \lg p \rceil$ merges of pairs of sequences of total size n , and the desired result follows. $\square$

Proof (of theorem 2.6.1) The theorem follows easily from proposition 2.6.1 by substituting for T_{mrg} the corresponding expression in proposition 2.3.2 (as adapted in theorem 2.3.1). $\square$

2.6.5 A Sorting Algorithm with n^2 Processors

In this section, we provide a BSP solution to the problem of sorting a sequence X of size n with $p = n^2$ processors. The algorithm is an adaptation of an algorithm due to [167] and is realized by exclusively employing broadcasting, parallel-prefix, regular routing and local operations.

Informal Description of Algorithm The deterministic sorting algorithm is based on broadcast and parallel-prefix techniques and is non-iterative in nature. The n^2 participating processors are assumed to be logically arranged in a mesh-like fashion and the input is assumed to be evenly but otherwise arbitrarily divided among the n processors of the first row before the beginning of the execution of the algorithm. Initially, each first-row processor broadcasts its item to all processors of its column and, in the next step, each first-column processor broadcasts its item to all processors of its row. Subsequently, processors compare the two items in their possession and compute the rank of each item, by means of a parallel-prefix scheme. Finally, the items are routed to their final position (the aforementioned routing operation is realized by employing the rank of each item).

The main result of this section is summarized in the following theorem.

Theorem 2.6.2 *There exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p such that $p = \Omega(n^2)$, requires time $O(\lg p)$ and $L \cdot O(\lg p) + g \cdot O(\lg p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 2.6.2 follows easily as a consequence of this result.

Proposition 2.6.2 *There exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p requires time*

$$C_{quad_srt}(n, p) = O\left(\left\lceil \frac{n^2}{p} \right\rceil\right) + C_{ppf}^{\lceil n^2/p \rceil}(\sqrt{p})$$

and

$$M_{quad_srt}(n, p) = L \cdot O(1) + g \cdot O\left(\left\lceil \frac{n^2}{p} \right\rceil\right) + 2 \cdot M_{bcast}^{\lceil n^2/p \rceil}(\sqrt{p}) + M_{ppf}^{\lceil n^2/p \rceil}(\sqrt{p}),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{quad_srt}(n, p) = C_{quad_srt}(n, p) + M_{quad_srt}(n, p)$.

Proof Consider input sequence $X = \langle x_1, x_2, \dots, x_n \rangle$ of size n and n^2 processors. We number processors in a mesh-like fashion, so that we can refer to processor at row i and column j , as $p_{i,j}$, $1 \leq i, j \leq n$. We also assume that the n input items are evenly distributed among the n processors of the first row – item x_j resides in the local memory of processor $p_{1,j}$, for all $1 \leq j \leq n$. By realizing n disjoint broadcast operations, processors $p_{1,j}$, for all $1 \leq j \leq n$, communicate item x_j to all processors of column j . Similarly, processors $p_{i,i}$, for all $1 \leq i \leq n$, communicate item x_i to all processors of row i . In a single superstep, processors $p_{i,j}$, for all $1 \leq i, j \leq n$, compare x_i and x_j , and subsequently, by realizing n disjoint parallel-prefix sum operations determine $\text{rank}(x_i : X)$, for all $1 \leq i \leq n$. The sorting operation is completed by routing each item to its final destination. The overall time complexity of the algorithm can be easily seen to be $O(1) + C_{ppf}(\sqrt{p})$ and $L + g \cdot O(1) + 2 \cdot M_{broadcast}(\sqrt{p}) + C_{ppf}(\sqrt{p})$. In case $p < n^2$, then each processor is assigned a block of the square grid of size $\lceil n^2/p \rceil$, and the proposition follows. $\square$

Proof (of theorem 2.6.2) The theorem follows easily from proposition 2.6.2 by substituting for $T_{broadcast}$ and T_{ppf} the corresponding expression in propositions 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.6.6 An Optimal Deterministic Sorting Algorithm

In this section, we provide an optimal BSP solution to the problem of sorting a sequence X that is based on deterministic *over-sampling* techniques [75, 183] and employs a partitioning scheme that meets the requirements of efficiency and insensitivity against initial data distribution.¹⁰ The algorithm achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic sorting algorithm is based on over-sampling techniques and is recursive in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, each processor sorts, by means of some optimal sequential sorting algorithm, the items in its possession. In each iteration of the algorithm, each processor constructs (in a deterministic way) a sub-sequence that is an appropriate over-sample of its input. Subsequently, the so constructed sub-sequences are merged, by means of the sub-optimal bitonic-sort algorithm of section 2.3.4, and a sub-sequence of size $k-1$ (where k is appropriately chosen) is constructed that is an appropriate sample of the resulting merged sequence. This sub-sequence is broadcast to all processors and is employed to partition and re-distribute the input into k non-overlapping partitions. Accordingly, processors are logically re-arranged into k non-intersecting groups. Subsequently, each processor sorts, by means of an optimal composite sequential merge / sort scheme, the items in its possession. Finally, after a pre-specified number of iterations the iterative process results in a single sorted sequence per processor.

Our main result is summarized in the following theorem.

¹⁰A distinction between the approach that we consider in this work and previous BSP approaches [75] is that our method does not sample the unsorted input data but the sorted initial runs. The accuracy of the splitting items depends solely on user defined parameters, which can be adapted to meet the worst-case requirements of parallel applications. Therefore, although we employ sampling [75, 117, 194] to realize efficiency, we give a precise worst-case estimation of the maximum imbalance which might occur.

The idea of regular (deterministic) sampling is not a new one, however, our techniques improve upon previous results on (i) the amount of parallel slackness required to achieve optimality, (ii) the communication requirements, and (iii) the multiplicative constant factors involved.

Theorem 2.6.3 *For arbitrarily small constant $\zeta > 0$ and any $\alpha > \zeta$, there exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p such that $p = O(n/\lg^{3+\alpha} n)$, requires time $(c_* + 3 \cdot (c_*/\alpha) + o(1)) \cdot ((n/p) \cdot \lg n)$ and $L \cdot O(\lg^2 p \cdot (\lg n / \lg(n/p))) + g \cdot O((n/p) \cdot (\lg n / \lg(n/p)))$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.6.3 follows easily as a consequence of these results. Additionally, an improved version of theorem 2.6.3 is shown in sub-section 2.6.6.1.

Optimality Claim 2.6.2 *Theorem 2.6.3 implies that there exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n , p , L and g such that $p = O(n/\lg^{3+\alpha} n)$, $L = o((n/p) \cdot (\lg(n/p)/\lg^2 p))$ and $g = o(\lg(n/p))$, has $\pi = 1 + 3 \cdot (c_*/\alpha) + o(1)$ and $\mu = o(1)$.*

The deterministic sorting algorithm is based on an efficient partitioning scheme that splits – almost evenly and independently of the input distribution – an arbitrary number of sorted sequences among the processors. The algorithm consists of the following phases:

- (1) *Local Sorting.* Each processor, in parallel with all the other processors, sorts the sub-sequence that resides in its local memory.
- (2) *Partitioning.* Processors cooperate to evenly split the sorted sequences among the processors.
- (3) *Merging.* Each processor, in parallel with all the other processors, merges a small number of sub-sequences of total size $(1 + o(1)) \cdot (n/p)$.

The iterative version of the algorithm performs m iterations of phases 2 and 3. In each iteration the data set is partitioned into k buckets of approximately equal size. In the next iteration a similar process of sub-partitioning each of these k buckets into k further buckets is performed, and so on. The maximum size of every bucket formed in any of the m iterations is shown to be $(1 + O(1/\omega_n)) \cdot (n/p)$, for any slowly growing function ω_n . By unfolding the iteration process and employing multi-way merging [133] we obtain the desired sorted sequence. The procedure to manage the aforementioned scheme is presented in the following result.

Proposition 2.6.3 *Let ω_n be any function of n such that $\omega_n = \Omega(1)$. Then, there exists a p -processor BSP algorithm for sorting n items, that for any integers n , p , m and r , and any $\bar{m}$ such that $p = O(n)$, $1 \leq \bar{m} \leq \lfloor \lg p \rfloor$, $1 \leq m \leq \lceil \bar{m} \rceil$, $0 \leq r \leq m$, and $\omega_n^2 \cdot m^2 \cdot p^{1/\bar{m}} = O(n/p)$, requires time,*

$$C_{srt}(n, p) = c_* \cdot \frac{n}{p} \cdot \lg \frac{n}{p} + \left(1 + O\left(\frac{1}{\omega_n}\right)\right) \cdot \frac{n}{p} \cdot \left(c_{mrg} \cdot \left(\frac{r \cdot (\bar{m} - m)}{\bar{m}} + \frac{r \cdot (r + 1)}{2 \cdot \bar{m}}\right) \cdot \lg p + c_* \cdot (m - r) \cdot \lg \frac{n}{p}\right) + \sum_{j=0}^{m-1} \left(\lceil \lg q_{(j)} \rceil \cdot C_{btn}(\lceil \omega_n \rceil \cdot m \cdot \lceil p^{1/\bar{m}} \rceil \cdot q_{(j)}, q_{(j)}) + C_{ppf}^{O(p^{1/\bar{m}})}(q_{(j)})\right) + \sum_{j=0}^{m-1} \left(O\left(\frac{n_{(j)}}{q_{(j)}}\right)\right)$$

and

$$M_{srt}(n, p) = L \cdot O(m) + g \cdot \sum_{j=0}^{m-1} O\left(\frac{n_{(j)}}{q_{(j)}}\right) + \sum_{j=0}^{m-1} \left(\lceil \lg q_{(j)} \rceil \cdot C_{btn}(\lceil \omega_n \rceil \cdot m \cdot \lceil p^{1/\bar{m}} \rceil \cdot q_{(j)}, q_{(j)}) + C_{bcast}^{O(p^{1/\bar{m}})}(q_{(j)}) + C_{ppf}^{O(p^{1/\bar{m}})}(q_{(j)})\right),$$

for computation and communication respectively, where $n_{(j)} \leq n$, $q_{(j)} \leq p$, $q_{(j)} \leq n_{(j)}$, $n_{(j)}/q_{(j)} = (1 + O(1/\omega_n)) \cdot (n/p)$. The overall time complexity of the algorithm is denoted by $T_{srt}(n, p) = C_{srt}(n, p) + M_{srt}(n, p)$.

Proof The algorithm is iterative in nature and each iteration is divided into a sequence of stages. We assume that the input set is initially evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm. Moreover, we assume that the items are distinct since we can always make them so by, for example, appending to them the code for their memory location. We later explain how we can remove such an assumption and still maintain optimality.

Algorithm DET_SORT

begin

SEQ_SORT

RECURSIVE_SORT (0, $p - 1$, n)

end

Procedure SEQ_SORT

begin

[Stage 1] Each processor sorts, by way of a deterministic sequential sorting algorithm, the items in its possession.

[Analysis of Stage 1] When this step is executed, each processor holds at most $\lceil n/p \rceil$ items. Thus, the cost of this stage is that of sequentially sorting $\lceil n/p \rceil$ items by some deterministic algorithm [133] and requires time $c_* \cdot \lceil n/p \rceil \cdot \lg \lceil n/p \rceil$.

end

In procedure RECURSIVE_SORT below the variables l , u and N have initially values 0, $p - 1$ and n respectively; in any recursive call of the algorithm, the former two variables give the range of the processors that participate in the call and the latter variable gives the input size.

Procedure RECURSIVE_SORT (l , u , N)

begin

[Stage 1] Each processor selects locally $\lceil \omega_n \rceil \cdot m \cdot k - 1$ evenly spaced sample items, that partition its input into $\lceil \omega_n \rceil \cdot m \cdot k$ evenly sized segments, where $k = \min \{q, \lceil p^{1/\bar{m}} \rceil\}$, $q = u - l + 1$ and m , $\bar{m}$ satisfy the conditions in the statement of the proposition. Additionally, each processor appends to this sorted list the largest item in its input. Let $s = \lceil \omega_n \rceil \cdot m \cdot k$ be the size of the so identified list.

[Analysis of Stage 1] Since the items residing in the local memory of any processor are sorted prior to the sample selection process, this stage requires time $O(s)$.

[Stage 2] The q sorted lists, each of size s , are merged; let sequence $\langle s_{ls+1}, s_{ls+2}, \dots, s_{ls+qs} \rangle$ be the result of the merge operation. By assumption, the sequence is evenly distributed among the q processors – sub-sequence $\langle s_{is+1}, \dots, s_{(i+1)s} \rangle$, $l \leq i \leq u$, resides in the local memory of processor p_i , $0 \leq i < p$.

[**Analysis of Stage 2**] The merging operation is realized by way of a BSP adaptation of bitonic-sort [22]. Thus, by proposition 2.3.1, this stage requires time $\lceil \lg q \rceil \cdot T_{btm}(s \cdot q, q)$.

[**Stage 3**] A broadcast operation is initiated, where splitter $s_{ls+iqs/k}$, $1 \leq i < k$, along with its index in the sequence of sample items is communicated to all q processors.

[**Analysis of Stage 3**] The so identified $k - 1$ splitter items are first sent to processor p_l and then broadcast to all q processors. This stage takes communication time $L + g \cdot O(k) + M_{bcst}^{O(k)}(q)$ for computation and communication respectively.

[**Stage 4**] (i) Each processor decides the position of every item it holds with respect to the $k - 1$ splitters it received in the previous stage, by way of merging, and counts the number of items that fall into each of the k so identified buckets; (ii) k independent parallel prefix operations are initiated (one for each bucket) to determine how to split the items of each bucket as evenly as possible among the processors using the information collected in step (i) of this stage.

[**Analysis of Stage 4**] Steps (i) and (ii) take $O(N/q + k)$ and $T_{ppf}^k(q)$ time respectively.

[**Stage 5**] Each processor knows where to route its items, and performs the routing. Processors route items in “column-major” order, that is, items destined to some bucket fill the processors assigned to this bucket in linear order, the first processor being determined by the information gathered in step (ii) of stage 4.

[**Analysis of Stage 5**] This stage takes time $L + g \cdot O(N/q)$.

[**Stage 6**] Each processor merges the at most q sorted sub-sequences that it received in the previous stage.

[**Analysis of Stage 6**] When this step is executed, each processor possesses at most $\min\{q, \lceil N/q \rceil\}$ sorted sequences for a total of at most $\lceil N/q \rceil$ items. The cost of this stage is that of sequential multi-way merging/sorting $\lceil N/q \rceil$ items by some deterministic algorithm [133], which is $\lceil N/q \rceil \cdot \min\{c_{mrg} \cdot \lg q, c_* \cdot \lg \lceil N/q \rceil\}$.

[**Stage 7**] If $l \geq u$, then exit; otherwise, recursively sort each of the k buckets generated in stages 1–6 by assigning to each bucket $\lceil q/k \rceil$ (or $\lfloor q/k \rfloor$) consecutive processors.

[**Analysis of Stage 7**] The depth of the recursion is $m \leq \lceil \bar{m} \rceil$. To simplify the exposition, in the remainder of this section we assume that q/k is integer valued. This assumption can be removed without affecting the asymptotic analysis of our algorithm.

end

Analysis The correctness of the algorithm follows from its description and is omitted. We proceed by establishing upper bounds on the computation and communication requirements of procedure DET_SORT. We start by showing the following bound on the maximum number of items $n_{(j)}$ per bucket in the j -th recursive call of the algorithm (in the subsequent analysis the j -th recursive call is denoted by the subscript notation $_{(j)}$). We assume $n_{(0)} = n$.

Claim 2.6.1 *Let ω_n be any function of n such that $\omega_n = \Omega(1)$. Let m be the depth of the recursion of procedure DET_RECURSIVE_SORT, and consider integers n , p , m and any $\bar{m}$ such that $\omega_n^2 \cdot m^2 \cdot p^{1/\bar{m}} = O(n/p)$. Then,*

$$n_{(j)} \leq \left((1 + \varepsilon)^j \cdot \frac{n_0}{q_0} + O\left(\omega_n \cdot m \cdot p^{1/\bar{m}} \cdot j\right) \right) \cdot q_{(j)}, \quad \text{for all } 1 \leq j \leq m,$$

where $\varepsilon = 1/(m \cdot \lceil \omega_n \rceil)$.

Proof Although it is not explicitly mentioned in the description of procedure RECURSIVE_SORT we assume that we initially pad the input so that each processor owns exactly $\lceil n/p \rceil$ items. At most one item is added to each processor (the maximum item can be such a choice). Before performing stage 1 we also pad the input so that after stage 1 all segments have the same number of items that is, $x = \lceil \lceil n/p \rceil / s \rceil$. The padding operation requires time at most $O(s)$ per iteration, which is within the lower order terms of the analysis of proposition 2.6.3, and therefore, does not affect the asymptotic complexity of the algorithm. We note that padding operations introduce duplicate items. We have shown how to handle duplicate items in the beginning of the section. A more refined approach is to be shown in a later section.

Refer to the j -th recursive call of stage 3 of procedure RECURSIVE_SORT and consider an arbitrary splitter $s_{ls_{(j)} + (iq_{(j)} s_{(j)})/k_{(j)}}$, where $1 \leq i < k_{(j)}$. There are at least $((i \cdot q_{(j)} \cdot s_{(j)})/k_{(j)}) \cdot x$ items which are not larger than $s_{ls_{(j)} + (iq_{(j)} s_{(j)})/k_{(j)}}$, since there are $(i \cdot q_{(j)} \cdot s_{(j)})/k_{(j)}$ segments each of size x whose items are not larger than $s_{ls_{(j)} + (iq_{(j)} s_{(j)})/k_{(j)}}$. Likewise, there are at least $(q_{(j)} \cdot s_{(j)} - (i \cdot q_{(j)} \cdot s_{(j)})/k_{(j)} - q_{(j)} + 1) \cdot x$ items which are not smaller than $s_{ls_{(j)} + (iq_{(j)} s_{(j)})/k_{(j)}}$, since there are $q_{(j)} \cdot s_{(j)} - (i \cdot q_{(j)} \cdot s_{(j)})/k_{(j)} - q_{(j)} + 1$ segments each of size x whose items are not smaller than $s_{ls_{(j)} + (iq_{(j)} s_{(j)})/k_{(j)}}$. Thus, by noting that the total number of items has been increased (by way of padding operations) from $n_{(j)}$ to $q_{(j)} \cdot s_{(j)} \cdot x$, the number of items $b_{(j)}^i$ that are smaller than $s_{ls_{(j)} + (iq_{(j)} s_{(j)})/k_{(j)}}$ is bounded as follows.

$$\frac{i \cdot q_{(j)} \cdot s_{(j)}}{k_{(j)}} \cdot x \leq b_{(j)}^i \leq q_{(j)} \cdot s_{(j)} \cdot x - \left(q_{(j)} \cdot s_{(j)} - \frac{i \cdot q_{(j)} \cdot s_{(j)}}{k_{(j)}} - q_{(j)} + 1 \right) \cdot x.$$

Similarly we obtain the following bound on $b_{(j)}^{i+1}$.

$$\frac{(i+1) \cdot q_{(j)} \cdot s_{(j)}}{k_{(j)}} \cdot x \leq b_{(j)}^{i+1} \leq q_{(j)} \cdot s_{(j)} \cdot x - \left(q_{(j)} \cdot s_{(j)} - \frac{(i+1) \cdot q_{(j)} \cdot s_{(j)}}{k_{(j)}} - q_{(j)} + 1 \right) \cdot x.$$

Substituting $s_{(j)} = \lceil \omega_n \rceil \cdot m \cdot k_{(j)}$ we therefore conclude the following.

$$b_{(j)}^{i+1} - b_{(j)}^i \leq \frac{q_{(j)} \cdot s_{(j)}}{k_{(j)}} \cdot x + q_{(j)} \cdot x = \lceil \omega_n \rceil \cdot m \cdot q_{(j)} \cdot x + q_{(j)} \cdot x.$$

As evidenced from the above expression, the difference $n_{(j)} = b_{(j)}^{i+1} - b_{(j)}^i$ is independent of i and gives the maximum number of items per bucket in the j -th iteration. Considering that $x \leq (n_{(j)} + q_{(j)} + q_{(j)} \cdot s_{(j)})/(q_{(j)} \cdot s_{(j)})$ and substituting $s_{(j)} = \lceil \omega_n \rceil \cdot m \cdot k_{(j)}$, the following bound is derived.

$$n_{(j)} \leq (1 + \varepsilon) \cdot \frac{n_{(j-1)} + q_{(j)} \cdot (s_{(j)} + 1)}{k_{(j)}}.$$

Solving the recurrence relation and substituting for $n_{(0)} = n$, $k_{(j)} * q_{(j)} = q_{(0)} = p$ and $s_{(j)} \leq \lceil \omega_n \rceil \cdot m \cdot \lceil p^{1/\bar{m}} \rceil$ we obtain the desired bound. $\square$

We proceed by deriving an upper bound on the number of items per processor at the j -th recursive call. Let parameters $n_{(j)}$ and $q_{(j)}$ denote, respectively, the number of items per processor and the number of processors in the j -th recursive call of procedure DET_RECURSIVE_SORT.

Initially, $n_{(0)} = n$ and $q_{(0)} = p$. Then, by way of claim 2.6.1 and the fact that $\omega_n^2 \cdot m^2 \cdot p^{1/\bar{m}} = O(n/p)$, for $\varepsilon = 1/(m \cdot \lceil \omega_n \rceil)$, we have that

$$\frac{n_{(j)}}{q_{(j)}} \leq (1 + O(\varepsilon))^j \cdot \frac{n}{p}. \quad (2.1)$$

We next establish the time complexity of stage 6 of algorithm DET_RECURSIVE_SORT. In particular, it is straightforward to verify that the time complexity of this stage over the m iterations of the procedure is bounded above as follows.

$$\left\lceil \frac{n_{(j)}}{q_{(j)}} \right\rceil \cdot \sum_{j=1}^m \min \left\{ c_{mrg} \cdot \lg q_{(j-1)}, c_* \cdot \lg \left\lceil \frac{n_{(j)}}{q_{(j)}} \right\rceil \right\} \leq \left\lceil \frac{n_{(j)}}{q_{(j)}} \right\rceil \cdot \left(c_{mrg} \cdot \sum_{j=0}^{r-1} \lg q_{(m-j)} + c_* \cdot \sum_{j=r}^{m-1} \lg \left\lceil \frac{n_{(j)}}{q_{(j)}} \right\rceil \right).$$

The value of r that minimizes the above expression is determined by comparing $q_{(j)}$ and $\lceil n_{(j)}/q_{(j)} \rceil$. Substituting $(1 + O(1/\omega_n)) \cdot (n/p)$ for $n_{(j)}/q_{(j)}$ we obtain the following bound on the cost of this stage.

$$\left(1 + O\left(\frac{1}{\omega_n}\right) \right) \cdot \frac{n}{p} \cdot \left(c_{mrg} \cdot \sum_{j=0}^{r-1} \lg q_{(m-j)} + c_* \cdot \sum_{j=r}^{m-1} \lg \left(\left(1 + O\left(\frac{1}{\omega_n}\right) \right) \cdot \frac{n}{p} \right) \right).$$

The following relations are noted: $q_{(0)} = p$, $\lfloor q_{(j-1)}/k_{(j-1)} \rfloor \leq q_{(j)} \leq \lceil q_{(j-1)}/k_{(j-1)} \rceil$, $k_{(j)} = \lceil p^{1/\bar{m}} \rceil$ if $j \neq m$ and $k_{(m)} = q_{(m)}$. To simplify the exposition, in the remainder of the analysis we assume that $q_{(j-1)}/k_{(j-1)}$ is integer valued; this assumption can be removed without affecting the asymptotic analysis of the algorithm. Hence, we have that $q_{(j)} = p/\lceil p^{1/\bar{m}} \rceil^{j-1}$ if $j \neq m$. We conclude that this stage requires time bounded as follows.

$$\left(1 + O\left(\frac{1}{\omega_n}\right) \right) \cdot \frac{n}{p} \cdot \left(c_{mrg} \cdot \left(\frac{r \cdot (\bar{m} - m)}{\bar{m}} + \frac{r \cdot (r + 1)}{2\bar{m}} \right) \cdot \lg p + c_* \cdot (m - r) \cdot \lg \frac{n}{p} + O(m) \right).$$

In the aforementioned bound we claimed inequality $\lg(x + y) \leq \lg x + 1.5 \cdot (y/x)$ which holds by way of $\log(1 + x) \leq x$, for any $x, y > 0$ [40].

By summing up the computation and communication requirements of procedures SEQ_SORT and DET_RECURSIVE_SORT we obtain the desired time bound. The bound on $n_{(j)}/q_{(j)}$ follows from equation 2.1.

The proposition follows. $\square$

The following theorem is derived from proposition 2.6.3 and establishes concrete bounds on the computation and communication requirements of algorithm DET_SORT.

Corollary 2.6.1 *There exists a p -processor BSP algorithm for sorting n items, that for any integers p , m and r , and any $\bar{m}$ and t such that $p = n^{1-t}$, $1 \leq \bar{m} < \lg p$, $1 \leq m \leq \lceil \bar{m} \rceil$, $0 \leq r \leq m$, $m = O(\lg n / \omega_n)$, $\omega_n^2 \cdot m^2 \cdot p^{1/\bar{m}} = O(n/p)$ and $\omega_n^2 \cdot m^2 \cdot p^{1/\bar{m}} \cdot \lg^2 p = O((n/p) \cdot \lg n)$ and $0 < t < 1$, and any function ω_n of n such that $\omega_n = \Omega(1)$, has,*

$$C_{srt}(n, p) = \left((1 + (m - r)) \cdot t + \frac{c_{mrg}}{c_*} \cdot \left(\frac{r \cdot (\bar{m} - m)}{\bar{m}} + \frac{r \cdot (r + 1)}{2\bar{m}} \right) \cdot (1 - t) + O\left(\frac{1}{\omega_n}\right) \right) \cdot \frac{n \cdot \lg n}{p}$$

and

$$M_{srt}(n, p) = \left(g \cdot O\left(\frac{\omega_n \cdot m^2 \cdot \lg n}{n^{t+(1/\bar{m})(t-1)}} \cdot (1 - t)^2 + \frac{m}{\lg n}\right) + O\left(\frac{1}{\omega_n}\right) \right) \cdot \frac{n \cdot \lg n}{p},$$

provided that $L = O((n/p) \cdot (\lg n / (\omega_n \cdot m \cdot \lg^2 p)))$ is satisfied.

Proof We first consider the computation time of the sorting algorithm. Let in proposition 2.6.3, $p = n^{1-t}$, $m = O(\lg n / \omega_n)$, $\omega_n^2 \cdot m^2 \cdot p^{1/\bar{m}} = O(n/p)$, $\omega_n^2 \cdot m^2 \cdot p^{1/\bar{m}} \cdot \lg^2 p = O((n/p) \cdot \lg n)$ and $n_{(j)}/q_{(j)} = (1 + O(1/\omega_n)) \cdot (n/p)$. Then, the bound on the computation time derives by substituting for T_{btn} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.3.1, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions.

The expression for the communication requirements is established similarly. $\square$

Proof (of theorem 2.6.3) Let $\omega_n = \omega(1)$, $\omega_n = O((\alpha \cdot \lg \lg n) / \lg(1 + \alpha))$ and $r = 1$. Then, the theorem follows from corollary 2.6.1 by substituting for $\bar{m} = (1 - t)/(t - \delta)$ and $\delta = (3 \cdot \lg \lg n + 2 \cdot \lg(\omega_n / (\alpha \cdot \lg \lg n))) / \lg n$, where $t = (3 + \alpha) \cdot (\lg \lg n / \lg n)$. $\square$

2.6.6.1 Improving Deterministic Sorting

We note that not only can the bounds given above be expressed in a variety of ways, but also the algorithm itself leaves room for several improvements. As stated, the algorithm sorts the splitters using the bitonic-sort sorting algorithm. In certain circumstances the performance can be improved by any of several alternative methods. For example, the sorting of the splitters can be performed by either using our algorithm recursively or by some completely different algorithm. In particular, since the set of splitters per processor is already sorted we may use a merging algorithm for sorting the splitters; that is, q sorted sequences, each of size s and one per processor, are merged. We perform this merging by using a binary tree execution sequence of depth $\lceil \lg q \rceil$, where in the i -th iteration pairs of sub-sequences of size $2^{i-1} \cdot s$ are merged into sequences of size $2^i \cdot s$, where each such sequence is evenly distributed among 2^i processors. This algorithmic idea can be implemented by employing the BSP merging algorithm of section 2.3.5. The algorithm executes in time $O(s) + O(C_{ppf}(q))$ and $L \cdot O(1) + g \cdot O(s) + O(M_{ppf}(q)) + M_{bcast}^{O(s)}(q)$ for computation and communication respectively. For the case $s = \Omega(\lg q)$, the computation and communication time (excluding synchronization costs) of stage 2 of RECURSIVE_SORT is reduced by a multiplicative factor of $O(\lg p)$. We thus obtain the following analogue of theorem 2.6.3.

Theorem 2.6.4 *For arbitrarily small constant $\zeta > 0$ and any $\alpha > \zeta$, there exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p such that $p = O(n / \lg^{2+\alpha} n)$, requires time $(c_* + 2 \cdot (c_*/\alpha) + o(1)) \cdot ((n/p) \cdot \lg n)$ and $L \cdot O(\lg^2 p \cdot (\lg n / \lg(n/p))) + g \cdot O((n/p) \cdot (\lg n / \lg(n/p)))$, for computation and communication respectively.*

Proof (of theorem 2.6.4) The proof is similar to that of theorem 2.6.6 and is omitted. $\square$

Optimality Claim 2.6.3 *Theorem 2.6.4 implies that there exists a p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n , p , L and g such that $p = O(n/\lg^{2+\alpha} n)$, $L = o((n/p) \cdot (\lg(n/p)/\lg^2 p))$ and $g = o(\lg(n/p))$, has $\pi = 1 + 2 \cdot (c_*/\alpha) + o(1)$ and $\mu = o(1)$.*

2.6.6.2 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the optimal sorting algorithm of section 2.6.6.

Implementation The prototype adopted for the *BSPlib* sorting procedure is outlined in figure 2.28.

```
void BSP_det_sort (byte *src, byte **trg, int n, int sb, void (*merge)(), void (*sort)(), int (*cmp)()) ;
    src      - input array.
    trg      - output array.
    n        - input size.
    sb       - item size.
    merge    - generic merge function.
    sort     - generic sort function.
    cmp      - generic compare function.
```

Figure 2.28: Procedure for BSP deterministic sorting.

The aim of function (procedure) `BSP_det_sort` is to provide a data-type independent high-performance *BSPlib* sorting algorithm implementation.¹¹

The source code – in C – of function `BSP_det_sort` is provided for illustration purposes in appendix B.1.1.

Experimental Results The theoretical performance of the algorithm is summarized in theorems 2.6.3 and 2.6.4. To determine the practical significance of the aforementioned results we benchmarked our implementation (`BSP_det_sort`) on two different machines with a varying number of processors and input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). Figures 2.30 and 2.32 present the execution time of procedure `BSP_det_sort` on the two machines (using as the underlying sequential sorting algorithm merge sort) – figure 2.29 reflects the execution time (%) of procedure `BSP_det_sort` with respect to its constituent phases. Similarly, figures 2.31 and 2.33 expose the efficiency of the corresponding timing results.

As shown in figures 2.30 and 2.32, the execution time of procedure `BSP_det_sort` using a fixed number of processors scales with the input size (approximately as $n \cdot \lg n$). Accordingly, the execution time of procedure `BSP_det_sort` for a fixed input size scales inversely with the number of processors (approximately as $1/p$). Both experimental results are in accordance with the theoretical analysis of section 2.6.6. In terms of efficiency, procedure `BSP_det_sort` utilizes approximately (within a 5%):

¹¹The terms generic and data-type independent imply that procedure `BSP_det_sort` can be employed to handle *any* data type. To this end, a comparison function `cmp` is used to facilitate the comparison of two elements e_1 and e_2 . In particular, the `cmp` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2 .

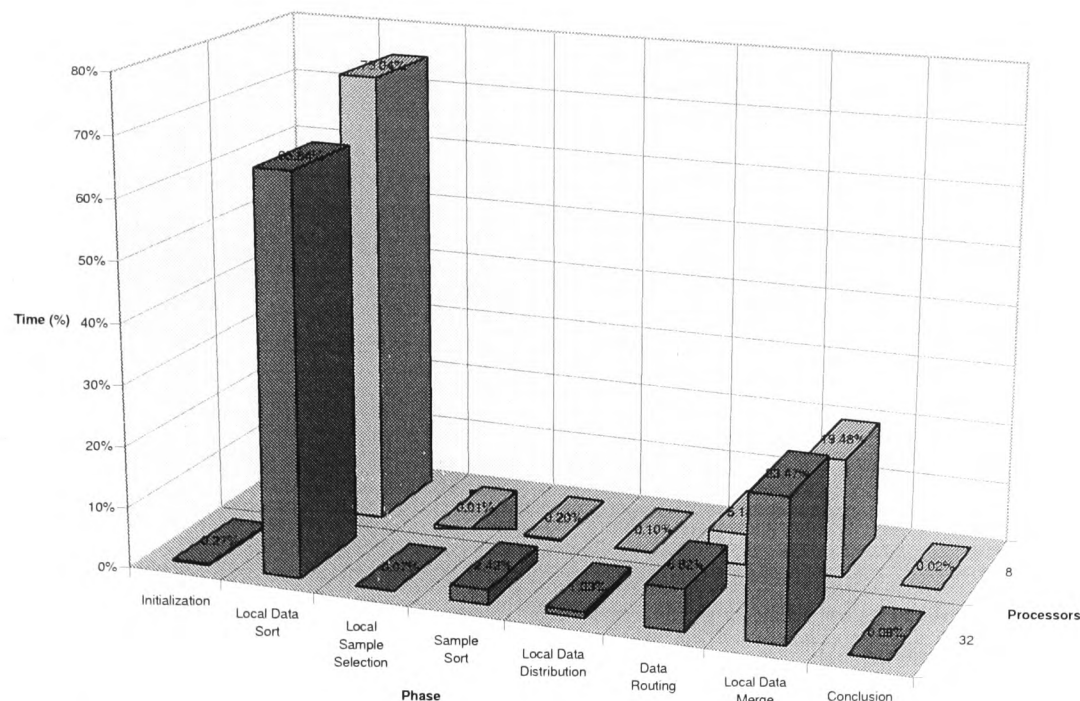


Figure 2.29: Execution time (% per phase) on a Cray T3D for deterministic sorting – uniform distribution (U), input size 2^{21} .

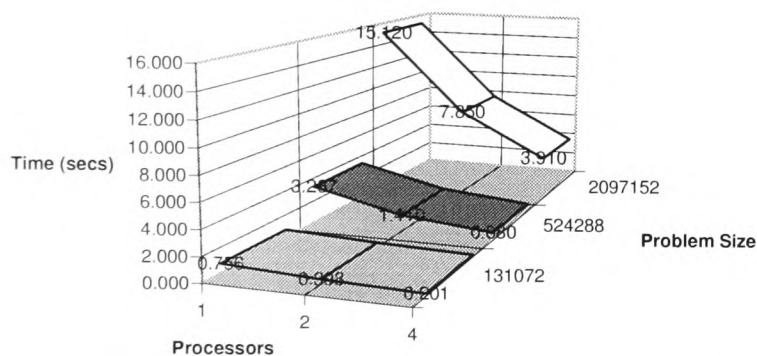


Figure 2.30: Execution time on a SGI Power Challenge for deterministic sorting – uniform distribution (U).

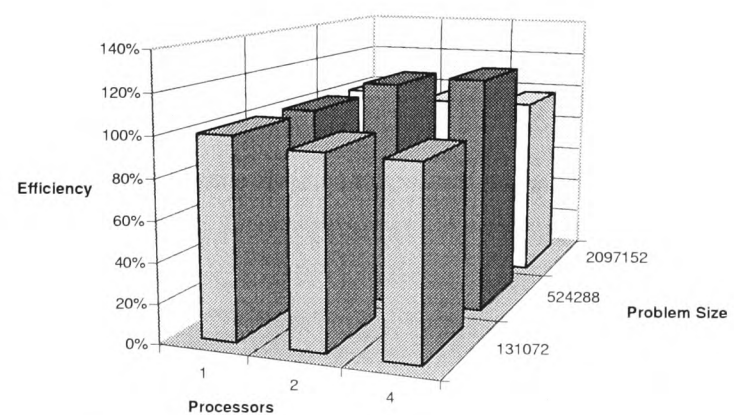


Figure 2.31: Efficiency on a SGI Power Challenge for deterministic sorting – uniform distribution (U).

- 85%–95% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 95%–100% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),
- 85%–95% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 95%–100% of the available processor resources on the Cray T3D machine (for large problem sizes).

In order to test the obliviousness of the sorting algorithm to the input data we benchmarked our implementation (`BSP_det_sort`) on a machine with a varying number of processors and input sets (consisting of 2^{21} integer valued items – unless otherwise indicated). Figure 2.34 presents the execution time of procedure `BSP_det_sort`. Similarly, figure 2.35 exposes the efficiency of the corresponding timing result.

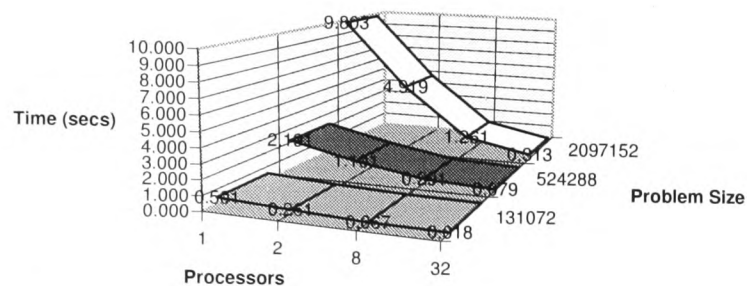


Figure 2.32: Execution time on a Cray T3D for deterministic sorting – uniform distribution (U).

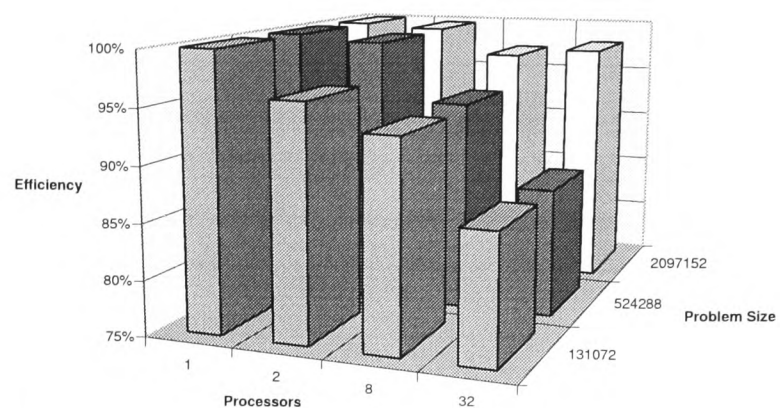


Figure 2.33: Efficiency on a Cray T3D for deterministic sorting – uniform distribution (U).

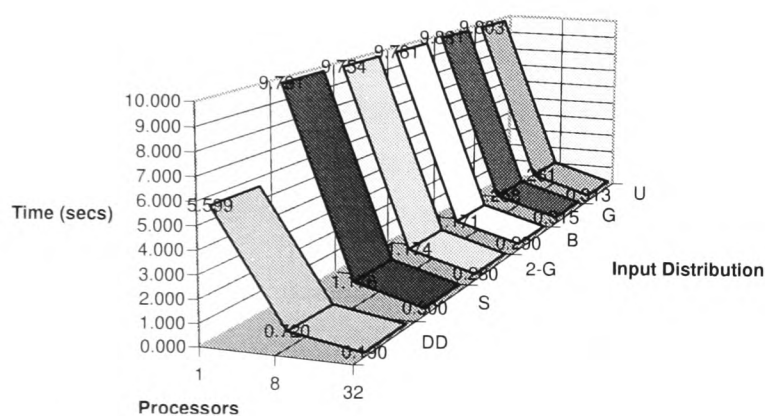


Figure 2.34: Execution time on a Cray T3D for deterministic sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

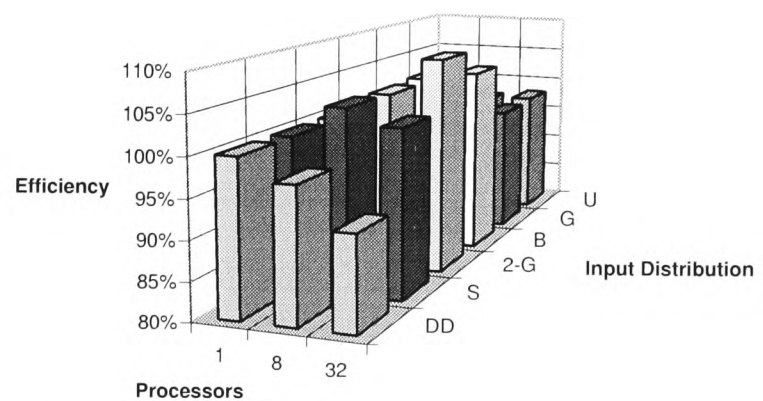


Figure 2.35: Efficiency on a Cray T3D for deterministic sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

Obviously, the performance of the algorithm is quite consistent across the different input data distributions with the exception of the deterministic-duplicates distribution (we note that the observable inconsistency in procedure `BSP_det_sort`, with respect to the deterministic-duplicates distribution, derives from the sequential merging and sorting implementations (`merge` and `sort` respectively) employed in the main body of `BSP_det_sort` and is irrelevant to the otherwise structure of the procedure).

2.6.7 An Optimal Randomized Sorting Algorithm

In this section, we provide an optimal randomized BSP solution to the problem of sorting a sequence X that is based on ideas of [73, 187], and utilizes *ordered h -level graphs* (refer to section 2.9.4) and *over-sampling* techniques [75, 183] – the required notation and concepts related to random over-sampling are presented in appendix A.2.1. The algorithm employs a partitioning scheme that meets the requirements of efficiency and insensitivity against initial data distribution and achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal randomized sorting algorithm is based on over-sampling techniques and is recursive in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. In

each recursive call of the algorithm, each processor constructs (in a randomized way) a sub-sequence that is an appropriate over-sample of its input. Subsequently, the so constructed sub-sequences are united and sorted, by means of either the sorting algorithm of section 2.6.5 (for small sample sizes) or the optimal sorting algorithm of section 2.6.8.2 (for large sample sizes), and a sub-sequence of size $k - 1$ (where k is appropriately chosen) is constructed that is an appropriate sample of the resulting sorted sequence. A d -ary tree (for some appropriate d) is actualized over the so constructed sequence and this tree is employed to partition and re-distribute the input into k non-overlapping partitions. Accordingly, processors are logically re-arranged into k non-intersecting groups. Finally, after a pre-specified number of iterations the iterative process results either in a single sequence per processor (which is sorted by means of some optimal sequential sorting algorithm) or in an input sequence of small enough size (which is sorted by means of the sub-optimal bitonic-sort algorithm of section 2.3.4).

Our main result is summarized in the following theorem.

Theorem 2.6.5 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p such that $p = o(n \cdot (\lg \lg n / \lg n))$, requires time [with probability $1 - O(n^{-\rho})$] $(c_* + o(1)) \cdot (n/p) \cdot \lg n$ and $L \cdot O(\lg p \cdot (\lg n / \lg(n/p))) + g \cdot O((n/p) \cdot (\lg n / \lg(n/p)))$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.6.5 follows easily as a consequence of these results.

Optimality Claim 2.6.4 *Theorem 2.6.5 implies that there exists a randomized p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n , p , L and g such that $p = o(n \cdot (\lg \lg n / \lg n))$, $L = o((n/p) \cdot (\lg(n/p) / \lg p))$ and $g = o(\lg(n/p))$, has $\pi = 1 + o(1)$ and $\mu = o(1)$.*

Remark We note that the underlying assumption in the analysis of this section is that $c_* \geq c_{srch}$. The validity and implications of this assumption are treated, both theoretically and experimentally, in the discussion of section 2.7.

The randomized sorting algorithm is based on an efficient partitioning scheme that splits – almost evenly and independently of the input distribution – an arbitrary number of sorted sequences among the processors. The algorithm consists of the following phases:

- (1) *Sample Selection.* Processors cooperate to select a sample of appropriately chosen size.
- (2) *Partitioning.* Processors cooperate to evenly split the input among the processors (as induced by the sample).
- (3) *Sorting.* Each processor, in parallel with all the other processors, sorts a set of total size $(1 + o(1)) \cdot (n/p)$.

The recursive version of the algorithm realizes m recursive calls of phases 2 and 3. In each recursive call the data set is partitioned into k buckets of approximately equal size. In the next iteration a similar process of sub-partitioning each of these k buckets into k further buckets is performed, and so on. The maximum size of every bucket formed in any of the m recursive calls is shown to be $(1 + O(1/\omega_n)) \cdot (n/p)$, for any slowly growing function ω_n , with high probability. The procedure to manage the aforementioned scheme is presented in the following result.

Proposition 2.6.4 *Let ω_n be any function of n such that $\omega_n = \Omega(1)$. Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n , $n_{(j)}$, p , $q_{(j)}$ and $k_{(j)}$ such that $p = O(n \cdot (\lg \lg n / \lg n))$, $n_{(j)} \leq n$, $q_{(j)} \leq p$, $k_{(j)}, q_{(j)} \leq n_{(j)}$ and $\omega_n = O(\lg n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$\begin{aligned} C_{srt}(n, p) = & \frac{1}{2^m} \cdot c_* \cdot \left(1 + O\left(\frac{1}{\omega_n}\right)\right) \cdot \frac{n}{p} \cdot \lg n + \sum_{j=0}^{m-1} O\left(\frac{\sqrt{n_{(j)}}}{q_{(j)}} \cdot \lg n_{(j)} + \lg n\right) \\ & \sum_{j=0}^{m-1} \left(\frac{2 \cdot \lg k_{(j)}}{\lg(n_{(j)}/q_{(j)})} \cdot C_{mway}(n_{(j)}, q_{(j)}) + C_{rdx_srt}(n_{(j)}, k_{(j)}, q_{(j)}) + C_{ppf}(q_{(j)})\right) + \\ & \lceil \lg \lg^2 n \rceil \cdot C_{btn} \left(\left\lfloor \lg \lg \lg n \right\rfloor, \left\lceil \frac{\lg \lg \lg n}{n_{(m)}/q_{(m)}} \right\rceil \right) + O\left(\frac{n}{p} \cdot \lg \lg^2 n + \lg \lg^3 n\right) \end{aligned}$$

and

$$\begin{aligned} M_{srt}(n, p) = & L \cdot \sum_{j=0}^{m-1} O\left(\lg q_{(j)} + \frac{\lg n_{(j)}}{\lg(n_{(j)}/q_{(j)})}\right) + g \cdot \sum_{j=0}^{m-1} O\left(\frac{\sqrt{n_{(j)}}}{q_{(j)}} \cdot \frac{\lg n_{(j)}}{\lg(n_{(j)}/q_{(j)})} + \lg n\right) + \\ & \sum_{j=0}^{m-1} \left(\frac{2 \cdot \lg k_{(j)}}{\lg(n_{(j)}/q_{(j)})} \cdot M_{mway}(n_{(j)}, q_{(j)}) + M_{rdx_srt}(n_{(j)}, k_{(j)}, q_{(j)}) + M_{ppf}(q_{(j)})\right) + \\ & \lceil \lg \lg^2 n \rceil \cdot C_{btn} \left(\left\lfloor \lg \lg \lg n \right\rfloor, \left\lceil \frac{\lg \lg \lg n}{n_{(m)}/q_{(m)}} \right\rceil \right), \end{aligned}$$

for computation and communication respectively, where $m \leq \lceil \lg \lg n \rceil$, $n_{(j)}/q_{(j)} = (1 + O(1/\omega_n)) \cdot (n/p) \pm O(\lg \lg n)$, $\sum_j \lg q_{(j)} = O(\lg n)$, $\sum_j \lg k_{(j)} = (1 - 1/2^m) \cdot \lg n + O(m^2)$ and $\tau_l = c_{srch} \cdot (1/2) \cdot \lg(n_{(j)}/q_{(j)})$ in the expression for T_{mway} (refer to proposition 2.9.2). The overall time complexity of the algorithm is denoted by $T_{srt}(n, p) = C_{srt}(n, p) + M_{srt}(n, p)$.

Proof The algorithm is recursive in nature and each recursive call is divided into a sequence of stages. We assume that the input set is initially evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm. Moreover, we assume that the items are distinct since we can always make them so by, for example, appending to them the code for their memory location. We later explain how we can remove such an assumption and still maintain optimality.

Algorithm RND_SORT

begin

RND_RECURSIVE_SORT (0, $p - 1$, n)

end

In procedure RECURSIVE_SORT below the variables l , u and N have initially values 0, $p - 1$ and n respectively; in any recursive call of the algorithm, the former two variables give the range of the processors that participate in the call and the latter variable gives the input size.

Procedure RND_RECURSIVE_SORT (l , u , N)

begin

[Stage 1] Let $q = u - l + 1$. If ($q = 1$) or ($q \neq 1$ and $N \leq \lg \lg \lg n$), then the input is sorted by a sequential algorithm in the former case and by a sub-optimal parallel algorithm in the latter case, and subsequently, the algorithm terminates.

[Analysis of Stage 1] The sorting process is realized by one of two methods depending on the values of l , r and N .

Case 1 If $q = 1$, then processor p_l sorts, by way of a deterministic sequential sorting algorithm, the items in its possession.

Case 2 If $q \neq 1$ and $N \leq \lg^{\lg \lg n} n$, then the q processors participating in this recursive call of the algorithm sort, by way of a BSP adaptation of bitonic-sort sort [22], the items in their possession.

The analysis of this stage is treated in detail in the discussion following the description of this algorithm.

[Stage 2] A sample of size $k \cdot s - 1 = \lfloor (\lfloor \sqrt{N} \rfloor + 1)/s \rfloor \cdot s - 1$ is selected, where the over-sampling factor s is set to $\lceil \lg^\delta n \rceil$, for any $1 < \delta < (\lg \lg n - 1)/2$. The sample selection method is as follows.

- (1) Each processor selects approximately the same number of processor identifiers $\lceil (k \cdot s - 1)/q \rceil$ (or $\lfloor (k \cdot s - 1)/q \rfloor$) uniformly at random; a sample of size $k \cdot s - 1 \leq \lfloor \sqrt{N} \rfloor$, over all processors, is selected.
- (2) The processor identifiers are routed to the corresponding processors.
- (3) Each processor p_i , $0 \leq i < p$, that receives t_i identifiers selects among its items uniformly at random without replacement t_i items.
- (4) A parallel-prefix sum on the number t_i of items selected per processor is realized.
- (5) The result of the parallel-prefix sum is used to evenly distribute the selected sample among the processors.

[Analysis of Stage 2] Step 1 requires time $O((k \cdot s)/q)$ for random processor identifier generation. We distinguish two cases in the analysis of step 2.

Case 1 If $\lceil (k \cdot s - 1)/q \rceil = \Omega(\lg n)$, then the inequality of theorem A.1.3 implies that each processor receives $O((k \cdot s)/q) = O(\sqrt{N}/q)$ processor identifiers with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Case 2 If $\lceil (k \cdot s - 1)/q \rceil = O(\lg n)$, then the inequality of theorem A.1.2 implies that each processor receives $O(\lg n)$ processor identifiers with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Thus, this step requires communication time $L + g \cdot O(\max\{(k \cdot s)/q, \lg n\})$. Similarly, we conclude that step 3 requires time $O(\max\{(k \cdot s)/q, \lg n\})$. Step 4 requires time $T_{ppf}(q)$ and step 5 can be shown (along the lines of step 2) to require communication time $L + g \cdot O(\max\{(k \cdot s)/q, \lg n\})$.

[Stage 3] The sample of size $k \cdot s - 1 = \lfloor \sqrt{N} \rfloor$ is sorted.

[Analysis of Stage 3] This stage is realized by one of two sorting methods depending on the size of the sample. We distinguish two cases in the analysis of this stage.

Case 1 If $q \geq (k \cdot s)^2$, then the algorithm implied in theorem 2.6.2 is employed. The time required for this case is therefore bounded above by $O(\lg q)$ and $L \cdot O(\lg q) + g \cdot O(\lg q)$, for computation and communication respectively.

Case 2 Otherwise, the algorithm implied in theorem 2.6.7 is employed.¹² The time required for this case is therefore bounded above by $O(((k \cdot s)/q') \cdot \lg(k \cdot s))$ and $L \cdot O(\lg(k \cdot s)/\lg((k \cdot s)/q')) + g \cdot O(((k \cdot s)/q') \cdot (\lg(k \cdot s)/\lg((k \cdot s)/q')))$, for computation and communication respectively, where $q' = \min\{q, (k \cdot s)/2\}$.

[Stage 4] A balanced d -ary tree is constructed over the $k = \lfloor (\lfloor \sqrt{N} \rfloor + 1)/s \rfloor$ buckets (ranges) induced by the $k - 1$ sorted splitters that evenly partition the sample.

[Analysis of Stage 4] Let $d = 2 \cdot \lceil (N/q)^\alpha \rceil$, for some appropriately chosen constant $0 < \alpha < 1$ to be specified later. In order to simplify the exposition we assume that k is an integral power of d – this assumption can be removed without affecting the asymptotic analysis of the algorithm. The balanced d -ary tree construction proceeds as follows. Let l be an arbitrary level of the tree. We define every d -th item of level l of the tree to be a *pivot* item. The construction of the balanced tree proceeds from the leaves to the root as follows. For each level l of the tree, level $l - 1$ is constructed by only considering the pivot items of level l . Since the height of the tree is $\lceil \log_d k \rceil$ the construction takes $\lceil \log_d k \rceil$ phases, of which $\lceil \log_d q \rceil$ phases are non-local (require inter-processor communication). Each such phase requires time $O(d)$ and $L + g \cdot O(d)$ for computation and communication respectively. Since the overall cost of local computations is bounded above by $O(\lceil k/q \rceil)$, this stage requires overall time $O(\lceil k/q \rceil) + O(d \cdot \log_d q)$ and $L \cdot \lceil \log_d q \rceil + g \cdot O(d \cdot \log_d q)$, for computation and communication respectively.

[Stage 5] Each processor decides the position of every item it holds with respect to the $k - 1$ splitters, by way of multi-way searching.

[Analysis of Stage 5] This stage is realized by employing the algorithm implied by proposition 2.9.2. Given that the height of the balanced d -ary tree of stage 4 is $\log_d k$ we obtain that this stage requires time $\log_d k \cdot T_{mway}(N, q)$, where $\tau_l = c_{srch} \cdot \lg d$ in proposition 2.9.2.

[Stage 6] The input items are redistributed according to their bucket destination.

[Analysis of Stage 6] The process of this stage is equivalent to integer sorting in the range $[0, 1, \dots, k - 1]$, and therefore, requires time $C_{rdx_srt}(N, k, q)$ and $L + g \cdot O(N/q) + M_{rdx_srt}(N, k, q)$, for computation and communication respectively.

[Stage 7] Recursively sort each of the k buckets generated in stages 1–6. If $q/k \geq 1$, then each bucket is assigned $\lceil q/k \rceil$ (or $\lfloor q/k \rfloor$) consecutive processors; otherwise each processor is assigned $\lceil k/q \rceil$ (or $\lfloor k/q \rfloor$) consecutive buckets.

[Analysis of Stage 7] The depth of the recursion is $m \leq \lceil \lg \lg n \rceil$ (refer to claim 2.6.3). In the remainder of this analysis, we assume that either q/k or k/q is integer valued. This assumption can be removed without affecting the asymptotic analysis of the algorithm.

end

¹²The choice of the optimal merge-sort algorithm of section 2.6.7 is not essential for realizing the required optimal time bounds. A tighter analysis reveals that even a sub-optimal sorting algorithm, such as bitonic-sort, is sufficient for achieving optimality.

Analysis The correctness of the algorithm follows from its description. We proceed by establishing upper bounds on the computation and communication requirements of procedure RND_SORT. We start by showing the following bound on the maximum number of items $n_{(j)}$ per bucket in the j -th recursive call of the algorithm (in the subsequent analysis the j -th recursive call is denoted by the subscript notation $_{(j)}$). We assume $n_{(0)} = n$.

Claim 2.6.2 *Let ω_n be any function of n such that $\omega_n = \Omega(1)$. Let m be the depth of the recursion of procedure RND_RECURSIVE_SORT, and consider integers n , $n_{(j)}$ and any δ such that $n_{(j)} \geq \lg^{\lg^{\lg n} n} n$ and $1 < \delta < (\lg \lg n - 1)/2$. Then, in the j -th recursive call, $1 \leq j \leq m$, for any constant $\rho > 0$,*

$$\Pr \left[n_{(j)} \leq (1 + O(\varepsilon))^{2-1/2^{j-1}} \cdot n_{(0)}^{1/2^j} \cdot \left\lceil \lg^\delta n \right\rceil^{2-1/2^{j-1}} + 2 \cdot (1 + O(\varepsilon))^2 \cdot \left\lceil \lg^\delta n \right\rceil^2 \right] \geq 1 - n^{-\rho \cdot \lg^{\delta-1} n / (m \cdot \lceil \omega_n \rceil)^2},$$

where $\varepsilon = 1/(m \cdot \lceil \omega_n \rceil)$.

Proof Consider an arbitrary recursive call j of procedure RND_RECURSIVE_SORT such that $n_{(j)} \geq \lg^{\lg \lg n} n$. By definition, the number of buckets generated in the j -th recursive call is $k_{(j)} = \lfloor (\lfloor \sqrt{n_{(j)}} \rfloor + 1)/s \rfloor$. Thus, by way of lemma A.2.2, for $\varepsilon = 1/(m \cdot \lceil \omega_n \rceil)$, $k = k_{(j)}$ and $s = \lceil \lg^\delta n \rceil$, in the $(j+1)$ -st recursive call, we have that $n_{(j+1)} \leq \lceil (1 + O(\varepsilon)) \cdot (n_{(j)} - k_{(j)} + 1)/k_{(j)} \rceil + 1$, with probability $1 - n^{-\rho \cdot \lg^{\delta-1} n / (m \cdot \lceil \omega_n \rceil)^2}$, for any constant $\rho > 0$. By substituting in this inequality, for $k_{(j)} = \lfloor (\lfloor \sqrt{n_{(j)}} \rfloor + 1)/s \rfloor$, we obtain that $n_{(j+1)} \leq (1 + O(\varepsilon)) \cdot (s \cdot \sqrt{n_{(j)}} + 2 \cdot s^2)$. By way of an inductive argument on j , we therefore obtain that in the j -th recursive call,

$$n_{(j)} \leq (1 + O(\varepsilon))^{2-1/2^{j-1}} \cdot n_{(0)}^{1/2^j} \cdot s^{2-1/2^{j-1}} + 2 \cdot (1 + O(\varepsilon))^2 \cdot s^2,$$

where $n_{(0)} = n$. By substituting in this expression for $n_{(0)} = n$ and $s = \lceil \lg^\delta n \rceil$ we obtain the stated bound. $\square$

Hence, we conclude the following on the depth m of the recursion.

Claim 2.6.3 *Consider integer n and any δ such that $1 < \delta < (\lg \lg n - 1)/2$. Then, the recursion in procedure RND_RECURSIVE_SORT has depth*

$$\left\lceil \lg \lg n - \lg \left(\lg \left(\max \left\{ \lg^{\lg \lg n} n, \frac{n}{p} \right\} \right) - (2 \cdot \delta + 1) \cdot \lg \lg n - O(1) \right) \right\rceil.$$

Proof Let m be the depth of the recursion. From the discussion of claim 2.6.2 we have that in the j -th recursive call,

$$n_{(j)} \leq (1 + O(\varepsilon))^{2-1/2^{j-1}} \cdot n_{(0)}^{1/2^j} \cdot s^{2-1/2^{j-1}} + 2 \cdot (1 + O(\varepsilon))^2 \cdot s^2,$$

where $n_{(0)} = n$, $\varepsilon = 1/(m \cdot \lceil \omega_n \rceil)$ and $s = \lceil \lg^\delta n \rceil$. Then, for $j \leq \lg n$, we obtain the following bound.

$$n_{(j)} \leq (1 + O(\varepsilon))^{2-1/2^{j-1}} \cdot n_{(0)}^{1/2^j} \cdot s^{2-1/2^{j-1}} + 2 \cdot (1 + O(\varepsilon))^2 \cdot s^2 = O(1) \cdot n^{1/2^j} \cdot \lg^{(2 \cdot \delta + 1)} n. \quad (2.2)$$

Additionally, for all j such that $j < m$, by way of stages 1 and 7 of procedure RND_RECURSIVE_SORT, we also have that

$$n_{(j)} \geq \max \left\{ \lg^{\lg \lg n} n, \frac{n}{p} \right\}.$$

By combining the previous observations and solving the following equation (with respect to j)

$$\max \left\{ \lg^{\lg \lg n} n, \frac{n}{p} \right\} = O(1) \cdot n^{1/2^j} \cdot \lg^{(2 \cdot \delta + 1)} n,$$

we obtain the stated upper bound on the depth of the recursion. $\square$

We proceed by deriving upper and lower bounds on the number of items per processor at the j -th recursive call. Let parameters $n_{(j)}$, $q_{(j)}$ and $k_{(j)}$ denote, respectively, the number of items, the number of processors, and the number of splitters in the j -th recursive call of procedure RND_RECURSIVE_SORT.

Initially, $n_{(0)} = n$ and $q_{(0)} = p$. In the $(j-1)$ -st recursive call of the algorithm $q_{(j-1)}$ processors are assigned to each bucket. In the next recursive call, $k_{(j-1)} - 1$ splitters are selected that split the buckets into $k_{(j-1)}$ further sub-buckets and $q_{(j)} = q_{(j-1)}/k_{(j-1)}$ processors are assigned to each such newly formed sub-bucket. Although in general we have that $q_{(j-1)} > k_{(j-1)}$, it is, however, possible that in the last recursive call to have $q_{(j-1)} \leq k_{(j-1)}$.

By way of claim A.2.2, for $\varepsilon = 1/(m \cdot \lceil \omega_n \rceil)$, we have that $n_{(j+1)} \leq \lceil (1 + O(\varepsilon)) \cdot (n_{(j)} - k_{(j)} + 1)/k_{(j)} \rceil + 1$, with high probability, and therefore, we conclude that $n_{(j+1)} \leq (1 + O(\varepsilon)) \cdot (n_{(j)}/k_{(j)}) + 2$. Similarly, by way of claim A.2.1, we have that $n_{(j+1)} \geq \lfloor (1 - O(\varepsilon)) \cdot (n_{(j)} - k_{(j)} + 1)/k_{(j)} \rfloor + 1$, and thus, we conclude that $n_{(j+1)} \geq (1 - O(\varepsilon)) \cdot (n_{(j)}/k_{(j)})$. We proceed by establishing upper and lower bounds on the ratio $n_{(j)}/q_{(j)}$.

Since $n_{(j+1)} \leq (1 + O(\varepsilon)) \cdot (n_{(j)}/k_{(j)}) + 2$ and $q_{(j)} = q_{(j-1)}/k_{(j-1)}$, we have that $n_{(j)}/q_{(j)} \leq (1 + O(\varepsilon)) \cdot n_{(j-1)}/(q_{(j)} \cdot k_{(j-1)}) + 2/q_{(j)}$, and therefore, $n_{(j)}/q_{(j)} \leq (1 + O(\varepsilon)) \cdot n_{(j-1)}/q_{(j-1)} + 2 \cdot k_{(j-1)}/q_{(j-1)}$. The latter recurrence relation gives that

$$\frac{n_{(j)}}{q_{(j)}} \leq (1 + O(\varepsilon))^j \cdot \frac{n_{(0)}}{q_{(0)}} + 2 \cdot \sum_{i=0}^{j-1} (1 + O(\varepsilon))^{j-1-i} \frac{k_{(i)}}{q_{(i)}}.$$

By way of stages 1 and 7 of procedure RND_RECURSIVE_SORT, it is true that $k_{(j)} < q_{(j)}$, except possibly for the last recursive call. Additionally, for all $j \leq m$, we have that $(1 + O(\varepsilon))^{j-1-i} = O(1)$. By way of claim 2.6.3 we have that $m \leq \lceil \lg \lg n \rceil$ and we thus conclude that for all j such that $k_{(j)} < q_{(j)}$ the following inequality holds.

$$\frac{n_{(j)}}{q_{(j)}} \leq (1 + O(\varepsilon))^j \frac{n}{p} + O(\lg \lg n). \quad (2.3)$$

Similarly, for all recursive calls, except possibly for the last one, we can prove a corresponding lower bound. Since $n_{(j+1)} \geq (1 - O(\varepsilon)) \cdot n_{(j)}/k_{(j-1)}$ and $q_{(j)} = q_{(j-1)}/k_{(j-1)}$, we conclude – using arguments similar to those employed for the derivation of the upper bound – that for all j such that $k_{(j)} < q_{(j)}$ the following inequality holds.

$$\frac{n_{(j)}}{q_{(j)}} \geq (1 - O(\varepsilon))^j \frac{n}{p} - O(\lg \lg n). \quad (2.4)$$

The former analysis applies to all but (possibly) the last recursive call, since $k_{(m-1)} \geq q_{(m-1)}$ is possible. The following analysis handles the case $k_{(m-1)} \geq q_{(m-1)}$. In the last recursive call, we have $k_{(m-1)}$ buckets and $q_{(m-1)}$ processors. Each processor is assigned $k_{(m-1)}/q_{(m-1)}$ buckets. Since $n_{(m)} \leq (1 + O(\varepsilon)) \cdot n_{(m-1)}/k_{(m-1)} + 2$, the size of all $k_{(m-1)}/q_{(m-1)}$ such buckets on any one processor is at most $n_{(m)} \cdot (k_{(m-1)}/q_{(m-1)}) \leq (1 + O(\varepsilon)) \cdot n_{(m-1)}/q_{(m-1)} + 2 \cdot k_{(m-1)}/q_{(m-1)}$, which by equation 2.3 is bounded above as follows.

$$n_{(m)} \cdot \frac{k_{(m-1)}}{q_{(m-1)}} \leq (1 + O(\varepsilon))^m \cdot \frac{n}{p} + O(\lg \lg n) + 2 \cdot \frac{k_{(m-1)}}{q_{(m-1)}}. \quad (2.5)$$

Similarly, a corresponding lower bound can be derived.

$$n_{(m)} \cdot \frac{k_{(m-1)}}{q_{(m-1)}} \geq (1 - O(\varepsilon))^m \cdot \frac{n}{p} - O(\lg \lg n). \quad (2.6)$$

We next derive an upper bound for the term $k_{(m-1)}/q_{(m-1)}$. By definition, we have that $k_{(m-1)} = \lfloor (\lfloor \sqrt{n_{(m-1)}} \rfloor + 1)/s \rfloor$, and thus, $k_{(m-1)} \leq (\sqrt{n_{(m-1)}} + 1)/s$. We also have that $n_{(m)} \leq (1 + O(\varepsilon)) \cdot n_{(m-1)}/k_{(m-1)} + 2$. Since $n_{(j+1)} \leq (1 + O(\varepsilon)) \cdot n_{(j)}/k_{(j)} + 2$ and $k_{(m-2)}/q_{(m-2)} \leq 1$, we obtain by equation 2.3 the following bound.

$$\begin{aligned} \frac{k_{(m-1)}}{q_{(m-1)}} &\leq \frac{(\sqrt{n_{(m-1)}} + 1)/s}{q_{(m-2)}/k_{(m-2)}} \\ &\leq \frac{(\sqrt{(1 + O(\varepsilon)) \cdot (n_{(m-2)}/k_{(m-2)}) + 2} + 1) \cdot k_{(m-2)}}{s \cdot q_{(m-2)}} \\ &= O\left(\frac{\sqrt{n/p} + \sqrt{\lg \lg n}}{s}\right). \end{aligned}$$

By substituting in equations 2.5 and 2.6, for $k_{(m-1)}/q_{(m-1)}$, and since $(1 + O(\varepsilon))^m = (1 + O(1/\omega_n))$, we obtain the following bounds respectively.

$$n_{(m)} \cdot \frac{k_{(m-1)}}{q_{(m-1)}} \leq (1 + O(1/\omega_n)) \cdot \frac{n}{p} + O(\lg \lg n) + O\left(\sqrt{\frac{n}{p \cdot s^2}} + \frac{\sqrt{\lg \lg n}}{s}\right) \quad (2.7)$$

and

$$n_{(m)} \cdot \frac{k_{(m-1)}}{q_{(m-1)}} \geq (1 - O(1/\omega_n)) \cdot \frac{n}{p} - O(\lg \lg n) - O\left(\sqrt{\frac{n}{p \cdot s^2}} + \frac{\sqrt{\lg \lg n}}{s}\right). \quad (2.8)$$

This analysis completes the separate examination of the last recursive call of the algorithm. We next establish the time complexity of stage 1 of algorithm RND_RECURSIVE_SORT. We distinguish two cases.

Case 1 If $q = 1$, then the $k_{(m-1)}/q_{(m-1)}$ buckets assigned to processor p_l are sorted by way of a deterministic sequential sorting algorithm. Each such bucket can be sorted in time $c_* \cdot n_{(m)} \cdot \lg n_{(m)}$ for a total time of $c_* \cdot n_{(m)} \cdot \lg n_{(m)} \cdot (k_{(m-1)}/q_{(m-1)})$. An upper bound on $n_{(m)} \cdot (k_{(m-1)}/q_{(m-1)})$ is given by equation 2.7, and therefore, we obtain the following bound on the time complexity of this step.

$$c_* \cdot n_{(m)} \cdot \lg n_{(m)} \cdot \frac{k_{(m-1)}}{q_{(m-1)}} \leq c_* \cdot \left((1 + O(\varepsilon))^m \cdot \frac{n}{p} + O \left(\lg \lg n + \sqrt{\frac{n + p \cdot \lg \lg n}{p \cdot s^2}} \right) \right) \cdot \lg n_{(m)}.$$

By equation 2.2 we have that $n_{(m)} = O(n^{1/2^m} \cdot \lg^{2 \cdot \delta + 1} n)$. Additionally, by way of claim 2.6.3, we have that $\lg n / 2^m \leq \lg(n/p) + (2 \cdot \delta + 1) \cdot \lg \lg n + O(1)$. Hence, since $(1 + O(\varepsilon))^m = (1 + O(1/\omega_n))$, $s = \lceil \lg^\delta n \rceil$ and $\delta > 1$ we obtain the following bound.

$$c_* \cdot n_{(m)} \cdot \lg n_{(m)} \cdot \frac{k_{(m-1)}}{q_{(m-1)}} \leq \frac{c_*}{2^m} \cdot \left(1 + O \left(\frac{1}{\omega_n} \right) \right) \cdot \frac{n}{p} \cdot \lg n + O \left(\delta \cdot \left(\frac{n}{p} + \lg \lg n \right) \cdot \lg \lg n \right).$$

Case 2 If $q \neq 1$ and $N \leq \lg^{\lg \lg n} n$, then the single bucket assigned to processors p_l through p_u is sorted by way of a BSP adaptation of bitonic-sort sort [22]. Each processor is assigned a segment of the single bucket of size $n_{(m)}/q_{(m)}$. Each such segment can be locally sorted in time $c_* \cdot (n_{(m)}/q_{(m)}) \cdot \lg(n_{(m)}/q_{(m)})$. Since $l \neq u$ we have that $k_{(m)} < q_{(m)}$, and therefore, by equation 2.3 and the fact that $n_{(m)} \leq \lg^{\lg \lg n} n$, we obtain the following bound on the time complexity of this local sorting step.

$$\begin{aligned} \frac{n_{(m)}}{q_{(m)}} \cdot \lg \frac{n_{(m)}}{q_{(m)}} &\leq \left((1 + O(\varepsilon))^m \cdot \frac{n}{p} + O(\lg \lg n) \right) \cdot \lg n_{(m)} \\ &\leq \left((1 + O(\varepsilon))^m \cdot \frac{n}{p} + O(\lg \lg n) \right) \cdot \lg \lg^2 n \\ &= O \left(\left(\frac{n}{p} + \lg \lg n \right) \cdot \lg \lg^2 n \right). \end{aligned}$$

Finally, by way of proposition 2.3.1, the BSP adaptation of bitonic-sort sort algorithm requires time at most $\lceil \lg(\lg^{\lg \lg n} n) \rceil \cdot T_{btm}(\lfloor \lg^{\lg \lg n} n \rfloor, \lceil \lg^{\lg \lg n} n / (n_{(m)}/q_{(m)}) \rceil)$.

We complete the examination of stages 1–7 of procedure RND_RECURSIVE_SORT, by deriving upper bounds on $\sum_{j=0}^{m-1} \lg q_{(j)}$ and $\sum_{j=0}^{m-1} \lg k_{(j)}$. In particular, by way of claim 2.6.2, equation 2.2, and the fact that $q_{(j)} \leq n_{(j)}$, we obtain the following bound.

$$\sum_{j=0}^{m-1} \lg q_{(j)} = O(\lg n + \delta \cdot m \cdot \lg \lg n). \quad (2.9)$$

We next derive an upper bound on $\sum_{j=0}^{m-1} \lg k_{(j)}$. By definition, $k_{(j)} \leq (\sqrt{n_{(j)}} + 1)/s$, where $s = \lceil \lg^\delta n \rceil$, and by way of the analysis of stage 1, $n_{(j)} \geq \lg^{\lg \lg n} n$, for all $j < m$. Thus, we obtain the following bound.

$$\begin{aligned} \sum_{j=0}^{m-1} \lg k_{(j)} &\leq \sum_{j=0}^{m-1} \lg(\sqrt{n_{(j)}} + 1) - m \cdot \lg s \\ &\leq \sum_{j=0}^{m-1} \lg \sqrt{n_{(j)}} + \frac{1.5 \cdot m}{\lg^{\lg \lg n/2} n} - m \cdot \lg s. \end{aligned}$$

In the aforementioned bound we claimed inequality $\lg(x + y) \leq \lg x + 1.5 \cdot (y/x)$ which holds by way of $\log(1 + x) \leq x$, for any $x, y > 0$ [40]. The first term can be further bounded, by claim 2.6.2, as follows.

$$\sum_{j=0}^{m-1} \lg \sqrt{n_{(j)}} \leq m \cdot \lg(1 + O(\varepsilon)) + \left(1 - \frac{1}{2^m}\right) \cdot \lg n + m \cdot \lg s + m^2.$$

By combining the previous inequalities we obtain the following bound.

$$\sum_{j=0}^{m-1} \lg k_{(j)} \leq \left(1 - \frac{1}{2^m}\right) \cdot \lg n + O(m^2). \quad (2.10)$$

By summing up the computation and communication requirements of procedure `RND_RECURSIVE_SORT` and setting $\alpha = 1/2$ in the analysis of stage 4 we obtain the desired time bound. The bounds on $n_{(j)}/q_{(j)}$, $\sum_j \lg q_{(j)}$ and $\sum_j \lg k_{(j)}$ follow from equations 2.3, 2.9 and 2.10 respectively. The probability bound follows from the analysis of stage 2 and claims 2.6.2 and 2.6.3 (for constant $\delta > 3$).

The proposition follows. $\square$

Remark A tighter probabilistic analysis reveals that the success probability of the algorithm implied in the aforementioned proposition can be shown to be

$$1 - O\left(n^{-\rho \cdot (n/p) \cdot \lg \lg^2 n} + n^{-\rho \cdot \lg^{\delta-1} n / (m \cdot \lceil \omega_n \rceil)^2}\right),$$

for any constant $\rho > 0$.

Proof (of theorem 2.6.5) Let $\omega_n = \omega(1)$ and $\omega_n = O(\lg n)$. Then, the theorem follows from proposition 2.6.4 by substituting for T_{mway} , T_{rdx_srt} , T_{btl} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.9.2, 2.4.1, 2.3.1, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.6.7.1 Improving Randomized Sorting

We note that not only can the bounds given above be expressed in a variety of ways, but also the algorithm itself leaves room for several improvements. As stated the algorithm sorts the sample of size $\sqrt{N}$ by employing a BSP adaptation of a sorting algorithm of [167]. In most circumstances its performance can be improved by any one of several methods. For example, the sorting could be performed by a more efficient randomized parallel algorithm [75]. We also note that the recursion may terminate for smaller values of N than $\lg^{\lg \lg n} n$. Then, if we terminate the recursion for all N such that $N \leq \lg^\beta n$, for any $\beta = o(\lg \lg n)$, the contribution to the lower order terms of bitonic-sort algorithm is $O((n/p) \cdot (\beta^2 \lg \lg^2 n))$ rather than $O((n/p) \cdot \lg \lg^4 n)$. This improves the asymptotic performance of the algorithm whereas at the same time affects the probability of success insignificantly. The random sample selection in stage 2 of algorithm `RND_SORT` can also be realized with higher probability by utilizing integer sorting techniques. Furthermore, the analysis of the ordered h -level graph algorithm can be improved by taking into consideration that the underlying structure is a d -ary tree, and that the queries progress in a balanced fashion through the tree.

2.6.7.2 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the optimal sorting algorithm of section 2.6.7.

Implementation The prototype adopted for the *BSPlib* sorting procedure is outlined in figure 2.36.

```
void BSP_rnd_sort (byte *src, byte **trg, int n, int sb, void (*merge)(), void (*sort)(), int (*cmp)()) ;
    src      - input array.
    trg      - output array.
    n        - input size.
    sb       - item size.
    merge    - generic merge function.
    sort     - generic sort function.
    cmp      - generic compare function.
```

Figure 2.36: Procedure for BSP randomized sorting.

The aim of function (procedure) `BSP_rnd_sort` is to provide a data-type independent high-performance *BSPlib* sorting algorithm implementation.¹³

Experimental Results The theoretical performance of the algorithm is summarized in theorem 2.6.5. To determine the practical significance of the aforementioned result we benchmarked our implementation (`BSP_rnd_sort`) on two different machines with a varying number of processors and input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). Figures 2.38 and 2.40 present the execution time of procedure `BSP_rnd_sort` on the two machines (using as the underlying sequential sorting algorithm merge sort) – figure 2.37 reflects the execution time (%) of procedure `BSP_rnd_sort` with respect to its constituent phases. Similarly, figures 2.39 and 2.41 expose the efficiency of the corresponding timing results.

As shown in figures 2.38 and 2.40, the execution time of procedure `BSP_rnd_sort` using a fixed number of processors scales with the input size (approximately as $n \cdot \lg n$). Accordingly, the execution time of procedure `BSP_rnd_sort` for a fixed input size scales inversely with the number of processors (approximately as $1/p$). Both experimental results are in accordance with the theoretical analysis of section 2.6.7. In terms of efficiency, procedure `BSP_rnd_sort` utilizes approximately (within a 5%):

- 80%–90% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 95%–100% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),
- 85%–95% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 95%–100% of the available processor resources on the Cray T3D machine (for large problem sizes).

¹³The terms generic and data-type independent imply that procedure `BSP_rnd_sort` can be employed to handle *any* data type. To this end, a comparison function `cmp` is used to facilitate the comparison of two elements e_1 and e_2 . In particular, the `cmp` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2 .

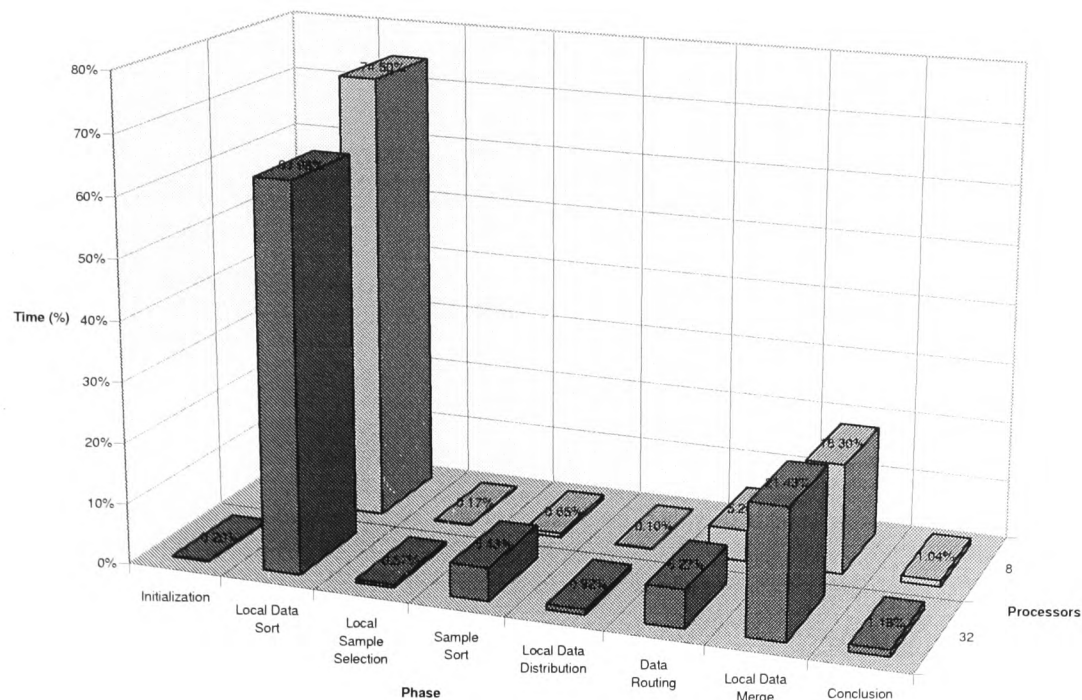


Figure 2.37: Execution time (% per phase) on a Cray T3D for randomized sorting – uniform distribution (U), input size 2^{21} .

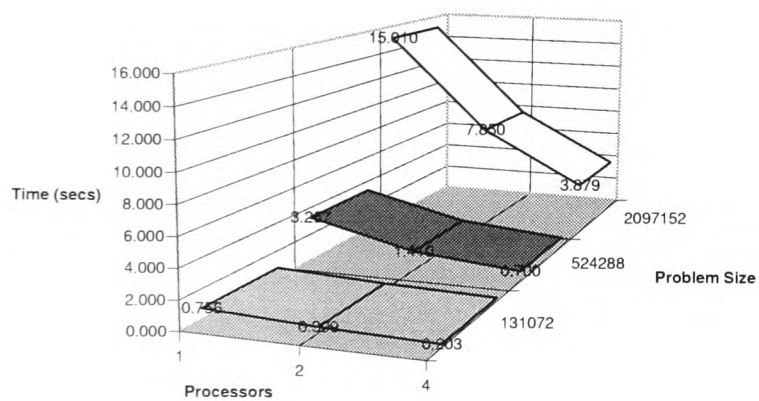


Figure 2.38: Execution time on a SGI Power Challenge for randomized sorting – uniform distribution (U).

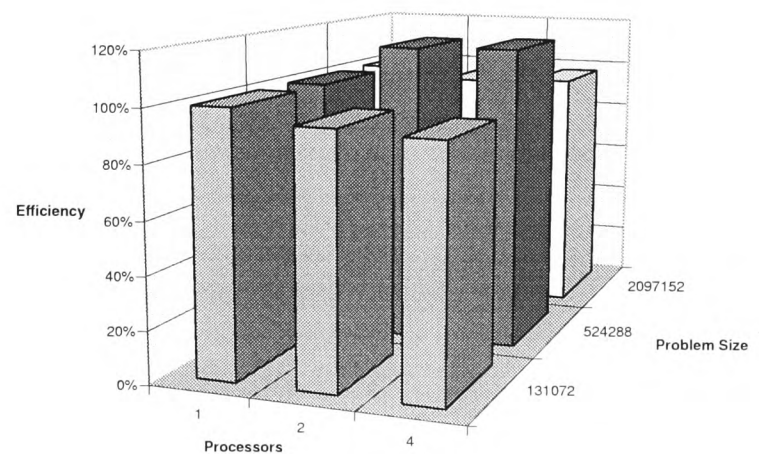


Figure 2.39: Efficiency on a SGI Power Challenge for randomized sorting – uniform distribution (U).

In order to test the obliviousness of the sorting algorithm to the input data distribution we benchmarked our implementation (`BSP_rnd_sort`) on a machine with a varying number of processors and input sets (consisting of 2^{21} integer valued items – unless otherwise indicated). Figure 2.42 presents the execution time of procedure `BSP_rnd_sort`. Similarly, figure 2.43 exposes the efficiency of the corresponding timing result.

Obviously, the performance of the algorithm is quite consistent across the different input data distributions with the exception of the deterministic-duplicates distribution (we note that the observable inconsistency in procedure `BSP_rnd_sort`, with respect to the deterministic-duplicates distribution, derives from the sequential merging and sorting implementations (`merge` and `sort` respectively) employed in the main body of `BSP_rnd_sort` and is irrelevant to the otherwise structure of the procedure).

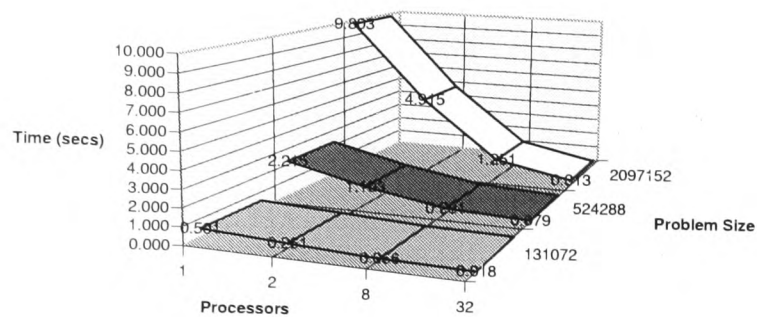


Figure 2.40: Execution time on a Cray T3D for randomized sorting – uniform distribution (U).

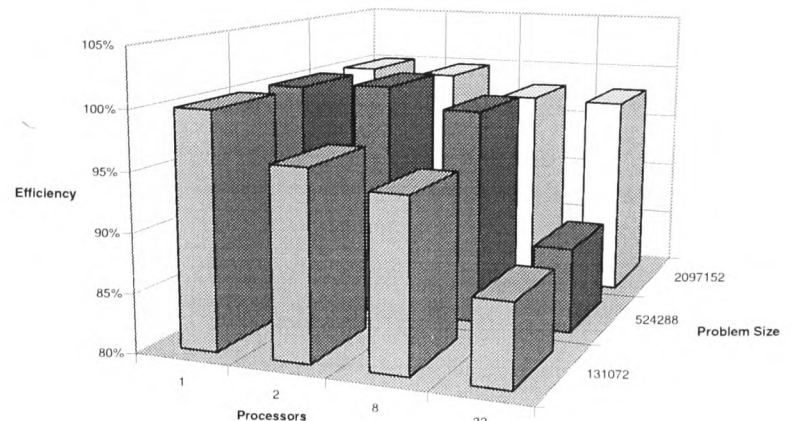


Figure 2.41: Efficiency on a Cray T3D for randomized sorting – uniform distribution (U).

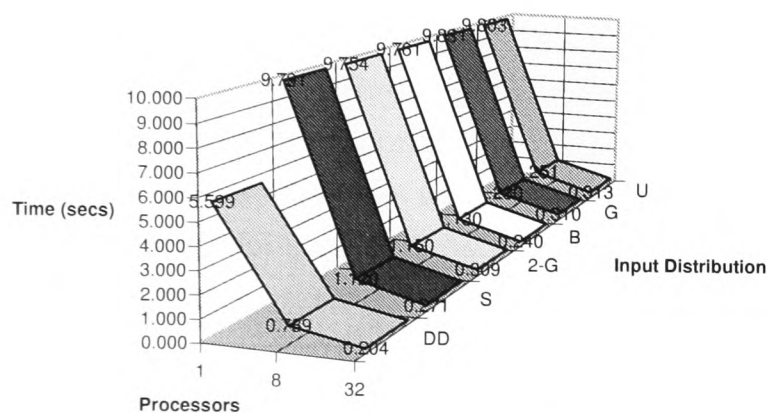


Figure 2.42: Execution time on a Cray T3D for randomized sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

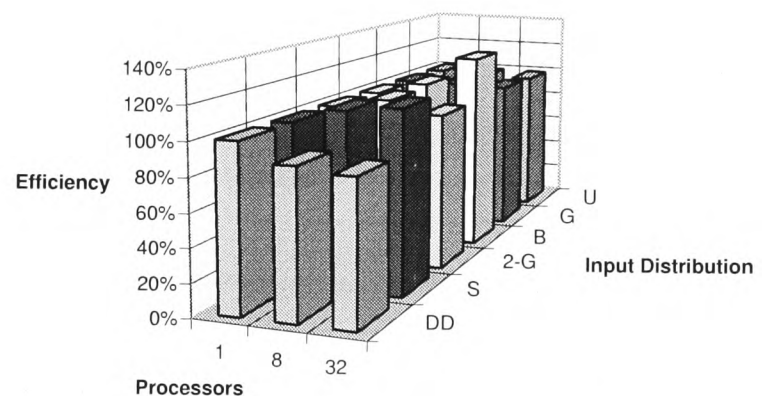


Figure 2.43: Efficiency on a Cray T3D for randomized sorting on uniform distribution (U), Gaussian distribution (G), bucket-sorted distribution (B), 2-group distribution (2-G), staggered distribution (S) and deterministic-duplicates distribution (DD) – input size 2^{21} .

2.6.8 A Fully Scaleable Optimal Sorting Algorithm

In this section, we provide an optimal BSP solution to the problem of sorting a sequence X that is based on a pipelined divide-and-conquer strategy similar to that of the merge-sort algorithm of [47] and achieves optimality for the full range of values of n and p . The underlying algorithm is initially presented and analyzed for the CREW-BSP model, and subsequently, extended to the BCAST-BSP and standard BSP models. Our main results are summarized in theorems 2.6.6 and 2.6.7 of sub-sections 2.6.8.1 and 2.6.8.2 respectively.

2.6.8.1 The CREW-BSP and BCAST-BSP Variants

In this section we present a deterministic CREW-BSP algorithm for the problem of sorting n items; moreover, we show how the algorithm can be adapted to run on the BCAST-BSP model.

Informal Description of Algorithm The optimal deterministic sorting algorithm is a synthetic generalization of the CREW PRAM sorting algorithm of [47] – with the noted difference that the underlying binary tree structure of the algorithm of [47] is non-trivially substituted by a d -ary tree structure (for some appropriate d) in order to improve communication and synchronization costs.

Our main result is summarized in the following theorem.

Theorem 2.6.6 *There exists a p -processor CREW-BSP (BCAST-BSP) algorithm for sorting a multi-set of size n that for all integers n and p such that $p = O(n)$, requires time $O((n/p) \cdot \lg n)$ and $L \cdot O(\lg n / \lg(n/p)) + g \cdot O((n/p) \cdot (\lg n / \lg(n/p)))$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.6.6 follows easily as a consequence of these results.

In the remainder of this section we address a slightly more general problem than sorting; its solution provides with a time optimal CREW-BSP sorting algorithm. The general formulation of our algorithm is as follows. Let T be a complete balanced k -ary tree such that each leaf node contains a single input item drawn from a linearly ordered set. For simplicity, suppose that n is a power of $k = \max\{2, \lceil \sqrt{n/p} \rceil\}$, and that all the items are distinct. Both assumptions can be removed without affecting the asymptotic time analysis of the algorithm. Let v be an internal node of T . We consider the problem of determining, for each internal node v , the non-decreasing sequence $S[v]$ that contains all the items initially stored at the leaves of the sub-tree rooted at v . Hence, for the remainder of this section we assume that T is a k -ary tree such that at most one item is stored in each leaf node and every internal node has exactly k children. Our goal is to determine $S[\text{root}(T)]$.

Our k -way merge-sort algorithm employs a pipelined divide-and-conquer strategy similar to that of the merge-sort algorithm of [47]; that is, $S[v]$ is determined over a number of phases such that, at phase l , $U_l[v]$ (a non-decreasing subset of $S[v]$) is an approximation of $S[v]$ that is improved at the next phase $l+1$. Simultaneously, a sample of $U_l[v]$ is propagated upward for producing approximations of the sequences to be generated at higher levels.

We next describe the algorithm for generating $U_l[v]$; later, we provide a correctness proof and an analysis of the resources required. Initially, $U_0[v]$ is empty if v is an internal node of T , and contains exactly one item if v is a leaf node. We define $\text{alt}(v)$, called *altitude* of node v [119], to be the length of the path from v to any leaf node (recall that T is a complete balanced k -ary tree, and therefore, all the leaves are on the same level). The algorithm realizes a sequence of phases, at each phase l , constructs a new approximation $U_l[v]$ of $S[v]$ for each *active* node $v \in T$. We say that v is active during phase l if $\text{alt}(v) \leq l \leq 3 \cdot \text{alt}(v)$. A node ceases being active when $l > 3 \cdot \text{alt}(v)$; later, we show that if $l = 3 \cdot \text{alt}(v)$, then node v is *full*; that is, $U_l[v] = S[v]$.

An additional notion is required before proceeding to the details of the algorithm. Let X and Y be non-decreasing sequences and let c be a positive integer. X is a c -sample of Y , denoted by $\text{smp}_c(Y)$, if X consists of every c -th item of Y . Define $\text{smp}(U_l[v])$ for an arbitrary node $v \in T$ as follows:

$$\text{smp}(U_l[v]) = \begin{cases} \text{smp}_{k^2}(U_l[v]) & \text{if } l \leq 3 \cdot \text{alt}(v), \\ \text{smp}_k(U_l[v]) & \text{if } l = 3 \cdot \text{alt}(v) + 1, \\ \text{smp}_1(U_l[v]) & \text{if } l = 3 \cdot \text{alt}(v) + 2. \end{cases}$$

Informally, $\text{smp}(U_l[v])$ consists of every k^2 -th item of $U_l[v]$ until it grows full; in the following phase (that is phase $3 \cdot \text{alt}(v) + 1$) $\text{smp}(U_l[v])$ consists of every k -th item of $U_l[v]$, and every item in phase $3 \cdot \text{alt}(v) + 2$. At each phase l , for each active node $v \in T$, the algorithm consists of the following two stages.

Algorithm CREW-MERGE-SORT

begin

[**Stage 1**] Sequence $\text{smp}(U_l[v])$ is constructed.

[**Stage 2**] Let z_i , for all $1 \leq i \leq k$, be the children of v . Sequence $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$ is constructed, where $\cup$ denotes merging.

end

It is clear that three phases after node v becomes full, the parent of v , say u , also becomes full. We thus obtain the following result.

Lemma 2.6.1 *Let v be an arbitrary node of T . Then, at the end of phase $l = 3 \cdot \text{alt}(v)$ of algorithm CREW-MERGE-SORT, v becomes full.*

Proof The proof is by induction on $\text{alt}(v)$. It is straightforward to verify that the claim is true for all nodes satisfying $\text{alt}(v) = 0$, since they all are leaves satisfying initially $U_0[v] = S[v]$.

Let v be an arbitrary internal node in T with $\text{alt}(v) > 0$, and let z_i , for all $1 \leq i \leq k$, be the children of v . By the induction hypothesis the children of v become full at phase $l = 3 \cdot (\text{alt}(v) - 1)$, since $\text{alt}(z_i) = \text{alt}(v) - 1$. Recall that $U_{l+1}[v]$ is obtained by merging $\text{smp}(U_l[z_i])$, for all $1 \leq i \leq k$, and similarly, $U_{l+3}[v]$ is obtained by merging $\text{smp}(U_{l+2}[z_i])$, for all $1 \leq i \leq k$. By definition, $\text{smp}(U_{l+2}[z_i]) = \text{smp}_1(U_l[z_i])$, and thus, $U_{l+3}[v] = S[v]$. Hence, at phase $l + 3 = 3 \cdot \text{alt}(v)$, v becomes full, and the lemma follows. $\square$

We therefore conclude the following.

Corollary 2.6.2 *Algorithm CREW-MERGE-SORT requires $3 \cdot \lg n / \lg \lceil \sqrt{n/p} \rceil$ phases, for all integers n and p such that $n \geq 4 \cdot p$.*

Proof We can readily verify that the height of T is $\log_k n = \lg n / \lg \lceil \sqrt{n/p} \rceil$, for all integers n and p such that $n \geq 4 \cdot p$. Combining this fact with lemma 2.6.1, we obtain the desired result. $\square$

A few additional technical results are required before a detailed description of the merging procedure is given. We start by establishing that $\text{smp}(U_l[v])$ is a $2k$ -cover of $\text{smp}(U_{l+1}[v])$.

Lemma 2.6.2 *Let v be an arbitrary node of T and let $l \geq 0$. Then, $\text{smp}(U_l[v])$ is a $2k$ -cover of $\text{smp}(U_{l+1}[v])$.*

Proof We use induction on l to prove the following claim.

Claim 2.6.4 *If $[x, y]$ and $\{-\infty, \text{smp}(U_l[v]), +\infty\}$ have $r \geq 2$ items in common, then $[x, y]$ intersects $\text{smp}(U_{l+1}[v])$ in at most $r \cdot k$ items.*

Proof The claim is true initially, for when $l = 0$, $U_0[v]$ has no more than one item, and thus $\text{smp}(U_0[v])$ and $\text{smp}(U_1[v])$ are both empty. Assume that the induction hypothesis holds up to phase $l - 1$; we show that the claim holds for phase l .

Let $[x, y]$ be an interval with $x, y \in \{-\infty, \text{smp}(U_l[v]), +\infty\}$ such that the number of common items is r . Assume that $l < 3 \cdot \text{alt}(v)$. Since $\text{smp}(U_l[v])$ is a k^2 -sample of $U_l[v]$ we obtain that $[x, y]$ and $\{-\infty, U_l[v], +\infty\}$ have at most $(r - 1) \cdot k^2 + 1$ items in common.

Recall that $U_l[v]$ is obtained by merging $\text{smp}(U_{l-1}[z_i])$, for all $1 \leq i \leq k$, where the z_i 's are the children of v . Thus, the items belonging to $[x, y]$ and $U_l[v]$ come from $\cup_i \text{smp}(U_{l-1}[z_i])$, where each sequence is viewed as a set of items. Let $[x_i, y_i]$, for all $1 \leq i \leq k$, be the smallest interval containing $[x, y]$ such

that $x_i, y_i \in \{-\infty, \text{smp}(U_{l-1}[z_i]), +\infty\}$, and let a_i be the number of items in common between $[x_i, y_i]$ and $\{-\infty, \text{smp}(U_{l-1}[z_i]), +\infty\}$. Since the items are distinct, we obtain that $\sum_i a_i \leq (r-1) \cdot k^2 + 1 + 2 \cdot (k-1)$, since $2 \cdot (k-1)$ additional items of $\{x_1, y_1, x_2, y_2, \dots, x_k, y_k\}$ are included.

By the induction hypothesis, $[x_i, y_i]$ intersects $\text{smp}(U_l[z_i])$ in at most $k \cdot a_i$ items. By definition, $U_{l+1}[v]$ is the sequence obtained by merging $\text{smp}(U_l[z_i])$, for all $1 \leq i \leq k$, and thus, $[x, y]$ intersects $U_{l+1}[v]$ in at most $\sum_i k \cdot a_i \leq (r-1) \cdot k^3 + 2 \cdot k^2 - k$ items. Since by definition $\text{smp}(U_{l+1}[v])$ is a k^2 -sample of $U_{l+1}[v]$ and $k \geq 2$, we obtain that $[x, y]$ intersects $\text{smp}(U_{l+1}[v])$ in at most $r \cdot k$ items, and the claim follows for $l < 3 \cdot \text{alt}(v)$.

It remains to prove the claim for the case $l \geq 3 \cdot \text{alt}(v)$. This is readily seen. Consider phase $l' = 3 \cdot \text{alt}(v)$ (the first phase that $U_{l'}[v]$ is full). Then, $\text{smp}(U_{l'}[v])$ consists of every k^2 -th item of $U_{l'}[v]$ and $\text{smp}(U_{l'+1}[v])$ consists of every k -th item of $U_{l'+1}[v]$; thus, the claim is true for this phase. The argument is similar for subsequent phases. $\square$

We now complete the proof of the lemma by taking $r = 2$. Let $x, y \in \{-\infty, \text{smp}(U_l[v]), +\infty\}$, such that x and y are adjacent. It is straightforward to verify that $[x, y]$ intersects $\{-\infty, \text{smp}(U_l[v]), +\infty\}$ in exactly two items, and therefore, from claim 2.6.4 we obtain that $[x, y]$ and $\text{smp}(U_{l+1}[v])$ have at most $2 \cdot k$ items in common. Hence, $\text{smp}(U_l[v])$ is a $2k$ -cover of $\text{smp}(U_{l+1}[v])$, and the lemma follows. $\square$

Similarly, we derive the following result that relates $U_l[v]$ and $U_{l+1}[v]$.

Corollary 2.6.3 *If $[x, y]$ and $\{-\infty, U_l[v], +\infty\}$ have $r \geq 2$ items in common, then $[x, y]$ intersects $U_{l+1}[v]$ in at most $r \cdot k + 2 \cdot (k^2 - k)$ items.*

Proof Let $[x, y]$ be an interval with $x, y \in \{-\infty, \text{smp}(U_l[v]), +\infty\}$ such that the number of common items is r . Recall that $U_l[v]$ is obtained by merging $\text{smp}(U_{l-1}[z_i])$, for all $1 \leq i \leq k$. Therefore, the items belonging to $[x, y]$ and $U_l[v]$ come from $\cup_i \text{smp}(U_{l-1}[z_i])$, where each sequence is viewed as a set of items. Let $[x_i, y_i]$, for all $1 \leq i \leq k$, be the smallest interval containing $[x, y]$ such that $x_i, y_i \in \{-\infty, \text{smp}(U_{l-1}[z_i]), +\infty\}$, and let a_i be the number of items in common between $[x_i, y_i]$ and $\{-\infty, \text{smp}(U_{l-1}[z_i]), +\infty\}$. By claim 2.6.4, there are at most $k \cdot a_i$ items in $\text{smp}(U_{l+1}[z_i])$ between x_i and y_i . Since the items are distinct, we obtain that $\sum_i a_i \leq r + 2 \cdot (k-1)$, since $2 \cdot (k-1)$ additional items of $\{x_1, y_1, x_2, y_2, \dots, x_k, y_k\}$ are included. This result implies that there are at most $\sum_i k \cdot a_i \leq r \cdot k + 2 \cdot (k^2 - k)$ items in $U_{l+1}[v] = \cup_i \text{smp}(U_{l+1}[z_i])$, between x and y , and the desired result follows. $\square$

The next lemma shows how to efficiently merge k sequences and maintain certain rank information on the CREW-BSP (BCAST-BSP) model. We assume that the input sequences are initially evenly distributed but otherwise arbitrarily among the processors before the beginning of the execution of the algorithm.

Lemma 2.6.3 *Let $X_1, X_2, \dots, X_k$ be k non-decreasing sequences, and let Y be a sequence such that, if interval I and $\{-\infty, Y, +\infty\}$ have $r \geq 2$ items in common, then I intersects $\cup_i X_i$, for all $1 \leq i \leq k$, in at most $r \cdot k + 2 \cdot (k^2 - k)$ items. Moreover, assume that $\text{rank}(\text{smp}_k(Y) : X_i)$, for all $1 \leq i \leq k$, is known. Then, there exists a $\lceil |Y|/k \rceil$ -processor BSP algorithm for merging $X_1, X_2, \dots, X_k$ that requires time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively.*

Proof Let $X = \cup_i X_i$, for all $1 \leq i \leq k$. By definition, the interval I between $k+1$ adjacent items, say $y_{j'k}, y_{j'k+1}, \dots, y_{(j'+1)k}$, from Y contains at most $3 \cdot k^2 - k$ items from X . In order to compute X ,

it therefore suffices, for each such interval in Y , to determine the relative order of at the most $3 \cdot k^2 - k$ items of X it contains. To carry out this procedure we associate one processor with each such interval in the sequence Y – items $y_{j'k}, y_{j'k+1}, \dots, y_{(j'+1)k}$ from Y are stored in the local memory of processor $p_{j'}$. The merging proceeds in two steps.

Step 1 The at most $3 \cdot k^2 - k$ items contained in I are those straddled by y_{jk} and $y_{(j+1)k}$. The leftmost of these items in X_i , for all $1 \leq i \leq k$ is determined, using $\text{rank}(\text{smp}_k(Y) : X_i)$; the rightmost items are obtained similarly. In a single communication round the so identified items are fetched. This step requires time $O(k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively.

Step 2 For each such interval I in Y , the k sub-sequences obtained in the previous step are merged using k -way merge [133]. Since the total size of these sub-sequences is at most $3 \cdot k^2 - k$ this step requires $O(k^2 \cdot \lg k)$ time.

The lemma follows. $\square$

In order to achieve communication efficiency, in the forthcoming analysis the following load-balancing properties are maintained at each node v in T .

Property 1 Let S be a sub-sequence stored at node v . If S is full, then S is partitioned into contiguous blocks of size k^2 each, and each such block (along with its associated information) is assigned to a single processor.

Property 2 If S is not full, then S is partitioned into contiguous blocks of size k each, and each such block (along with its associated information) is assigned to a single processor.

In both cases, we assume that contiguous blocks of S are associated with consecutive processors. The following result bounds the space needed by sequences satisfying load-balancing property 2.

Lemma 2.6.4 *The total space required by sequences satisfying load-balancing property 2 is $O(n/k)$.*

Proof It is not difficult to verify that the total size of the sequences satisfying load-balancing property 1 is $O(n)$. For although each full sequence requires $O(n)$ space, the space can be reused from sequence to sequence. Let l be the level in T such that the sequences at level l are full. Each sequence at level l advances at most $1/k$ of its items to level $l-1$; similarly, each sequence at level $l-1$ advances at most $1/k^2$ of its items to level $l-1$ and so on. Hence, the total space required by sequences satisfying load-balancing property 2 is bounded above by $O((n/k) \cdot \sum_l 1/k^{2 \cdot l}) = O(n/k)$, and the lemma follows. $\square$

The procedure to compute $U_{l+1}[v]$, for an arbitrary active node v of T with children z_i , for all $1 \leq i \leq k$, is summarized in the following proposition.

Proposition 2.6.5 *Let l be a given phase of algorithm CREW-MERGE-SORT, and assume that node v of T is active. Then, there exists a p -processor CREW-BSP (BCAST-BSP) procedure for computing $U_{l+1}[v]$, that requires time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, where $k = \max \{2, \lceil \sqrt{n/p} \rceil\}$.*

Proof We assume that the items are distinct since we can always make them so by, for example, appending to them the code for their memory location. Additionally, in order to achieve computation and

communication efficiency, the algorithm requires at each node v in T the maintenance of the following sequences: $U_l[v]$, $\text{smp}(U_l[v])$, and $\text{smp}(U_{l-1}[v])$. To simplify the exposition, we also assume that the nodes of T and their associated sequences are stored in a structured pattern; that is, sequences are partitioned in such a way, that given the name of a sequence, say S , and an index i , the processor storing $L[i]$ can be determined as a local computation. The input sequences are initially evenly distributed among the processors and satisfy load-balancing property 2. Furthermore, when a new sequence is created, for each of its items, a record is maintained of where it comes from.

Let z_i , for all $1 \leq i \leq k$, be the children of an arbitrary node v of T and let $\text{alt}(v) \leq l < 3 \cdot \text{alt}(v)$. The following procedure shows how to efficiently compute $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$. As part of the input to node v , for phase l , we assume the following *rank invariants*.

- (1) $\text{rank}(\text{smp}(U_{l-1}[v]) : \text{smp}(U_l[v]))$.
- (2) $\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(U_{l-1}[z_j]))$, for all $1 \leq i, j \leq k$.

Notice that since $l < 3 \cdot \text{alt}(v)$ none of the above-mentioned sequences can possibly satisfy the conditions of load-balancing property 1.

Procedure CREW-MERGE-SORT

begin

[Stage 1] Sequence $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$, for all $1 \leq i \leq k$, is constructed.

[Analysis of Stage 1] Let $\text{smp}_k(U_l[v])$ be a k sample of $U_l[v]$. We first show how to compute $\text{rank}(\text{smp}_k(U_l[v]) : \text{smp}(U_l[z_i]))$, for all $1 \leq i \leq k$. We start with a few observations. Since $U_l[v]$ satisfies load-balancing property 2, each item of $\text{smp}_k(U_l[v])$ is associated with a single processor. Moreover, since $U_l[v] = \cup_i \text{smp}(U_{l-1}[z_i])$, we deduce that $\text{rank}(\text{smp}(U_{l-1}[z_i]) : U_l[v])$, for all $1 \leq i \leq k$, can be computed in time $O(k)$ and $L \cdot O(1) + g \cdot O(k)$, for computation and communication respectively, with $\lceil |U_l[v]|/k \rceil$ processors. Hence, since $\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(U_l[z_i]))$ is given, we immediately deduce that, for each $x \in \text{smp}_k(U_l[v])$ that originated from $\text{smp}(U_{l-1}[z_i])$, the rank of x in $\text{smp}(U_l[z_i])$ can be computed within the same time bound. Then, $\text{rank}(x : \text{smp}(U_l[z_j]))$, for all $1 \leq j \leq k$, is computed as follows. Let x_j and x'_j be the two items in $\text{smp}(U_{l-1}[z_j])$ that straddle x . Since by definition $\text{rank}(\text{smp}(U_{l-1}[z_j]) : \text{smp}(U_l[z_j]))$ is known, the boundaries of the interval I , between x_j and x'_j , can be easily determined from $\text{smp}(U_l[z_j])$. By lemma 2.6.2 we have that $\text{smp}(U_{l-1}[z_j])$ is a $2k$ -cover of $\text{smp}(U_l[z_j])$, and hence we conclude that $|I| \leq 2 \cdot k$. Thus, the rank of x in $\text{smp}(U_l[z_j])$, for all $1 \leq j \leq k$, can be determined in time $O(k \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, with $\lceil |U_l[v]|/k \rceil$ processors (by way of k distinct binary searches on non-decreasing sequences of size at most $2 \cdot k$). Finally, by employing lemma 2.6.3, with $X_i = \text{smp}(U_l[z_i])$ and $Y = U_l[v]$, we obtain $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$, in time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, with $\lceil |U_l[v]|/k \rceil$ processors.

Notice that this stage, as described, requires the CREW-BSP model. In the remainder of this analysis we sketch the details of a similar procedure that requires however the BCAST-BSP model. Consider $U_l[v]$ and let x be an arbitrary item in $\text{smp}_k(U_l[v])$. The goal is to compute for each such $x \in \text{smp}_k(U_l[v])$ the rank of x in $\text{smp}(U_l[z_j])$, for all $1 \leq j \leq k$. Consider x_j and x'_j , the two items in $\text{smp}(U_{l-1}[z_j])$ that straddle x . The problem is that between x_j and x'_j there are many items from $\text{smp}_k(U_l[v])$; these items, however, have to be adjacent in $\text{smp}_k(U_l[v])$. It is therefore sufficient for

the first such item to advance – as in the CREW-BSP variant of the procedure – and then broadcast all the necessary information from $U_l[z_j]$ to the appropriate items. Notice that, since the destination items are adjacent in $\text{smp}_k(U_l[v])$, the destination of the broadcast operation can be described in a constant number of words. This step requires $L \cdot O(1) + g \cdot O(k^2)$ communication time on the BCAST-BSP model. Having broadcast the necessary information to the appropriate processors the procedure proceeds as in the CREW-BSP version.

[Stage 2] If $l < 3 \cdot \text{alt}(v) - 1$, the necessary rank invariants for phase $l + 1$ are updated.

[Analysis of Stage 2] We first show how to obtain $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(U_{l+1}[v]))$ [giving invariant 1]. By corollary 2.6.3, the interval I between $k + 1$ adjacent items from $U_l[v]$, contains at most $3 \cdot k^2 - k$ items from $U_{l+1}[v]$. Moreover, since by way of stage 1 these intervals have already been identified and assigned to a single processor, the required ranks can be obtained by simply merging (locally) the corresponding portions of $U_l[v]$ and $U_{l+1}[v]$. This step requires $O(k^2)$ time with $\lceil |U_l[v]|/k \rceil$ processors. Having determined $\text{rank}(U_l[v] : U_{l+1}[v])$, the procedure can trivially compute $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(U_{l+1}[v]))$ within the same time bound.

We next consider the problem of determining $\text{rank}(\text{smp}(U_l[z_i]) : \text{smp}(U_l[z_j]))$, for all $1 \leq i, j \leq k$, [giving invariant 2]. Let $\text{smp}_k(U_l[v])$ be defined as above, and consider x and x' , two adjacent items in $\text{smp}_k(U_l[v])$. By corollary 2.6.3, the interval I between x and x' contains at most $3 \cdot k^2 - k$ items from $U_{l+1}[v]$. Moreover, the ranks of x and x' in $\text{smp}(U_l[z_i])$, for all $1 \leq i \leq k$, are known. Hence, since by way of stage 1, one processor is associated with each such interval, the rank of every k -th item from $U_{l+1}[v]$ in $\text{smp}(U_l[z_i])$ can be computed in $O(k^2)$ time. The $U_{l+1}[v]$ sequence can then be redistributed, in time $O(k^2)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, so as to maintain the load-balancing properties. Finally, since $l < 3 \cdot \text{alt}(v) - 1$, we have that $U_{l+1}[v]$ satisfies the conditions of load-balancing property 2, and consequently, each processor associated with a portion of $U_{l+1}[v]$ maintains at most k items. Hence, the remaining ranks can be computed in $O(k^2)$ time with $\lceil |U_{l+1}[v]|/k \rceil$ processors (notice that each processor receives at least one ranked item). The overall time complexity of this stage is $O(k^2)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively.

[Stage 3] The sequences stored at node v of T are redistributed according to load-balancing properties 1 and 2.

[Analysis of Stage 3] When this step is executed, every processor holds at most $O(k^2)$ items, hence, this stage requires time $O(k^2)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively.

end

Analysis The correctness of the algorithm follows from its description and is omitted. We can easily verify that the time complexity (over all stages) of procedure CREW-MERGE-SORT is $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively. To complete the proof it remains to establish that only p processors are required by the procedure. This however is an easy consequence of lemma 2.6.4.

The proposition follows trivially. $\square$

We have therefore established the following result.

Proposition 2.6.6 *There exists a p -processor CREW-BSP (BCAST-BSP) algorithm for sorting a multi-set of size n that for all integers n , p and k such that $k = \max \{2, \lceil \sqrt{n/p} \rceil\}$ and $p = O(n)$, requires time*

$$C_{srt}(n, p) = O(k^2 \cdot \lg n)$$

and

$$M_{srt}(n, p) = L \cdot O\left(\frac{\lg n}{\lg k}\right) + g \cdot O\left(k^2 \cdot \frac{\lg n}{\lg k}\right),$$

for computation and communication respectively.

Proof Let T' be the tree that is obtained from T by assigning to each leaf of T a single input item. By corollary 2.6.2, the height of T' is $O(\lg n / \lg k)$. Therefore, algorithm CREW-MERGE-SORT consists of $O(\lg n / \lg k)$ phases. Furthermore, by proposition 2.6.5 each such phase requires time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, and the claim follows. $\square$

Proof (of theorem 2.6.6) The theorem follows easily from proposition 2.6.6. $\square$

2.6.8.2 The BSP Variant

In this section we present a deterministic BSP algorithm for the problem of sorting n items.

Informal Description of Algorithm The optimal deterministic sorting algorithm is a synthetic generalization of the EREW PRAM sorting algorithm of [47] – with the noted difference that the underlying binary tree structure of the algorithm of [47] is non-trivially substituted by a d -ary tree structure (for some appropriate d) in order to improve communication and synchronization costs.

Our main result is summarized in the following theorem.

Theorem 2.6.7 *There exists a p -processor BSP algorithm for sorting a multi-set of size n that for all integers n and p such that $p = O(n)$, requires time $O((n/p) \cdot \lg n)$ and $L \cdot O(\lg n / \lg(n/p)) + g \cdot O((n/p) \cdot (\lg n / \lg(n/p)))$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.6.7 follows easily as a consequence of these results.

The algorithm uses the same tree T as for the CREW-BSP variant. In order however to avoid concurrent-read conflicts, at each node v in T , an additional sequence $D_l[v]$ is maintained, for each phase l . Simultaneously, a sample of $D_l[v]$ is propagated downward to facilitate the avoidance of concurrent-read conflicts at lower levels.

We next describe the algorithm for generating $U_l[v]$ and $D_l[v]$; later, we provide a correctness proof and an analysis of the resources required. Initially, $D_0[v]$ is empty, and $U_0[v]$ is empty if v is an internal node of T , and contains exactly one item if v is a leaf node. An additional definition is required before a description of the algorithm is given. Define $\text{smp}(D_l[v])$ for an arbitrary node $v \in T$ as follows:

$$\text{smp}(D_l[v]) = \text{smp}_{k^2}(D_l[v]).$$

Informally, $\text{smp}(D_l[v])$ consists of every k^2 -th item of $D_l[v]$. At each phase l , for each active node $v \in T$, the algorithm consists of the following 2 stages.

Algorithm MERGE-SORT

begin

[Stage 1] Sequences $\text{smp}(U_l[v])$ and $\text{smp}(D_l[v])$ are constructed.

[Stage 2] Let u , z_i , and w_j , for all $1 \leq i \leq k$ and $1 \leq j < k$, be, respectively, the parent, children and siblings of v . Sequence $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$ is constructed, and if $U_{l+1}[v]$ is not full, also sequence $D_{l+1}[v] = \cup_j \text{smp}(U_l[w_j]) \cup \text{smp}(D_l[u])$ is constructed, where $\cup$ denotes merging.

end

The following result relates the sizes of $D_l[v]$ and $U_l[v]$.

Lemma 2.6.5 *Let v be an arbitrary node of T and let $l < 3 \cdot \text{alt}(v)$. Then $|D_l[v]| \leq ((k-1) \cdot k^2 / (k^5 - 1)) \cdot |U_l[v]|$.*

Proof The proof is by induction on l . The base case $l = 0$ is trivial, since $D_0[v]$ is empty and $U_0[v]$ has one item. Assume that the induction hypothesis holds up to phase $l-1$; we show that the claim holds for phase l .

Let u , z_i , and w_j , for all $1 \leq i \leq k$ and $1 \leq j < k$, be, respectively, the parent, children and siblings of v . Recall that $D_l[v]$ is obtained by merging $\text{smp}(D_{l-1}[u])$ and $\text{smp}(U_{l-1}[w_j])$, for all $1 \leq j < k$. Hence,

$$\begin{aligned}
 |D_l[v]| &= |\cup_j \text{smp}(U_{l-1}[w_j])| + |\text{smp}(D_{l-1}[u])| \\
 &\leq (k-1) \cdot |\text{smp}(U_{l-1}[v])| + \frac{1}{k^2} \cdot |D_{l-1}[u]| \\
 &\leq \frac{k-1}{k^2} \cdot |U_{l-1}[v]| + \frac{k-1}{k^5-1} \cdot |U_{l-1}[u]| \\
 &\leq \frac{k-1}{k^3} \cdot |U_l[v]| + \frac{k-1}{k^5-1} \cdot |\cup_j \text{smp}(U_{l-2}[w_j]) \cup \text{smp}(U_{l-2}[v])| \\
 &\leq \frac{k-1}{k^3} \cdot |U_l[v]| + \frac{k \cdot (k-1)}{k^5-1} \cdot |\text{smp}(U_{l-2}[v])| \\
 &\leq \frac{k-1}{k^3} \cdot |U_l[v]| + \frac{k-1}{(k^5-1) \cdot k^3} \cdot |U_l[v]| \\
 &= \frac{(k-1) \cdot k^2}{k^5-1} \cdot |U_l[v]|.
 \end{aligned}$$

The lemma follows. $\square$

A few additional technical results are required before a detailed description of the merging procedure is given. We start by establishing that $\text{smp}(D_l[v])$ is a $2k$ -cover of $\text{smp}(D_{l+1}[v])$.

Lemma 2.6.6 *Let v be an arbitrary node of T and let $l \geq 0$. Then, $\text{smp}(D_l[v])$ is a $2k$ -cover of $\text{smp}(D_{l+1}[v])$.*

Proof We use induction on l to prove the following claim.

Claim 2.6.5 *If $[x, y]$ and $\{-\infty, \text{smp}(D_l[v]), +\infty\}$ have $r \geq 2$ items in common, then $[x, y]$ intersects $\text{smp}(D_{l+1}[v])$ in at most $r \cdot k$ items.*

Proof The claim is true initially, for when $l = 0$, $D_0[v]$ is empty, and therefore, $\text{smp}(D_0[v])$ and $\text{smp}(D_1[v])$ are both empty. Assume that the induction hypothesis holds up to phase $l - 1$; we show that the claim holds for phase l .

Let $[x, y]$ be an interval with $x, y \in \{-\infty, \text{smp}(D_l[v]), +\infty\}$ such that the number of common items is r . Assume that $l < 3 \cdot \text{alt}(v)$, for when $l \geq 3 \cdot \text{alt}(v)$, $D_l[v]$ does not exist. Since $\text{smp}(D_l[v])$ is a k^2 -sample of $D_l[v]$ we obtain that $[x, y]$ and $\{-\infty, D_l[v], +\infty\}$ have at most $(r - 1) \cdot k^2 + 1$ items in common.

Recall that $D_l[v]$ is obtained by merging $\text{smp}(D_{l-1}[u])$ and $\text{smp}(U_{l-1}[w_j])$, for all $1 \leq j < k$, where the w_j 's are the siblings of v and u is the parent of v . Thus, the items belonging to $[x, y]$ and $D_l[v]$ come from $\cup_j \text{smp}(U_{l-1}[w_j]) \cup \text{smp}(D_{l-1}[u])$, where each sequence is viewed as a set of items. Let $[x', y']$ be the smallest interval containing $[x, y]$ such that $x', y' \in \{-\infty, \text{smp}(D_{l-1}[u]), +\infty\}$, and let a' be the number of items in common between $[x', y']$ and $\{-\infty, \text{smp}(D_{l-1}[u]), +\infty\}$. Likewise, let $[x_j, y_j]$, for all $1 \leq j < k$, be the smallest interval containing $[x, y]$ such that $x_j, y_j \in \{-\infty, \text{smp}(U_{l-1}[w_j]), +\infty\}$, and let a_j be the number of items in common between $[x_j, y_j]$ and $\{-\infty, \text{smp}(U_{l-1}[w_j]), +\infty\}$. Since the items are distinct, we obtain that $a' + \sum_j a_j \leq (r - 1) \cdot k^2 + 1 + 2 \cdot (k - 1)$, since $2 \cdot (k - 1)$ additional items of $\{x', y', x_1, y_1, \dots, x_{k-1}, y_{k-1}\}$ are included.

By the induction hypothesis, $[x', y']$ intersects $\text{smp}(D_l[u])$ in at most $k \cdot a'$ items. Moreover, by claim 2.6.4, $[x_j, y_j]$ intersects $\text{smp}(U_l[w_j])$ in at most $k \cdot a_j$ items. By definition, $D_{l+1}[v]$ is the sequence obtained by merging $\text{smp}(D_l[u])$ and $\text{smp}(U_l[w_j])$, for all $1 \leq j < k$, and thus, $[x, y]$ intersects $D_{l+1}[v]$ in at most $k \cdot a' + \sum_i k \cdot a_i \leq (r - 1) \cdot k^3 + 2 \cdot k^2 - k$ items. Since by definition $\text{smp}(D_{l+1}[v])$ is a k^2 -sample of $D_{l+1}[v]$ and $k \geq 2$, we obtain that $[x, y]$ intersects $\text{smp}(D_{l+1}[v])$ in at most $r \cdot k$ items, and the claim follows. $\square$

We now complete the proof of the lemma by taking $r = 2$. Let $x, y \in \{-\infty, \text{smp}(D_l[v]), +\infty\}$, such that x and y are adjacent. It is easy to verify that $[x, y]$ intersects $\{-\infty, \text{smp}(D_l[v]), +\infty\}$ in exactly two items, and therefore, from claim 2.6.5 we obtain that $[x, y]$ and $\text{smp}(D_{l+1}[v])$ have at most $2 \cdot k$ items in common. Hence, $\text{smp}(D_l[v])$ is a $2k$ -cover of $\text{smp}(D_{l+1}[v])$, and the lemma follows. $\square$

The next lemma shows how to efficiently merge m sequences and maintain certain rank information on the BSP model. We assume that the input sequences are initially evenly but otherwise arbitrarily among the processors before the beginning of the execution of the algorithm.

Lemma 2.6.7 *Let $X_1, X_2, \dots, X_m$ be m non-decreasing sequences and let $\text{cvr}(X_i)$ be c_i -covers for X_i , $1 \leq i \leq m$. Moreover, assume that $\text{rank}(\text{cvr}(X_i) : \text{cvr}(X_j))$ and $\text{rank}(\text{cvr}(X_i) : X_j)$, for all $1 \leq i, j \leq k$, are known. Then, there exists a $\lceil |\cup_i \text{cvr}(X_i)|/d_1 \rceil$ -processor BSP algorithm for merging $X_1, X_2, \dots, X_m$ and determining $\text{rank}(\cup_i \text{cvr}(X_i) : \cup_i X_i)$, for all $1 \leq i \leq m$, that requires time $O(\max_i \{c_i\} \cdot (d_1 + m) \cdot \lg m + d_1 \cdot m)$ and $L \cdot O(1) + g \cdot O(\max_i \{c_i\} \cdot (d_1 + m) + d_1 \cdot m)$, for computation and communication respectively, where d_1 is any positive integer. Given $\cup_i X_i$, for all $1 \leq i \leq m$, there also exists a $\lceil |\cup_i X_i|/d_2 \rceil$ -processor BSP algorithm for determining $\text{rank}(X_i : X_j)$, for all $1 \leq i, j \leq m$, that requires time $O(\max_i \{c_i\} \cdot m \cdot (d_1 + m)/d_2 + d_2 \cdot m)$ and $L \cdot O(1) + g \cdot O(\max_i \{c_i\} \cdot (d_1 + m) + d_2)$, for computation and communication respectively, where d_2 is any positive integer.*

Proof We first state the following result.

Claim 2.6.6 Let $X_1, X_2, \dots, X_m$ be m non-decreasing sequences and let $\text{cvr}(X_i)$ be c_i -covers for X_i , $1 \leq i \leq m$. If $[x, y]$ and $\{-\infty, \cup_i \text{cvr}(X_i), +\infty\}$, $1 \leq i \leq m$, have $r \geq 2$ items in common, then $[x, y]$ intersects $\cup_i X_i$ in at most $\max_i \{c_i\} \cdot (r + m - 2) + m$ items.

Proof Let $X = \cup_i X_i$, $\text{cvr}(X) = \cup_i \text{cvr}(X_i)$, for all $1 \leq i \leq m$, and define $[x, y]$ to be an interval with $x, y \in \{-\infty, \text{cvr}(X), +\infty\}$, such that the number of common items is r . Similarly, define $[x_i, y_i]$, for all $1 \leq i \leq m$, to be the smallest interval containing $[x, y]$ such that $x_i, y_i \in \{-\infty, \text{cvr}(X_i), +\infty\}$, and let a_i be the number of items in common between $[x_i, y_i]$ and $\{-\infty, \text{cvr}(X), +\infty\}$.

It is not difficult to verify that $[x_i, y_i]$ intersects X_i in at most $c_i \cdot (a_i - 1) + 1$ items, which implies that $[x, y]$ and X have in common at most $\sum_i (c_i \cdot (a_i - 1) + 1) \leq \sum_i (\max_i \{c_i\} \cdot (a_i - 1) + 1)$ items. Since all the items are distinct, we have that $\sum_i a_i = r + 2 \cdot (m - 1)$, since $2 \cdot (m - 1)$ additional items of $\{x_1, y_1, x_2, y_2, \dots, x_m, y_m\}$ are included. Thus, $[x, y]$ intersects X in at most $\max_i \{c_i\} \cdot (r + m - 2) + m$ items, and the claim follows. $\square$

We now complete the proof of the lemma. Let $X = \cup_i X_i$ and $\text{cvr}(X) = \cup_i \text{cvr}(X_i)$, for all $1 \leq i \leq m$. We first show how to compute $\text{cvr}(X)$. Consider an arbitrary item $x \in \text{cvr}(X)$, we can easily verify that $\text{rank}(x : \text{cvr}(X)) = \sum_i \text{rank}(x : \text{cvr}(X_i))$. Therefore, the sequence $\text{cvr}(X)$ can be computed in time $O(d_1 \cdot m)$ and $L \cdot O(1) + g \cdot O(d_1 \cdot m)$, for computation and communication respectively, with $\lceil |\text{cvr}(X)|/d_1 \rceil$ processors. We next show how to compute X .

By definition, $\text{cvr}(X_i)$ is a c_i -cover of X_i , for all $1 \leq i \leq m$. Hence, by claim 2.6.6, the interval I between $d_1 + 1$ adjacent items, say $x_{j'd_1}, x_{j'd_1+1}, \dots, x_{(j'+1)d_1}$, from $\text{cvr}(X)$ contains at most $\max_i \{c_i\} \cdot (d_1 + m - 1) + m$ items from X . In order to compute X , it therefore suffices, for each such interval in $\text{cvr}(X)$, to determine the relative order of at the most $\max_i \{c_i\} \cdot (d_1 + m - 1) + m$ items of X it contains. To carry out this procedure we associate one processor with each such interval in the sequence $\text{cvr}(X)$ – items $x_{j'd_1}, x_{j'd_1+1}, \dots, x_{(j'+1)d_1}$ from $\text{cvr}(X)$ are stored in the local memory of processor $p_{j'}$. The merging proceeds in two steps.

Step 1 The at most $\max_i \{c_i\} \cdot (d_1 + m - 1) + m$ items contained in I are those straddled by $x_{j'd_1}$ and $x_{(j'+1)d_1}$. If $x_{j'd_1}$ is in $\text{cvr}(X_i)$, then the leftmost of these items in X_j , for all $1 \leq j \leq m$, is determined, using $\text{rank}(\text{cvr}(X_i) : X_j)$; the rightmost item is obtained similarly. In a single communication round the so identified items are fetched. This step requires $O(m)$ and $L \cdot O(1) + g \cdot O(\max_i \{c_i\} \cdot (d_1 + m))$, for computation and communication respectively.

Step 2 For each such interval I in $\text{cvr}(X)$ the m sub-sequences obtained in the previous step are merged using m -way merge [133]. Since the total size of all the sub-sequences is at most $\max_i \{c_i\} \cdot (d_1 + m - 1) + m$ this step requires $O(\max_i \{c_i\} \cdot (d_1 + m) \cdot \lg m)$ time.

We now consider the problem of determining $\text{rank}(\text{cvr}(X) : X)$. Let x be an item of $\text{cvr}(X)$. The rank of x in X_j , is known, and therefore, the rank of x in X is determined by simply adding its ranks in X_j , for all $1 \leq j \leq m$. This procedure can be easily verified to require time $O(d_1 \cdot m)$ and $L \cdot O(1) + g \cdot O(d_1 \cdot m)$, for computation and communication respectively, with $\lceil |\text{cvr}(X)|/d_1 \rceil$ processors.

We finally show how to compute $\text{rank}(X_i : X_j)$, for all $1 \leq i, j \leq m$. By claim 2.6.6, the interval I between d_1 adjacent items, say $x_{j'd_1}, x_{j'd_1+1}, \dots, x_{(j'+1)d_1}$ from $\text{cvr}(X)$, contains at most $\max_i \{c_i\} \cdot (d_1 + m - 1) + m$ items from X . Moreover, the rank of $x_{j'd_1}$ and $x_{(j'+1)d_1}$ in X_i , for all $1 \leq i \leq m$, is known. Therefore, since by way of the previous steps we associate one processor with each such interval, the rank of every d_2 -th item of X in X_i , for all $1 \leq i \leq m$, can be computed in $O(\max_i \{c_i\} \cdot m \cdot (d_1 + m)/d_2)$ time.

Then, X is partitioned into contiguous blocks of size d_2 each, and each such block (along with its associated information) is assigned to a single processor. This step takes $L \cdot O(1) + g \cdot O(\max_i \{c_i\} \cdot (d_1 + m) + d_2)$ communication time. Finally, since each processor is associated with a portion of X of size at most d_2 , we conclude that the remaining ranks can be computed in $O(d_2 \cdot m)$ time, with $\lceil |X|/d_2 \rceil$ processors (notice that each processor receives at least one ranked item). The lemma follows. $\square$

In order to achieve communication efficiency, in the forthcoming analysis we maintain the following load-balancing properties at each node v in T .

Property 1 Let S be a sequence stored at node v . If S is full, then S is partitioned into contiguous blocks of size k^2 each, and each such block (along with its associated information) is assigned to a single processor.

Property 2 If S is not full, and S consists of every k -th item of some full sequence, then S is partitioned into contiguous blocks of size k each, and each such block (along with its associated information) is assigned to a single processor.

Property 3 If S is not full, and S consists of less than every k -th item of some full sequence, then S is partitioned into contiguous blocks of unit size each, and each such block (along with its associated information) is assigned to a single processor (notice that S consists of at most every k^2 -th item of some full sequence).

In all three cases, we assume that contiguous blocks of S are associated with consecutive processors. The following result bounds the space needed by sequences satisfying properties 2 and 3.

Lemma 2.6.8 *The total space required by sequences satisfying load-balancing properties 2 and 3 is, respectively, $O(n/k)$ and $O(n/k^2)$.*

Proof This result derives from the analysis of lemmas 2.6.4 and 2.6.5. $\square$

The procedure to compute $U_{l+1}[v]$ and $D_{l+1}[v]$, for an arbitrary active node v of T , is summarized in the following proposition.

Proposition 2.6.7 *Let l be a given phase of algorithm MERGE-SORT, and assume that node v of T is active. Then, there exists a p -processor BSP procedure for computing $U_{l+1}[v]$ and $D_{l+1}[v]$, that requires time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, where $k = \max \{2, \lceil \sqrt{n/p} \rceil\}$.*

Proof We assume that the items are distinct since we can always make them so by, for example, appending to them the code for their memory location. Additionally, in order to achieve computation and communication efficiency, the procedure requires at each node v in T the maintenance of the following sequences: $U_l[v]$, $\text{smp}(U_l[v])$, $\text{smp}(U_{l-1}[v])$, $D_l[v]$, $\text{smp}(D_l[v])$, $\text{smp}(D_{l-1}[v])$. To simplify the exposition, we also assume that the nodes of T and their associated sequences are stored in a structured pattern; that is, sequences are partitioned in such a way, that given the name of a sequence, say S , and an index i , the processor storing $L[i]$ can be determined as a local computation. The input sequences are initially evenly distributed among the processors and satisfy load-balancing properties 2 and 3. Furthermore, when a new sequence is created, for each of its items, a record is maintained of where it comes from.

Let u , z_i , and w_j , for all $1 \leq i \leq k$ and $1 \leq j < k$, be, respectively, the parent, children and siblings of an arbitrary node $v \in T$ and let $\text{alt}(v) \leq l < 3 \cdot \text{alt}(v)$. The following procedure shows how to efficiently compute $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$. As part of the input to node v , for phase l , we are given the following *rank invariants*.

- (1) $\text{rank}(\text{smp}(U_{l-1}[v]) : \text{smp}(U_l[v]))$.
- (2) $\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(U_{l-1}[z_j]))$, for all $1 \leq i, j \leq k$.
- (3) $\text{rank}(\text{smp}(U_{l-1}[v]) : \text{smp}(D_{l-1}[u]))$ and $\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(U_{l-1}[v]))$.
- (4) $\text{rank}(\text{smp}(D_{l-1}[v]) : \text{smp}(D_l[v]))$.
- (5) $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_l[v]))$ and $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(U_l[v]))$.
- (6) $\text{rank}(U_l[v] : \text{smp}(D_l[v]))$ and $\text{rank}(\text{smp}(D_l[v]) : U_l[v])$.
- (7) $\text{rank}(\text{smp}(U_l[v]) : D_l[v])$ and $\text{rank}(D_l[v] : \text{smp}(U_l[v]))$.

Notice that since $l < 3 \cdot \text{alt}(v)$, none of the above-mentioned sequences can possibly satisfy load-balancing property 1. Furthermore, since none of the items in $D[v]$ comes from the tree rooted at v , we have that $D[v]$ and $U[z_i]$, $1 \leq i \leq k$, have no items in common. Hence, all the merge and rank operations that are performed below satisfy the assumption that the items involved are distinct.

Initially, the variable d_1 (to be used later in the procedure) has the value k or 1, depending on whether the sequences involved in the procedure satisfy load-balancing property 2 or 3 respectively. Intuitively, the variable d_1 , for a given sequence S , indicates the number of items of S stored per processor.

Procedure MERGE-SORT

begin

[Stage 1] Rank-info $\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(U_l[v]))$ and $\text{rank}(\text{smp}(U_{l-1}[w_j]) : \text{smp}(U_l[v]))$, for all $1 \leq j < k$, is computed.

[Analysis of Stage 1] Recall that sequence $D_l[v]$ is obtained by merging $\cup_j \text{smp}(U_{l-1}[w_j])$ and $\text{smp}(D_{l-1}[u])$, for all $1 \leq j < k$. Hence, since $\text{rank}(D_l[v] : \text{smp}(U_l[v]))$ is given [from invariant 7], we deduce that $\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(U_l[v]))$ and $\text{rank}(\text{smp}(U_{l-1}[w_j]) : \text{smp}(U_l[v]))$, for all $1 \leq j < k$, can be computed in time $O(d_1)$ and $L \cdot O(1) + g \cdot O(d_1)$, for computation and communication respectively, with $\lceil |D_l[v]|/d_1 \rceil$ processors, where d_1 assumes the value k or 1 depending on whether the sequence $D_l[v]$ satisfies load-balancing property 2 or 3 respectively.

[Stage 2] Sequence $U_{l+1}[v]$ is constructed.

[Analysis of Stage 2] By lemma 2.6.2, we have that $\text{smp}(U_{l-1}[z_i])$, $1 \leq i \leq k$, is a $2k$ -cover for $\text{smp}(U_l[z_i])$. Additionally, the following information is available.

$\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(U_{l-1}[z_j]))$, for all $1 \leq i, j \leq k$, [from invariant 2].

$\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(U_l[z_i]))$, for all $1 \leq i \leq k$, [from invariant 1].

$\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(U_l[z_j]))$, for all $1 \leq i, j \leq k$, [from stage 1].

Hence, by employing lemma 2.6.7, with $X_i = \text{smp}(U_l[z_i])$ and $\text{cvr}(X_i) = \text{smp}(U_{l-1}[z_i])$, we obtain $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$, in time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, with $\lceil |U_l[v]|/d_1 \rceil$ processors, where d_1 assumes the value k or 1 depending on whether the sequence $U_l[v]$ satisfies load-balancing property 2 or 3 respectively.

[**Stage 3–9**] If $l < 3 \cdot \text{alt}(v) - 1$, in stages 3–9, the necessary rank invariants for phase $l + 1$ are updated.

[**Stage 3**] Rank-info $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(U_{l+1}[v]))$ [giving invariant 1] and $\text{rank}(\text{smp}(U_l[z_i]) : \text{smp}(U_l[z_j]))$, for all $1 \leq i, j \leq k$, [giving invariant 2], is computed.

[**Analysis of Stage 3**] It is not difficult to verify that the computation in stage 2, also yields $\text{rank}(U_l[v] : U_{l+1}[v])$ and $\text{rank}(\text{smp}(U_l[z_i]) : \text{smp}(U_l[z_j]))$, for all $1 \leq i, j \leq k$, in time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication, with $\lceil |U_l[v]|/d_1 \rceil$ and $\lceil |U_{l+1}[v]|/d_2 \rceil$ processors respectively, where d_1 is defined as in stage 2 and $d_2 = k$. Having determined $\text{rank}(U_l[v] : U_{l+1}[v])$, the procedure can trivially compute $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(U_{l+1}[v]))$ within the same time bound.

[**Stage 4**] Rank-info $\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(D_l[v]))$, for all $1 \leq i \leq k$, is computed.

[**Analysis of Stage 4**] Recall that $U_l[v]$ is obtained by merging $\text{smp}(U_{l-1}[z_i])$, for all $1 \leq i \leq k$. Thus, since $\text{rank}(U_l[v] : \text{smp}(D_l[v]))$ is given [from invariant 6], we conclude that $\text{rank}(\text{smp}(U_{l-1}[z_i]) : \text{smp}(D_l[v]))$, for all $1 \leq i \leq k$, can be computed in time $O(d_1)$ and $L \cdot O(1) + g \cdot O(d_1)$, for computation and communication respectively, with $\lceil |U_l[v]|/d_1 \rceil$ processors, where $d_1 = 1$.

[**Stage 5**] Rank-info $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_l[u]))$ and $\text{rank}(\text{smp}(D_l[u]) : \text{smp}(U_l[v]))$ [giving invariant 3], is computed.

[**Analysis of Stage 5**] By lemma 2.6.2, we have that $\text{smp}(U_{l-1}[v])$ is a $2k$ -cover for $\text{smp}(U_l[v])$, and by lemma 2.6.6, we have that $\text{smp}(D_{l-1}[u])$ is a $2k$ -cover for $\text{smp}(D_l[u])$. The following information is also available.

$\text{rank}(\text{smp}(U_{l-1}[v]) : \text{smp}(D_{l-1}[u]))$ and $\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(U_{l-1}[v]))$ [from invariant 3].

$\text{rank}(\text{smp}(U_{l-1}[v]) : \text{smp}(U_l[v]))$ [from invariant 1].

$\text{rank}(\text{smp}(U_{l-1}[v]) : \text{smp}(D_l[u]))$ [from stage 4].

$\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(U_l[v]))$ [from stage 1].

$\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(D_l[u]))$ [from invariant 4].

By employing lemma 2.6.7, with $X_1 = \text{smp}(U_l[v])$, $X_2 = \text{smp}(D_l[u])$, $\text{cvr}(X_1) = \text{smp}(U_{l-1}[v])$, and $\text{cvr}(X_2) = \text{smp}(D_{l-1}[u])$, we obtain $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_l[u]))$ and $\text{rank}(\text{smp}(D_l[u]) : \text{smp}(U_l[v]))$ in time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, with $\max \{ \lceil |\text{smp}(U_{l-1}[v]) \cup \text{smp}(D_{l-1}[u])|/d_1 \rceil, \lceil |\text{smp}(U_l[v]) \cup \text{smp}(D_l[u])|/d_2 \rceil \}$ processors, where $d_1 = 1$ and $d_2 = k$.

[**Stage 6**] Sequence $D_{l+1}[v]$ and rank-info $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(D_{l+1}[v]))$ [giving invariant 4], are computed.

[**Analysis of Stage 6**] Recall that in stage 3 at node u the procedure computed $\text{rank}(\text{smp}(U_l[w_i]) : \text{smp}(U_l[w_j]))$, for all $1 \leq i, j < k$, and in stage 5 at node w_j it also computed $\text{rank}(\text{smp}(U_l[w_j]) : \text{smp}(D_l[u]))$ and $\text{rank}(\text{smp}(D_l[u]) : \text{smp}(U_l[w_j]))$, for all $1 \leq j < k$. Hence, since $D_{l+1}[v] = \cup_j \text{smp}(U_l[w_j]) \cup \text{smp}(D_l[u])$, for all $1 \leq j < k$, the procedure can trivially compute $D_{l+1}[v]$ by simply adding, for each item x in $D_{l+1}[v]$, its ranks in $\text{smp}(D_l[u])$ and $\text{smp}(U_l[w_j])$, for all $1 \leq j < k$.

This step requires time $O(d_1 \cdot k)$ and $L \cdot O(1) + g \cdot O(d_1 \cdot k)$, for computation and communication respectively, with $\lceil |D_{l+1}[v]|/d_1 \rceil$ processors, where d_1 assumes the value k or 1 depending on whether the sequence $D_{l+1}[v]$ satisfies load-balancing property 2 or 3 respectively.

We now consider the problem of determining $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(D_{l+1}[v]))$. Recall that $D_l[v]$ is obtained by merging $\cup_j \text{smp}(U_{l-1}[w_j])$, for all $1 \leq j < k$, and $\text{smp}(D_{l-1}[u])$. Let x be an item in $D_l[v]$. The rank of x in $\text{smp}(D_l[u])$ and $\text{smp}(U_l[w_j])$, $1 \leq j < k$, is known; these ranks are obtained from $\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(D_l[u]))$ [from invariant 4], $\text{rank}(\text{smp}(D_{l-1}[u]) : \text{smp}(U_l[w_j]))$ [from stage 1], $\text{rank}(\text{smp}(U_{l-1}[w_j]) : \text{smp}(D_l[u]))$ [from stage 4], and $\text{rank}(\text{smp}(U_{l-1}[w_i]) : \text{smp}(U_l[w_j]))$ [from invariant 1 and stage 1], $1 \leq i, j < k$. Thus, the rank of x in $\text{smp}(D_{l+1}[u]) = \cup_j \text{smp}(U_l[w_j]) \cup \text{smp}(D_l[u])$ can be determined by simply adding its ranks in $\text{smp}(D_l[u])$ and $\text{smp}(U_l[w_j])$, for all $1 \leq j < k$. This step requires time $O(d_1 \cdot k)$ and $L \cdot O(1) + g \cdot O(d_1 \cdot k)$, for computation and communication respectively, with $\lceil |D_l[v]|/d_1 \rceil$ processors, where $d_1 = 1$. Having determined $\text{rank}(D_l[v] : \text{smp}(D_{l+1}[v]))$, the procedure can trivially obtain $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(D_{l+1}[v]))$ within the same time bound.

[Stage 7] Rank-info $\text{rank}(\text{smp}(U_{l+1}[v]) : \text{smp}(D_{l+1}[v]))$ and $\text{rank}(\text{smp}(D_{l+1}[v]) : \text{smp}(U_{l+1}[v]))$ [giving invariant 5], is computed.

[Analysis of Stage 7] We first consider the problem of determining rank-info $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(U_{l+1}[v]))$. Recall that in stage 5 at node z_i the procedure computed $\text{rank}(\text{smp}(U_l[z_i]) : \text{smp}(D_l[v]))$ and $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(U_l[z_i]))$, for all $1 \leq i \leq k$. Since $U_{l+1}[v] = \cup_i \text{smp}(U_l[z_i])$, for all $1 \leq i \leq k$, the procedure can therefore trivially compute $\text{rank}(U_{l+1}[v] : \text{smp}(D_l[v]))$ and $\text{rank}(\text{smp}(D_l[v]) : U_{l+1}[v])$, by simply adding, for each item x in $D_l[v]$, its ranks in $\text{smp}(U_l[z_i])$, for all $1 \leq i \leq k$. This step requires time $O(d_1 \cdot k)$ and $L \cdot O(1) + g \cdot O(d_1 \cdot k)$, for computation and communication respectively, with $\lceil |U_{l+1}[v]|/d_1 \rceil$ processors, where d_1 assumes the value k or 1 depending on whether the sequence $U_{l+1}[v]$ satisfies load-balancing property 2 or 3 respectively. Having determined $\text{rank}(U_{l+1}[v] : \text{smp}(D_l[v]))$ and $\text{rank}(\text{smp}(D_l[v]) : U_{l+1}[v])$, the procedure can trivially obtain $\text{rank}(\text{smp}(U_{l+1}[v]) : \text{smp}(D_l[v]))$ and $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(U_{l+1}[v]))$, within the same time bound.

We now consider the problem of computing rank-info $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_{l+1}[v]))$. Recall that in stage 3 at node u the procedure computed $\text{rank}(\text{smp}(U_l[w_j]) : \text{smp}(U_l[v]))$ and $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(U_l[w_j]))$, for all $1 \leq j < k$. Moreover, in stage 5 the procedure computed $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_l[u]))$ and $\text{rank}(\text{smp}(D_l[u]) : \text{smp}(U_l[v]))$. Hence, since $D_{l+1}[v] = \cup_j \text{smp}(U_l[w_j]) \cup \text{smp}(D_l[u])$, for all $1 \leq j < k$, the procedure can deduce $\text{rank}(\text{smp}(U_l[v]) : D_{l+1}[v])$ and $\text{rank}(D_{l+1}[v] : \text{smp}(U_l[v]))$, by simply adding, for each item x in $D_{l+1}[v]$, its ranks in $\text{smp}(D_l[u])$ and $\text{smp}(U_l[w_j])$, for all $1 \leq j < k$. This step requires time $O(d_1 \cdot k)$ and $L \cdot O(1) + g \cdot O(d_1 \cdot k)$, for computation and communication respectively, with $\lceil |D_{l+1}[v]|/d_1 \rceil$ processors, where d_1 assumes the value k or 1 depending on whether the sequence $D_{l+1}[v]$ satisfies load-balancing property 2 or 3 respectively. Having determined $\text{rank}(\text{smp}(U_l[v]) : D_{l+1}[v])$ and $\text{rank}(D_{l+1}[v] : \text{smp}(U_l[v]))$, the procedure can trivially obtain $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_{l+1}[v]))$ and $\text{rank}(\text{smp}(D_{l+1}[v]) : \text{smp}(U_l[v]))$, within the same time bound.

By lemma 2.6.2, we have that $\text{smp}(U_l[v])$ is a $2k$ -cover for $\text{smp}(U_{l+1}[v])$, and similarly, by lemma 2.6.6, we have that $\text{smp}(D_l[v])$ is a $2k$ -cover for $\text{smp}(D_{l+1}[v])$. Additionally, the following information is available.

$\text{rank}(\text{smp}(D_l[v]) : \text{smp}(U_l[v]))$ and $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_l[v]))$ [from invariant 5].

$\text{rank}(\text{smp}(D_l[v]) : \text{smp}(D_{l+1}[v]))$ [from stage 6].

$\text{rank}(\text{smp}(U_l[v]) : \text{smp}(U_{l+1}[v]))$ [from stage 3].

Since rank-info $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(U_{l+1}[v]))$ and $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_{l+1}[v]))$ is also available, by lemma 2.6.7, with $X_1 = \text{smp}(D_{l+1}[v])$, $X_2 = \text{smp}(U_{l+1}[v])$, $\text{cvr}(X_1) = \text{smp}(D_l[v])$ and $\text{cvr}(X_2) = \text{smp}(U_l[v])$, we obtain $\text{rank}(\text{smp}(D_{l+1}[v]) : \text{smp}(U_{l+1}[v]))$ and $\text{rank}(\text{smp}(U_{l+1}[v]) : \text{smp}(D_{l+1}[v]))$, in time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, with $\max \{ \lceil |\text{smp}(D_l[v]) \cup \text{smp}(U_l[v])|/d_1 \rceil, \lceil |\text{smp}(U_{l+1}[v]) \cup \text{smp}(D_{l+1}[v])|/d_2 \rceil \}$ processors, where $d_1 = 1$ and $d_2 = k$.

[Stage 8] Rank-info $\text{rank}(U_{l+1}[v] : \text{smp}(D_{l+1}[v]))$ and $\text{rank}(\text{smp}(D_{l+1}[v]) : U_{l+1}[v])$ [giving invariant 6], is computed.

[Analysis of Stage 8] By definition $\text{smp}(U_{l+1}[v])$ consists of every k^2 -th item of $U_{l+1}[v]$, and thus we have that $\text{smp}(U_{l+1}[v])$ is a k^2 -cover of $U_{l+1}[v]$. By lemma 2.6.6, we also have that $\text{smp}(D_l[v])$ is a $2k$ -cover for $\text{smp}(D_{l+1}[v])$. Furthermore, in the analysis of stage 7, rank-info $\text{rank}(\text{smp}(D_l[v]) : U_{l+1}[v])$, $\text{rank}(\text{smp}(U_{l+1}[v]) : \text{smp}(D_l[v]))$ and $\text{rank}(\text{smp}(D_l[v]) : \text{smp}(U_{l+1}[v]))$ is computed. The following information is also available.

$\text{rank}(\text{smp}(U_{l+1}[v]) : U_{l+1}[v])$.

$\text{rank}(\text{smp}(D_l[v]) : \text{smp}(D_{l+1}[v]))$ [from stage 6].

$\text{rank}(\text{smp}(U_{l+1}[v]) : \text{smp}(D_{l+1}[v]))$ [from stage 7].

Thus, by employing lemma 2.6.7, with $X_1 = U_{l+1}[v]$, $X_2 = \text{smp}(D_{l+1}[v])$, $\text{cvr}(X_1) = \text{smp}(U_{l+1}[v])$ and $\text{cvr}(X_2) = \text{smp}(D_l[v])$, we obtain $\text{rank}(U_{l+1}[v] : \text{smp}(D_{l+1}[v]))$ and $\text{rank}(\text{smp}(D_{l+1}[v]) : U_{l+1}[v])$, in time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, with $\max \{ \lceil |\text{smp}(U_{l+1}[v]) \cup \text{smp}(D_l[v])|/d_1 \rceil, \lceil |U_{l+1}[v] \cup \text{smp}(D_{l+1}[v])|/d_2 \rceil \}$ processors, where $d_1 = 1$ and $d_2 = k$.

[Stage 9] Rank-info $\text{rank}(D_{l+1}[v] : \text{smp}(U_{l+1}[v]))$ and $\text{rank}(\text{smp}(U_{l+1}[v]) : D_{l+1}[v])$ [giving invariant 7], is computed.

[Analysis of Stage 9] By definition $\text{smp}(D_{l+1}[v])$ consists of every k^2 -th item of $D_{l+1}[v]$, and thus we have that $\text{smp}(D_{l+1}[v])$ is a k^2 -cover of $D_{l+1}[v]$. By lemma 2.6.2, we also have that $\text{smp}(U_l[v])$ is a $2k$ -cover for $\text{smp}(U_{l+1}[v])$. Moreover, in (the analysis of) stage 7 the procedure computed $\text{rank}(\text{smp}(U_l[v]) : D_{l+1}[v])$, $\text{rank}(\text{smp}(D_{l+1}[v]) : \text{smp}(U_l[v]))$ and $\text{rank}(\text{smp}(U_l[v]) : \text{smp}(D_{l+1}[v]))$. The following information is also available.

$\text{rank}(\text{smp}(D_{l+1}[v]) : D_{l+1}[v])$.

$\text{rank}(\text{smp}(U_l[v]) : \text{smp}(U_{l+1}[v]))$ [from stage 3].

$\text{rank}(\text{smp}(D_{l+1}[v]) : \text{smp}(U_{l+1}[v]))$ [from stage 7].

Hence, by employing lemma 2.6.7, with $X_1 = D_{l+1}[v]$, $X_2 = \text{smp}(U_l[v])$, $\text{cvr}(X_1) = \text{smp}(D_{l+1}[v])$ and $\text{cvr}(X_2) = \text{smp}(U_l[v])$, we obtain $\text{rank}(D_{l+1}[v] : \text{smp}(U_{l+1}[v]))$ and $\text{rank}(\text{smp}(U_{l+1}[v]) : D_{l+1}[v])$, in time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, with $\max \{ \lceil |\text{smp}(D_{l+1}[v]) \cup \text{smp}(U_l[v])|/d_1 \rceil, \lceil |\text{smp}(U_{l+1}[v]) \cup D_{l+1}[v]|/d_2 \rceil \}$ processors, where $d_1 = 1$ and $d_2 = k$.

[Stage 10] The sequences stored at node v of T are redistributed according to load-balancing properties 1–3.

[Analysis of Stage 10] When this step is executed, every processor holds at most $O(k^2)$ items, hence, this stage requires time $O(k^2)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively.

end

Remark In the above procedure (stages 3–9), a sequence S stored at some arbitrary node v might be accessed (shared) by at most k nodes. Since however by load-balancing properties 2 and 3, each processor stores a segment of S of size at most k , no processor will receive more than $O(k^2)$ requests for S , and therefore, the overall time complexity of the procedure is not affected.

Analysis The correctness of the algorithm follows from its description and is omitted. We can easily verify that the time complexity (over all stages) of procedure MERGE-SORT is $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively. To complete the proof it remains to establish that only p processors are required by the procedure. This however is an easy consequence of lemma 2.6.8.

The proposition follows trivially. $\square$

We have therefore established the following result.

Proposition 2.6.8 *There exists a p -processor BSP algorithm for sorting a multi-set of size n that for all integers n , p and k such that $k = \max\{2, \lceil \sqrt{n/p} \rceil\}$ and $p = O(n)$, requires time*

$$C_{srt}(n, p) = O(k^2 \cdot \lg n)$$

and

$$M_{srt}(n, p) = L \cdot O\left(\frac{\lg n}{\lg k}\right) + g \cdot O\left(k^2 \cdot \frac{\lg n}{\lg k}\right),$$

for computation and communication respectively.

Proof Let T' be the tree that is obtained from T by assigning to each leaf of T a single input item. By corollary 2.6.2, the height of T' is $O(\lg n / \lg k)$. Therefore, algorithm MERGE-SORT consists of $O(\lg n / \lg k)$ phases. Furthermore, by proposition 2.6.7 each such phase requires time $O(k^2 \cdot \lg k)$ and $L \cdot O(1) + g \cdot O(k^2)$, for computation and communication respectively, and the claim follows. $\square$

Proof (of theorem 2.6.7) The theorem follows easily from proposition 2.6.8. $\square$

2.6.8.3 A Lower Bound

In the previous sections, we derived $O((n/p) \cdot \lg n)$ and $L \cdot O(\lg n / \lg(n/p)) + g \cdot O((n/p) \cdot (\lg n / \lg(n/p)))$ computation and communication time BSP algorithms for the problem of sorting n items. The natural question is whether this time bound can be improved. We conclude our discussion with an indication that $\Omega(\lg n / \lg(h+1))$ communication rounds is the best that can be achieved, regardless of the number

of processors, where $h \geq 1$ is the maximum number of items that can be transmitted or received by any processor in a single communication round.¹⁴ We also show that sorting n items, by storing only $O(n/p)$ items per processor, requires the communication of $\Omega((n/p) \cdot (\lg n / \lg(n/p)))$ items on a BSP machine with any number of processors $p \leq n$. Finally, we prove that by combining these two lower bounds we obtain that sorting requires $L \cdot \Omega(\lg n / \lg(n/p)) + g \cdot \Omega((n/p) \cdot (\lg n / \lg(n/p)))$ communication time – thus substantiating the claim that our methods are indeed optimal and fully scaleable.

Preliminaries and Notation Lower-bound arguments require a formalized framework in which precise statements about the progress of a computation can be made. In the next few paragraphs we establish – along the lines of [53] – such a framework. We first provide the following precise definition of a BSP computation. A BSP computation consists of a set $\{p_0, p_1, \dots, p_{p-1}\}$ of p processors, a set $\{m_1, m_2, \dots, m_m\}$ of m memory locations, an alphabet Σ , a number of n inputs, and an execution time T . To be consistent with the definitions of section 1.2, we assume that p and m are finite integers; this assumption, however, can be removed without affecting the results of this section. Computation proceeds in a succession of supersteps; a superstep may be conceived as a segment of a computation during which each processor performs a given computation thread using data already available there locally before the start of the superstep. The thread may include: computation operations on values held locally at the start of the superstep, message transmissions and message receipts. At the end of each superstep, all data exchanges are assumed to have completed before proceeding to the next superstep. To simplify the exposition we assume that remote data exchanges are effected during the superstep that they occur, however, the contents of the memory locations associated with such remote data exchanges are assumed to be available for access in the next superstep. Moreover, message transmissions and receipts are assumed to be realized as h -relations. That is, the maximum number of messages communicated to or from any processor is at most h , where each message may accommodate the contents of one memory location (word).

Each processor p_i , $1 \leq i < p$, consists of a (possibly infinite) state set Q_i . At the beginning of superstep t , $0 \leq t \leq T$, processor p_i , $0 \leq i < p$, is in state $q_i^t \in Q_i$ and memory location m_j , $1 \leq j \leq m$, contains symbol $s_j^t \in \Sigma$. Processor p_i then assumes a new state q_i^{t+1} that depends on the contents of the memory locations it read at superstep t and its previous state q_i^t . Similarly, memory location m_j assumes a new content s_j^{t+1} that depends on the values written (or *not* written) into m_j at superstep $t+1$ and its previous content s_j^t . At superstep zero, processor p_i is assumed to be in the distinguished state q_i^0 , and memory location m_j contains the input $x_j \in \Sigma$. That is, $s_j^0 = x_j$, for all $1 \leq j \leq n$, and $s_j^0 = \varepsilon$, for $j > n$, where ε is some distinguished (possibly blank) member of Σ .

Indices Affecting a Processor or a Memory Location Let $f(x_1, x_2, \dots, x_n)$ be a Boolean function with n inputs. We shall employ the notation $I = x_1 x_2 \dots x_n$ to denote the input $x_1, x_2, \dots, x_n$, and $I(i)$ to denote the input I whose i -th component is complemented. An input I is *critical* for f if and only if $f(I) \neq f(I(i))$, for all $1 \leq i \leq n$. Consider the computation of the Boolean function $f(x_1, x_2, \dots, x_n)$ on a p -processor BSP machine, where p may be arbitrarily large. Assume that the input values of $x_1, x_2, \dots, x_n$ are stored initially in memory locations $m_1, m_2, \dots, m_n$ with memory locations $m_{ih+1}, \dots, m_{(i+1)h}$ stored in the local memory of processor p_i , for all $1 \leq i \leq \lceil n/h \rceil$. At the end of the computation, the value of the function f is to be reflected in the state of processor p_0 .

We say that an input index i *affects the contents of a memory location* l at superstep t on some input

¹⁴In the remainder of this section we use the terms “superstep” and “communication round” interchangeably.

I if the contents of l at superstep t with input I differ from the contents of l at superstep t with input $I(i)$. Define the set $L(l, t, i)$ as follows.

$$L(l, t, I) = \{i \mid i \text{ affects } l \text{ at superstep } t \text{ on input } I\}.$$

Similarly, we say that an input index i *affects processor* q at superstep t on some input I if the state of q at superstep t with input I differs from the state of q at superstep t with input $I(i)$. The set $K(q, t, I)$ is defined in a way analogous to that of $L(l, t, I)$.

$$K(q, t, I) = \{i \mid i \text{ affects } q \text{ at superstep } t \text{ on input } I\}.$$

The following two lemmas are employed in the proof of the main theorem of this section.

Lemma 2.6.9 *If $i \in K(q, t, I)$, where $t \geq 1$, then one of the following two cases must hold.*

- (1) $i \in K(q, t - 1, I)$.
- (2) Processor q accesses the contents of a memory location l at superstep t on input I , and $i \in L(l, t - 1, I)$.

Proof Let $i \in K(q, t, I)$ and $i \notin K(q, t - 1, I)$. Then, the state of processor q at superstep $t - 1$ is the same, regardless of whether the input is I or $I(i)$. The only way for the state of processor q to change at superstep t is for processor q to access the contents of a memory location l , and the contents of l to be different on the inputs I and $I(i)$. This observation implies that $i \in L(l, t - 1, I)$, where m is the memory location accessed by processor q at superstep t . $\square$

Similarly we can prove the following lemma.

Lemma 2.6.10 *If $i \in L(l, t, I)$, where $t \geq 1$, then one of the following three cases must hold.*

- (1) $i \in L(l, t - 1, I)$.
- (2) A processor q writes into memory location l at superstep t on input I , and $i \in K(q, t, I)$.
- (3) A processor q writes into memory location l at superstep t on input $I(i)$.

Proof The proof is similar to that of lemma 2.6.9 and is omitted. $\square$

A Generalized Lower Bound on the Number of Communication Rounds We are now ready to state the following lemma, which gives the key fact needed to establish the desired lower bound for the BSP model.

Lemma 2.6.11 *Let q be a processor of a BSP machine, and let l be a memory location. Then, after t supersteps of an arbitrary computation, we have $|K(q, t, I)| \leq h \cdot (h^2 + 4 \cdot h + 4)^t$ and $|L(l, t, I)| \leq h \cdot (h^2 + 4 \cdot h + 4)^t$, where $h \geq 1$ is the maximum number of messages that can be transmitted or received by any processor in a single communication round.*

Proof Let K_t and L_t satisfy the recurrence equations:

$$K_t = \begin{cases} h & \text{if } t = 0, \\ K_{t-1} + h \cdot L_{t-1} & \text{if } t > 0, \end{cases}$$

and

$$L_t = \begin{cases} h & \text{if } t = 0 \\ (h+3) \cdot K_{t-1} + (h \cdot (h+3) + 1) \cdot L_{t-1} & \text{if } t > 0 \end{cases}$$

We prove by induction the following claim.

Claim 2.6.7 $|K(q, t, I)| \leq K_t$ and $|L(l, t, I)| \leq L_t$, for any $t \geq 0$.

Proof We start with the base case $t = 0$. Initially, each processor maintains in its local memory at most h input symbols. Since at the beginning of superstep zero only the contents of local memory locations can affect processor q and at most one index can affect any memory location (if an input is stored in that location), we have that $|K(q, 0, I)| \leq h$. Similarly, by the end of superstep zero at most h indices can affect the contents of any memory location l ; that is, if the processor associated with memory location l decides to write its state into l . Hence, $|L(l, 0, I)| \leq h$, and the base case follows.

Suppose that the induction hypothesis holds for $K(q, t, I)$ and $L(l, t, I)$, $t \geq 0$. We show that the claim holds for $K(q, t+1, I)$ and $L(l, t+1, I)$. By assumption, the maximum number of messages communicated to or from any processor is at most h , where each message may accommodate the contents of one memory location. Hence, lemma 2.6.9 implies that $K(q, t+1, I) \subseteq K(q, t, I) \cup_i L(m_{w(i)}, t, I)$, $1 \leq i \leq h$. Therefore,

$$|K(q, t+1, I)| \leq |K(q, t, I)| + h \cdot |L(l, t, I)| \leq K_t + h \cdot L_t, \quad (2.11)$$

and the inductive claim for $K(q, t+1, I)$ follows. To obtain a bound on $|L(l, t+1, I)|$ we employ lemma 2.6.10. Then, if $i \in L(l, t+1, I)$ one of the following three cases must hold.

- (1) $i \in L(l, t, I)$.
- (2) A processor q writes into memory location l on input I at superstep $t+1$, and $i \in K(q, t+1, I)$.
By assumption, the maximum number of messages that can be communicated to the processor associated with memory location l is at most h , where each message may accommodate the contents of one memory location. We therefore deduce that the maximum number of messages h_1 that can be communicated (in this step) to memory location l is h .
- (3) A processor q writes into memory location l at superstep $t+1$ on input $I(i)$. In the latter case, we say that index i *causes processor* q to write into memory location l at superstep $t+1$ with input I . Similarly to the previous case, the maximum number of messages h_2 that can be communicated (in this step) to memory location l is $h - h_1$.

Hence, if $h_2 = 0$, then

$$|L(l, t+1, I)| \leq |L(l, t, I)| + h \cdot |K(q, t+1, I)|; \quad (2.12)$$

otherwise

$$|L(l, t+1, I)| \leq |L(l, t, I)| + |Y(l, t+1, I)| + h_1 \cdot |K(q, t+1, I)|, \quad (2.13)$$

where $Y(l, t+1, I)$ denotes the set of those indices that cause some processor q to write into l at superstep $t+1$ with input I , $h_1 = h - h_2$, and $h_2 \geq 1$. The goal is to obtain an upper bound on $|Y(l, t+1, I)|$.

Let $Y(l, t+1, I) = \{y_1, y_2, \dots, y_r\}$, where $r = |Y(l, t+1, I)|$, and suppose that index y_i causes processor $p_{w(i)}$ to write into l at superstep $t+1$ with input I , $1 \leq i \leq r$. That is, processor $p_{w(i)}$ writes into l at superstep $t+1$ on input $I(y_i)$.

Claim 2.6.8 *If $\bar{Y} = \{y_{u_1}, y_{u_2}, \dots, y_{u_{h_2+1}}\}$ is a subset of $Y(l, t+1, I)$ of size h_2+1 , such that for any pair $y_u, y_{u'} \in \bar{Y}$, $p_{w(u)} \neq p_{w(u')}$, then there is a pair $y_z, y_{z'} \in \bar{Y}$ such that $y_z \neq y_{z'}$ and y_z affects processor $p_{w(z')}$ at superstep $t+1$ on input $I(y_{z'})$.*

Proof We prove the claim by contradiction. Suppose that there is no pair $y_z, y_{z'} \in \bar{Y}$ such that $y_z \neq y_{z'}$ and y_z affects processor $p_{w(z')}$ on input $I(y_{z'})$ at superstep $t+1$. Then, on input $I(y_{u_1})(y_{u_2}) \dots (y_{u_{h_2+1}})$ processors $p_{w(u_1)}, p_{w(u_2)}, \dots, p_{w(u_{h_2+1})}$ write into memory location l at superstep $t+1$, which is not possible in our model, since the maximum number of messages that can be communicated to any memory location in a single communication round is at most $h = h_1 + h_2$. $\square$

We next show that $r \leq (h_2 + 3) \cdot K_{t+1}$, $1 \leq h_2 \leq h$. The proof is by contradiction. Suppose that $r > (h_2 + 3) \cdot K_{t+1}$. Along the lines of [53], we construct the bipartite graph $G = (Y \cup V, E)$ consisting of the set of vertices, $Y = Y(l, t+1, I) = \{y_1, y_2, \dots, y_r\}$ and $V = \{v_1, v_2, \dots, v_r\}$, where the v 's are any r distinct new vertices. We say that there is an edge between y_i and v_j if and only if y_i affects $p_{w(j)}$ at superstep $t+1$ on input $I(y_j)$. To get the desired bound on r , we derive upper and lower bounds on the number $|E|$ of edges in G .

By the construction of the bipartite graph, for each vertex v_j , we have at most $K(p_{w(j)}, t+1, I(u_j)) \leq K_{t+1}$ edges. Therefore, we obtain the following upper bound on the number $|E|$ of edges in G .

$$|E| \leq r \cdot K_{t+1}.$$

To obtain a lower bound on $|E|$ we proceed as follows. Let $\mathcal{G}$ denote the set of sub-graphs of G , such that for any sub-graph $\bar{G} = (\bar{Y} \cup \bar{V}, \bar{E})$ in $\mathcal{G}$ the following two conditions are satisfied:

- (1) $|\bar{Y}| = h_2 + 1$,
- (2) for any pair $y_u, y_{u'}$ in $\bar{Y}$, $p_{w(u)} \neq p_{w(u')}$.

Then, by claim 2.6.8, $\bar{G}$ contains at least one edge of G . We therefore obtain that $|E| \geq |\mathcal{G}|$. It is not difficult to verify that by placing a vertex $y_i \in Y(l, t+1, I)$ in $\bar{G}$ we disqualify at most K_{t+1} other vertices from being placed in $\bar{G}$, since there are at most $|K(p_{w(j)}, t+1, I)| \leq K_{t+1}$ indices y_j such that $p_{w(i)} = p_{w(j)}$. Hence, we obtain the following bound on the size of $\mathcal{G}$.

$$|\mathcal{G}| \geq \frac{r \cdot (r - K_{t+1}) \cdot \dots \cdot (r - h_2 \cdot K_{t+1})}{(h_2 + 1)!}.$$

Combining this inequality with the facts that $r > (h_2 + 3) \cdot K_{t+1}$, $|E| \leq r \cdot K_{t+1}$, and $|E| \geq |\mathcal{G}|$, we obtain that

$$\begin{aligned} r \cdot K_{t+1} &\geq \frac{r \cdot (r - K_{t+1}) \cdot \dots \cdot (r - h_2 \cdot K_{t+1})}{(h_2 + 1)!} \\ &\geq K_{t+1}^{h_2+1} \cdot \frac{r/K_{t+1}}{h_2 + 1} \cdot \frac{r/K_{t+1} - 1}{h_2} \cdot \dots \cdot \frac{r/K_{t+1} - h_2}{1} \\ &\geq K_{t+1}^{h_2+1} \left(\frac{r}{(h_2 + 1) \cdot K_{t+1}} \right)^{h_2+1} \\ &= \left(\frac{r}{h_2 + 1} \right)^{h_2+1}. \end{aligned}$$

Since $h_2 \geq 1$ and $K_{t+1} \geq 1$, this inequality implies that

$$\begin{aligned} r &\leq ((h_2 + 1)^{h_2+1} \cdot K_{t+1})^{1/h_2} \\ &\leq (h_2 + 1) \cdot K_{t+1} + (h_2 + 1)^{1/h_2} \cdot K_{t+1} \\ &\leq (h_2 + 3) \cdot K_{t+1}, \end{aligned}$$

which cannot be true, since we have assumed that $r > (h_2 + 3) \cdot K_{t+1}$. Hence, we conclude that $r \leq (h_2 + 3) \cdot K_{t+1}$. We therefore obtain, by way of equations 2.11, 2.12, and 2.13, the following bound on $|L(l, t + 1, I)|$.

$$\begin{aligned} |L(l, t + 1, I)| &\leq L_t + (h_2 + 3) \cdot K_{t+1} + h_1 \cdot K_{t+1} \\ &\leq L_t + (h + 3) \cdot (K_t + h \cdot L_t) \\ &= (h + 3) \cdot K_t + (h \cdot (h + 3) + 1) \cdot L_t \\ &= L_{t+1}. \end{aligned}$$

The claim follows. $\square$

We can now complete the proof of the lemma by establishing bounds on K_t and L_t . We can verify that the recurrences defining K_t and L_t have the following solutions.

$$\begin{aligned} K_t &= a_1 \rho_1^t + a_2 \rho_2^t \\ L_t &= b_1 \rho_1^t + b_2 \rho_2^t, \end{aligned}$$

where $\rho_1 = ((h^2 + 3 \cdot h + 2) + (h + 2) \cdot \sqrt{h^4 + 6 \cdot h^3 + 13 \cdot h^2 + 12 \cdot h})/2$, $\rho_2 = ((h^2 + 3 \cdot h + 2) - (h + 2) \cdot \sqrt{h^4 + 6 \cdot h^3 + 13 \cdot h^2 + 12 \cdot h})/2$, $a_i = (-h \cdot \rho_i + (h^3 + 2 \cdot h^2 + h))/(2/\rho_i - (h^2 + 3 \cdot h + 2))$ and $b_i = (-h \cdot \rho_i - (h^2 + 2 \cdot h))/(2/\rho_i - (h^2 + 3 \cdot h + 2))$, $1 \leq i \leq 2$.¹⁵ Hence, after some calculations, we obtain that $K_t \leq h \cdot (h^2 + 4 \cdot h + 4)^t$ and $L_t \leq h \cdot (h^2 + 4 \cdot h + 4)^t$, and the lemma follows. $\square$

¹⁵**Bounding K_t and L_t .** Let

$$K_t = \begin{cases} h & \text{if } t = 0, \\ K_{t-1} + h \cdot L_{t-1} & \text{if } t > 0, \end{cases}$$

Specific Lower Bounds on the Number of Communication Rounds

Theorem 2.6.8 *Let $f : \{0, 1\}^n \rightarrow \{0, 1\}$ have a critical input. Then, $\Omega(\lg n / \lg(h + 1))$ supersteps are required to compute f on a BSP machine with any number of processors, where $h \geq 1$ is the maximum number of messages that can be transmitted or received by any processor in a single superstep.*

and

$$L_t = \begin{cases} h & \text{if } t = 0, \\ (h + 3) \cdot K_{t-1} + (h \cdot (h + 3) + 1) \cdot L_{t-1} & \text{if } t > 0. \end{cases}$$

We derive closed forms for K_t and L_t by using generating functions. From the above definitions we have

$$\begin{aligned} K_t &= K_{t-1} + h \cdot L_{t-1} + h \cdot [t = 0], \\ L_t &= (h + 3) \cdot K_{t-1} + (h \cdot (h + 3) + 1) \cdot L_{t-1} + h \cdot [t = 0], \end{aligned}$$

for all n . Then, for $K(z) = \sum_t k_t \cdot z^t$ and $L(z) = \sum_t l_t \cdot z^t$ it is not difficult to verify that

$$\begin{aligned} K(z) &= \sum_t k_{t-1} \cdot z^t + h \cdot \sum_t l_{t-1} \cdot z^t + h \cdot \sum_t [t = 0] \cdot z^t \\ &= \sum_t k_t \cdot z^{t+1} + h \cdot \sum_t l_t \cdot z^{t+1} + h \\ &= z \cdot K(z) + h \cdot z \cdot L(z) + h, \end{aligned}$$

and similarly,

$$\begin{aligned} L(z) &= (h + 3) \cdot \sum_t k_{t-1} \cdot z^t + (h \cdot (h + 3) + 1) \cdot \sum_t l_{t-1} \cdot z^t + h \cdot \sum_t [t = 0] \cdot z^t \\ &= (h + 3) \cdot \sum_t k_t \cdot z^{t+1} + (h \cdot (h + 3) + 1) \cdot \sum_t l_t \cdot z^{t+1} + h \\ &= (h + 3) \cdot z \cdot K(z) + (h \cdot (h + 3) + 1) \cdot z \cdot L(z) + h. \end{aligned}$$

This system of two equations in two unknowns yields

$$K(z) = \frac{-(h^3 + 2 \cdot h^2 + h) \cdot z + h}{1 - (h^2 + 3 \cdot h + 2) \cdot z + z^2}, \quad (2.14)$$

$$L(z) = \frac{(h^2 + 2 \cdot h) \cdot z + h}{1 - (h^2 + 3 \cdot h + 2) \cdot z + z^2}. \quad (2.15)$$

The required closed forms for K_t and L_t are, respectively, the coefficients of z^t when $K(z)$ and $L(z)$ are expanded in a power series. The following result due to [103] is employed to compute these coefficients.

Lemma 2.6.12 ([103]) *If $R(z) = P(z)/Q(z)$, where P and Q are polynomials, $Q(z) = q_0 \cdot (1 - \rho_1 \cdot z) \cdot \dots \cdot (1 - \rho_k \cdot z)$, and the numbers $\rho_1, \dots, \rho_k$ are distinct, and if $P(z)$ is a polynomial of degree less than k , then the coefficient of z^t in $R(z)$ is*

$$\alpha_1 \cdot \rho_1^t + \dots + \alpha_k \cdot \rho_k^t, \quad \text{where } \alpha_i = \frac{-\rho_i \cdot P(1/\rho_i)}{\frac{dQ}{dz}(1/\rho_i)}.$$

In our case we have

$$\begin{aligned} Q(z) &= 1 - (h^2 + 3 \cdot h + 2) \cdot z + z^2 \\ &= (1 - \rho_1 \cdot z)(1 - \rho_2 \cdot z) \end{aligned}$$

and, $\frac{dQ}{dz}(z) = -(h^2 + 3 \cdot h + 2) + 2 \cdot z$, where $\rho_1 = ((h^2 + 3 \cdot h + 2) + (h + 2) \cdot \sqrt{h^4 + 6 \cdot h^3 + 13 \cdot h^2 + 12 \cdot h})/2$ and $\rho_2 = ((h^2 + 3 \cdot h + 2) - (h + 2) \cdot \sqrt{h^4 + 6 \cdot h^3 + 13 \cdot h^2 + 12 \cdot h})/2$. Therefore, after some calculations, we obtain from lemma 2.6.12, and equations 2.14 and 2.15 that

$$\begin{aligned} K_t &= a_1 \cdot \rho_1^t + a_2 \cdot \rho_2^t \\ L_t &= b_1 \cdot \rho_1^t + b_2 \cdot \rho_2^t, \end{aligned}$$

where $a_i = (-h \cdot \rho_i + (h^3 + 2 \cdot h^2 + h))/(2/\rho_i - (h^2 + 3 \cdot h + 2))$ and $b_i = (-h \cdot \rho_i - (h^2 + 2 \cdot h))/(2/\rho_i - (h^2 + 3 \cdot h + 2))$, $1 \leq i \leq 2$.

Proof According to our conventions, the output value should be reflected in the state of processor p_0 at the end of the computation. Thus, the number of indices that must affect processor p_0 is n ; that is, $|K(p_0, t, I)| \geq n$. By lemma 2.6.11, we have that $h \cdot (h^2 + 4 \cdot h + 4)^t \geq |K(p_0, t, I)| \geq n$, and hence the theorem follows. $\square$

Using the fact that there is a simple reduction from the problem of computing the Boolean *OR* of n inputs to the problem of sorting a multi-set of size n , and the fact that the *OR* function has a critical input [53], we derive the following result.

Theorem 2.6.9 *The problem of sorting a multi-set of size n , requires $\Omega(\lg n / \lg(h + 1))$ supersteps on a BSP machine with any number of processors, where $h \geq 1$ is the maximum number of messages that can be transmitted or received by any processor in a single superstep.*

A Lower Bound on the Number of Messages Communicated Along the lines of [4], we establish the following bound on the communication requirements of BSP sorting.

Theorem 2.6.10 *The problem of sorting a multi-set of size n , by storing only $O(n/p)$ items per processor, requires the communication of $\Omega((n/p) \cdot (\lg n / \lg(n/p)))$ items on a BSP machine with any number of processors $p \leq n$.*

Proof Let $\mathcal{A}$ be some BSP sorting algorithm, and let $n_i = c_i \cdot (n/p)$ denote the number of data items residing in the local memory of processor p_i , $0 \leq i < p$, for some positive constant c_i . Moreover, let π_1 denote the number of relative orderings that can be distinguished by algorithm $\mathcal{A}$, at the beginning of the computation, by only comparing the items that reside in the local memories of the p_i 's, for all $0 \leq p_i < p$. Then,

$$\pi_1 = \prod_{i=0}^{p-1} n_i! \leq \prod_{i=0}^{p-1} \left(c \cdot \frac{n}{p} \right)^{c \cdot (n/p)},$$

where $c = \max_i \{c_i\}$. Furthermore, let m denote the maximum number of items communicated to or from any processor during the communication steps of algorithm $\mathcal{A}$. Then, the additional number of relative orderings π_2 that can be distinguished by algorithm $\mathcal{A}$ is bounded above by,

$$\pi_2 \leq \prod_{i=0}^{p-1} n_i^m \leq \prod_{i=0}^{p-1} \left(c \cdot \frac{n}{p} \right)^m,$$

where $c = \max_i \{c_i\}$. Thus, the overall number of permutations π that can be distinguished, is bounded above by,

$$\pi = \pi_1 \cdot \pi_2 \leq \prod_{i=0}^{p-1} \left(c \cdot \frac{n}{p} \right)^{c \cdot (n/p)} \cdot \prod_{i=0}^{p-1} \left(c \cdot \frac{n}{p} \right)^m = \left(c \cdot \frac{n}{p} \right)^{p \cdot (c \cdot (n/p) + m)}.$$

Let U denote the total number of relative orderings that can be distinguished by algorithm $\mathcal{A}$. Then,

$$U \leq \left(c \cdot \frac{n}{p} \right)^{p \cdot (c \cdot (n/p) + m)},$$

and therefore,

$$m \geq \frac{\lg U}{p \cdot \lg(c \cdot (n/p))} - c \cdot \frac{n}{p}.$$

Since $U \geq n!$, we obtain, for $p \leq n$,

$$m \geq \frac{n \cdot \lg n}{p \cdot \lg(c \cdot (n/p))} - c \cdot \frac{n}{p} = \Omega\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right).$$

The desired result follows. $\square$

A Lower Bound on both the Number of Communication Rounds and the Number of Messages Communicated By combining the results of theorems 2.6.9 and 2.6.10 we obtain the following bound on BSP sorting.

Theorem 2.6.11 *The problem of sorting a multi-set of size n , by storing only $O(n/p)$ data items per processor, requires time $\Omega((n/p) \cdot \lg n)$ and $L \cdot \Omega(\lg n / \lg(n/p)) + g \cdot \Omega((n/p) \cdot (\lg n / \lg(n/p)))$, for computation and communication respectively, on a BSP machine with any number of processors $p \leq n$.*

Proof The bound on the computation time is trivial [133], and the lower bound on the communication time derives directly from theorems 2.6.9 and 2.6.10. $\square$

2.6.8.4 Applications of Sorting in Computational Geometry Algorithms

We have presented in section 2.6.8 a variant of the pipelined divide-and-conquer strategy of [47] as applied to the following problem.

Problem Given a k -ary tree T such that each leaf node contains a single input item drawn from a linearly ordered set, determine, for each internal node v , the non-decreasing sequence $S[v]$ that contains all the items initially stored at the leaves of the sub-tree rooted at v .

In theorem 2.6.7, we established that such a computation can be handled in time $O(n \cdot \lg n)$ and $L \cdot O(\lg n / \lg(n/p)) + g \cdot O((n/p) \cdot \lg n / \lg(n/p))$ for computation and communication respectively. The significance of the aforementioned algorithm – for the scope of this section – derives from the fact that when the algorithm terminates, we can assume that we have the non-decreasing sequence $S[v]$ at each node v and the rank-info $\text{rank}(S[z_j], S[v])$, where z_j is a child of v , $1 \leq j \leq k$. In addition, the sequences generated at each sibling pair of nodes are cross-ranked. Therefore, if z_i , for all $1 \leq i \leq k$, are the children of v , then the predecessor of an arbitrary element of $S[z_j]$ in the sequences $S[z_i]$, for all $1 \leq i \leq k$, can be efficiently determined. It is this very property of the pipelined merge-sort algorithm of section 2.6.8 that is exploited in a multiplicity of previous approaches in order to realize optimal computational geometry algorithms [16, 50, 119]. The following problems admit optimal solutions on the PRAM model by way of the pipelined merge-sort method.

- Segment intersection detection, trapezoidal decomposition, planar-point location, fractional cascading, three-dimensional maxima, two-set dominance counting and visibility from a point [16].

- Convex hull construction and all-nearest neighbors [50].

It is our belief that by employing the pipelined merge-sort algorithm of section 2.6.8 it is possible to derive communication efficient BSP solutions for the aforementioned problems – there is no fundamental impediment known that would preclude this derivation done by communication efficient BSP algorithms.¹⁶

2.6.9 Duplicate Item Handling

The explicit assumption in the analysis of the sorting algorithms of sections 2.6.6 and 2.6.7 is that items are unique; in case of duplicate items, the optimality bounds presented in the aforementioned sections do not apply. To remedy the situation, in the remainder of this section we present a simple scheme for handling duplicates.¹⁷

The algorithmic scheme that follows pertains to procedure DET_SORT of section 2.6.6; a similar scheme can be applied to procedure RND_SORT of section 2.6.7. Recall that algorithm DET_SORT consists of two sub-procedures: SEQUENTIAL_SORT and RECURSIVE_SORT. Procedure SEQUENTIAL_SORT is implemented by means of some stable sequential sorting algorithm. In procedure RECURSIVE_SORT all items are appended the address of their memory location. The attached information is only used in stages 2–4 (note that these stages require distinct items to achieve stability and load-balance). The computation and communication cost of these stages is within the lower order terms of the analysis of section 2.6.6, and therefore, the optimality claims are not affected by the aforementioned algorithmic scheme.

2.7 Converting Sorting Algorithms into Fast Randomized Ones through Searching

In this section we design, analyze and implement an algorithmic transformation for *converting sorting algorithms to fast randomized ones via searching*. For the sake of simplicity, in the remainder of this section, we confine the study of the algorithmic transformation to the sequential case; then, the generalization to the parallel case is trivial.¹⁸ The material of this section is based on the following papers:

- (1) “Converting sorting algorithms to fast randomized ones via searching.” Submitted for journal publication, 1996 [83]. Also as Technical Report PRG-TR-22-97, Computing Laboratory, Oxford University, June 1997 [91].

2.7.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a sequences whose items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments. Without loss of

¹⁶Unfortunately, the underlying details of these solutions are rather complicated and technical, and therefore, beyond the scope of this work.

¹⁷One way to allow duplicates is by making items distinct. This can be achieved by attaching to each item the address of the memory location it is stored in. For data types whose bit or byte length is comparable to that of their memory location, such a transformation leads – in most cases – to doubling the overall number of comparisons performed. For more complex data types such as strings of characters the extra cost may be negligible.

¹⁸Recall that the discussion of this section is motivated by the assumption in section 2.6.7 that the multiplicative constant factor associated with the time complexity of sorting is larger than the multiplicative constant factor associated with the time complexity of binary searching. In this section we study the validity and implications of the aforementioned claim both theoretically and experimentally.

generality, we assume that all the items appearing in sequence X are distinct – if the items are not distinct we can make them so by appending to them their index. Additionally, let SEQ_SORT be a sequential sorting procedure, and similarly, let SEQ_SEARCH be a sequential searching procedure. We consider, in this section, the problem of providing a sequential randomized procedure (SEQ_RND_SORT), that is based on procedures SEQ_SORT and SEQ_SEARCH and that with high probability requires less time than SEQ_SORT for sorting sequence X into a non-decreasing sequence – say $\langle x_{\pi(1)}, \dots, x_{\pi(n)} \rangle = \text{non-decreasing}(X)$.

2.7.2 Cost Model

The performance of the algorithmic transformation of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations. Furthermore, for some constants c_{srt} and c_{srch} , time $c_{srt} \cdot n \cdot \lg^\gamma n$ and $c_{srch} \cdot \lg^\beta n$ is charged respectively for sorting a sequence of size n and for searching (for a single item) in a non-decreasing sequence of size n .

2.7.3 An Algorithmic Transformation

In this section, we provide an algorithmic transformation for *converting sorting algorithms to fast randomized ones via searching* that is based on *over-sampling* techniques [75, 183]. In particular, we claim that, if SEQ_SORT is a sequential sorting procedure and SEQ_SEARCH is a sequential searching procedure – with time complexity $c_{srt} \cdot n \cdot \lg^\gamma n$ and $c_{srch} \cdot \lg^\beta n$ respectively – then there exists a randomized sequential sorting procedure SEQ_RND_SORT that requires time (with high probability) $(c_{srch} + o(1)) \cdot n \cdot \lg^\beta n$.

Informal Description of Algorithm The optimal randomized sequential sorting algorithm is based on over-sampling techniques and is non-iterative in nature. Initially, a sub-sequence is constructed (in a randomized way) that is an appropriate over-sample of the input. Subsequently, the so constructed sub-sequence is sorted, by means of the given sequential sorting algorithm, and a sub-sequence of size $k - 1$ (where k is appropriately chosen) is constructed that is an appropriate sample of the resulting sorted sequence. This sample is employed to partition, by means of the given sequential searching algorithm, the input into k non-overlapping partitions. Finally, each partition is independently sorted, by means of the given sequential sorting algorithm.

Our main result is summarized in the following theorem.

Theorem 2.7.1 *Consider sequential sorting and searching algorithms with execution times $c_{srt} \cdot n \cdot \lg^\gamma n$ and $c_{srch} \cdot \lg^\beta n$ respectively, where $0 \leq \beta \leq \gamma$. Then, for any constant $\rho > 0$, there exists a randomized sequential algorithm for sorting a sequence of size n that requires [with probability $1 - O(n^{-\rho})$] time $(c_{srch} + o(1)) \cdot n \cdot \lg^\beta n$.*

Proof We assume that the input is in array $X[1, 2, \dots, n]$. The following auxiliary arrays are also required: $T[1, 2, \dots, n]$ holds the sorted output, and $A[1, 2, \dots, n - k + 1]$, $C[0, 1, \dots, k]$, $D[0, 1, \dots, k]$, $S[1, 2, \dots, k \cdot s - 1]$ and $\bar{S}[1, 2, \dots, k - 1]$ provide temporary working space, where $k, s \geq 1$ and $k \cdot s < n$. The space requirements can be reduced by re-utilizing unused space, that is, arrays T and $\bar{S}$ are necessary and sufficient for the algorithm to execute. We also assume the existence of a random number generator, realized by function $\text{RND}(k)$, that generates uniformly distributed integers in the range $[0, 1, \dots, k - 1]$.

Let SEQ_SORT be the sequential sorting procedure, and similarly, let SEQ_SEARCH be the sequential searching procedure. The resulting procedure is presented in figure 2.44. With reference to this figure

the following conventions apply. Procedure $\text{SEQ_SORT}(X, n)$ requires two parameters: (i) the sequence of items X to be sorted, and (ii) the cardinality n of set X . Procedure $\text{SEQ_SEARCH}(x, X, n)$ returns the rank of x in X and requires three parameters: (i) the element x to be located, (ii) the ordered sequence X to be searched, and (iii) the cardinality n of sequence X .

```

begin PROCEDURE SEQ_RND_SORT ( $X, n$ )
1.  for  $i = 1$  to  $k \cdot s - 1$ 
2.      do let  $r = \text{RND}(n - i + 1) + 1$  ;
3.      let  $S[i] = X[r]$  ;
4.      let  $X = X - X[r]$  ;
5.  SEQ_SORT( $S, k \cdot s - 1$ ) ;
6.  for  $i = 1$  to  $k - 1$ 
7.      do let  $\bar{S}[i] = S[i \cdot s]$  ;
8.  let  $X = X \cup (S - \bar{S})$  ;
9.  for  $i = 1$  to  $n - k + 1$ 
10.     do let  $A[i] = \text{SEQ\_SEARCH}(X[i], \bar{S}, k - 1)$  ;
11.  for  $i = 1$  to  $k$ 
12.     do let  $C[i] = 0$  ;
13.  for  $i = 1$  to  $n - k + 1$ 
14.     do let  $C[A[i]] = C[A[i]] + 1$  ;
15.  for  $i = 2$  to  $k$ 
16.     do let  $C[i] = C[i] + C[i - 1]$  ;
17.     let  $D[i] = C[i]$  ;
18.  for  $i = n - k + 1$  downto 1
19.     do let  $T[C[A[i]]] = X[i]$  ;
20.     let  $C[A[i]] = C[A[i]] - 1$  ;
21.  let  $D[0] = 0$  ;
22.  for  $i = 1$  to  $k$ 
23.     do SEQ_SORT( $T[D[i - 1] + 1, \dots, D[i]], D[i] - D[i - 1]$ ) ;
24.   $T = T \cup_{\text{merge}} \bar{S}$ 
end SEQ_RND_SORT

```

Figure 2.44: Procedure SEQ_RND_SORT.

Analysis In lines 1–4, a sample S of size $k \cdot s - 1$ is selected at random without replacement among the n input items of X . The “over-sampling” parameter s is chosen so that it satisfies the conditions of lemma A.2.2. We note that the operation $X = X - X[r]$ can be easily implemented by swapping elements $X[r]$ and $X[n - i + 1]$. Hence, this step requires time $O(k \cdot s)$. In line 5 the sample S of size $k \cdot s - 1$ is sorted by employing procedure SEQ_SORT. By the definition of procedure SEQ_SORT, this step takes time $c_{\text{srt}} \cdot (k \cdot s) \cdot \lg^\gamma(k \cdot s)$. In lines 6–7, the sorted sample S is employed to determine in $O(k)$ time the sub-sequence $\bar{S}$ of $k - 1$ splitters that evenly partition S . In lines 8–10, the bucket destination of each element in $X - \bar{S}$ is determined with respect to the $k - 1$ splitters by employing procedure SEQ_SEARCH. Since, $|X - \bar{S}| = n - k + 1$, by the definition of procedure SEQ_SEARCH, this step takes time $c_{\text{srch}} \cdot (n - k + 1) \cdot \lg^\beta(k - 1)$. In lines 11–20, the items in $X - \bar{S}$ are redistributed according to their bucket

destination. This step is equivalent to one of integer sorting [133], where the $n - k + 1$ integers to be sorted belong to the range $[1, 2, \dots, k]$, induced by the $k - 1$ splitters. It follows that this step takes time $O(n)$. Subsequently, in lines 21–23, the items in each bucket are sorted by employing procedure SEQ_SORT. By way of lemma A.2.2, the size of each bucket is bounded above by $\lceil (1 + \varepsilon) \cdot (n - k + 1)/k \rceil$ with high probability, for some appropriately chosen $0 < \varepsilon < 1$. Hence, by the definition of procedure SEQ_SORT, this step requires time $c_{srt} \cdot (1 + \varepsilon) \cdot (n - k + 1) \lg^\gamma ((1 + \varepsilon) \cdot (n - k + 1)/k)$ with high probability. Finally, in line 24, the sorted sequence of splitters $\bar{S}$ is merged with the sorted sequence of items. We note that no actual merging has to occur since, by way of the analysis of lines 8–20, the rank of $\bar{S}$ in the sorted sequence of items is already known. Hence, this step requires time $O(k)$. We conclude that with high probability the overall time of algorithm SEQ_RND_SORT is bounded above, for $0 < \varepsilon < 1$, by

$$c_{srch} \cdot (n - k + 1) \cdot \lg^\beta(k - 1) + O\left((n - k + 1) \cdot \lg^\gamma \frac{(n - k + 1)}{k} + (k \cdot s) \cdot \lg^\gamma(k \cdot s) + n\right).$$

Then, for any constant $\rho > 0$, by substituting in the aforementioned expression and in lemma A.2.2 for constant $0 < \varepsilon < 1$, $s = \Theta(\lg n)$ and $k = n/(\lg^{1+\gamma} n)$, we obtain that procedure SEQ_RND_SORT requires time [with probability $1 - O(n^{-\rho})$] $(c_{srch} + o(1)) \cdot n \cdot \lg^\beta n$.

Remark A tighter probabilistic analysis reveals that the success probability of the algorithm implied in the aforementioned theorem can be shown to be

$$1 - O\left(n^{-\rho \cdot \lg^\delta n}\right),$$

for any constant $\rho > 0$ and $\delta > 0$.

The theorem follows. $\square$

In practical applications procedure SEQ_SEARCH realizes a binary search method, and therefore, requires $c_{srch} \cdot \lg n$ time, for some constant $c_{srch} > 0$, to search for an item in a sequence of size n . Hence, $\beta = 1$ in the previous analysis, and our method can be used to speedup asymptotically any sorting procedure SEQ_SORT with time complexity $c_{srt} \cdot n \cdot \lg^\gamma n$, where $\gamma > 1$. For the special case $\gamma = 1$ [115, 133], our method provides an algorithm that improves upon the multiplicative constant factor of the underlying sorting algorithm, provided that $c_{srt} > c_{srch}$.

It is not difficult to verify that the parallelization of the sequential algorithmic transformation of this section is straightforward, given parallel versions of procedures SEQ_SORT and SEQ_SEARCH. In fact, the randomized algorithm of section 2.6.7 is a special case of the algorithmic transformation of this section – SEQ_SORT is realized by means of a sub-optimal BSP sorting algorithm and SEQ_SORT by means of an optimal BSP searching algorithm based on ordered h -level graphs (refer to section 2.9.4).

2.7.3.1 Implementation and Experimental Results

In this section we discuss an implementation – in C – of a variant of the randomized sequential algorithmic transformation of section 2.7.3.¹⁹

¹⁹We decided to study the sequential version (as opposed to the BSP version) of the randomized algorithmic transformation on a twofold basis: (i) the performance of the resulting BSP sorting algorithm can be extrapolated from the performance of the sequential sorting algorithm (by independently considering the performance of the BSP implementations of the constituent parts of the algorithm), and (ii) the resulting sequential sorting algorithm can be easily compared to previous (well-known) approaches on sorting.

Implementation The prototype adopted for the sequential randomized sorting procedure is outlined in figure 2.23.

```
void rnd_sort (byte *src, byte **trg, int n, int sb, int (*search)(), void (*sort)(), int (*cmp)()) ;
    src      - input array.
    trg      - output array.
    n        - input size.
    sb       - item size.
    search    - generic search function.
    sort      - generic sort function.
    cmp      - generic compare function.
```

Figure 2.45: Procedure for sequential randomized sorting.

The aim of function (procedure) `rnd_sort` is to provide a data-type independent high-performance randomized sorting algorithm implementation.²⁰

Experimental Results The theoretical performance of the algorithm is summarized in theorem 2.7.1. To determine the practical significance of the aforementioned result we benchmarked our implementation (`rnd_sort`) on a machine with a varying number of input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). Figures 2.46, 2.47 and 2.48 present the execution time of procedure `rnd_sort` (modified merge sort (M^*), quick sort (Q^*) and heap sort (H^*) as suggested by the algorithmic transformation of section 2.7.3) along with the execution time of the corresponding unmodified comparing algorithms – merge sort (M) [133], quick sort (Q) [115] and heap sort (H) [133].²¹

As shown in figures 2.46, 2.47 and 2.48, the execution time of procedure `rnd_sort` scales with the input size (approximately as $n \cdot \lg n$). Additionally, as the problem size increases, the performance of procedure `rnd_sort` converges, for all three underlying algorithms, to the practical performance of binary search. Both experimental results are in accordance with the theoretical analysis of section 2.7.3. In most cases of interest, our randomized algorithms outperform the corresponding comparing algorithms by a wide margin. Figure 2.49 establishes the aforementioned claim.

Obviously, as indicated in figure 2.49, the speedup of our randomized algorithms is consistently greater than 1 (for all large enough problem sizes), and ranges from 0.88 to 2.87.

2.8 Selection

In this section we design, analyze and implement BSP algorithms for a fundamental combinatorial operation, namely *selection*. The selection operation [133] has always been a fruitful research topic both in sequential [39, 70, 118, 128, 133, 166, 191] and parallel computation [6, 27, 49, 187, 208, 175]. Parallel selection is a frequently recurring operation in computer science and is fundamental for many parallel applications, such as improving the performance of parallel priority queues and implementing branch-and-bound algorithms [27, 78, 79, 189]. The material of this section is based on the following papers:

²⁰The terms generic and data-type independent imply that procedure `rnd_sort` can be employed to handle *any* data type. To this end, a comparison function `cmp` is used to facilitate the comparison of two elements e_1 and e_2 . In particular, the `cmp` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2 .

²¹Algorithms merge sort (M^*), quick sort (Q^*) and heap sort (H^*) result from applying the algorithmic transformation of section 2.7.3 to the well-known algorithms merge sort (M) [133], quick sort (Q) [115] and heap sort (H) [133].

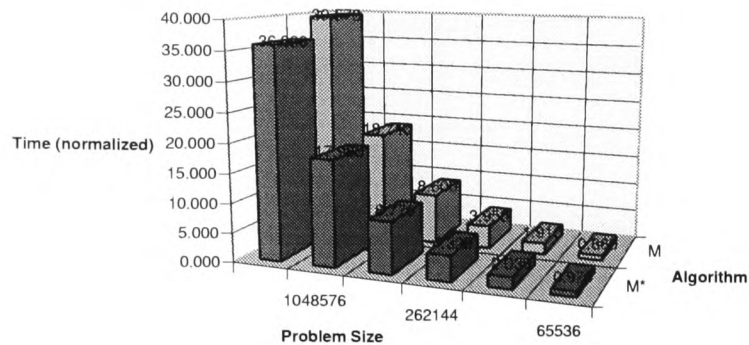


Figure 2.46: Normalized execution time on a SGI Power Challenge for randomized merge sort (M*) and merge sort (M*) – uniform distribution (U).

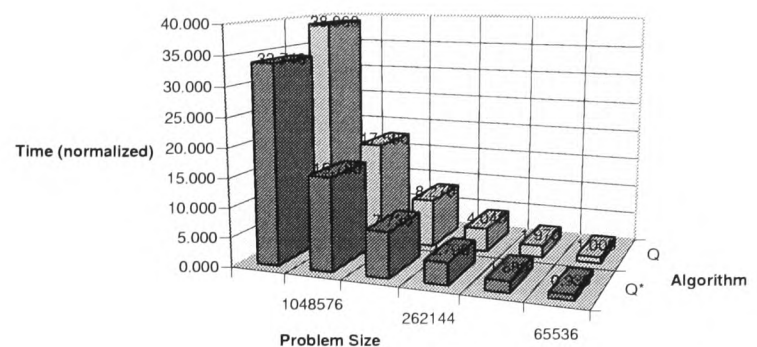


Figure 2.47: Normalized execution time on a SGI Power Challenge for randomized quick sort (Q*) and quick sort (Q*) – uniform distribution (U).

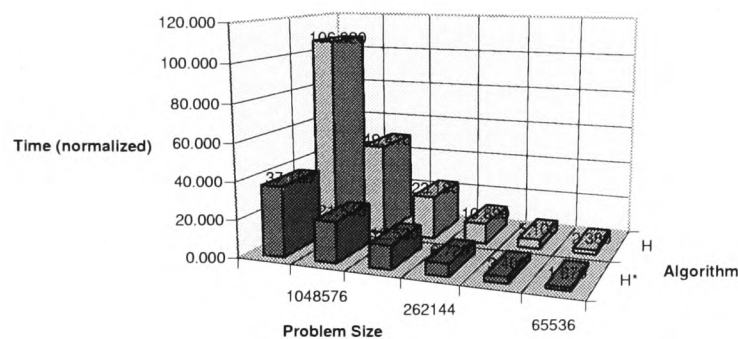


Figure 2.48: Normalized execution time on a SGI Power Challenge for randomized heap sort (H*) and heap sort (H*) – uniform distribution (U).

- (1) “Portable and scaleable algorithm design on the IBM SP2: The bulk-synchronous parallel paradigm.” In *Proceedings of the Sup-Eur’96 Conference*, Krakow, Poland, September 1996 [11].
- (2) “Deterministic sorting and randomized median finding on the BSP model.” In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996 [77].
- (3) “Concurrent heaps on the BSP model.” Technical Report PRG-TR-14-96, Computing Laboratory, Oxford University, June 1996 [78].
- (4) “Selection on the bulk-synchronous parallel model with applications to priority queues.” In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, August 1996 [79].
- (5) “Selection on the BSP model with applications to parallel priority queues.” Submitted for journal publication, 1997 [94].

2.8.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a sequence whose items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments. Without loss of generality, we assume that all the items appearing in sequence X are distinct – if the items are not

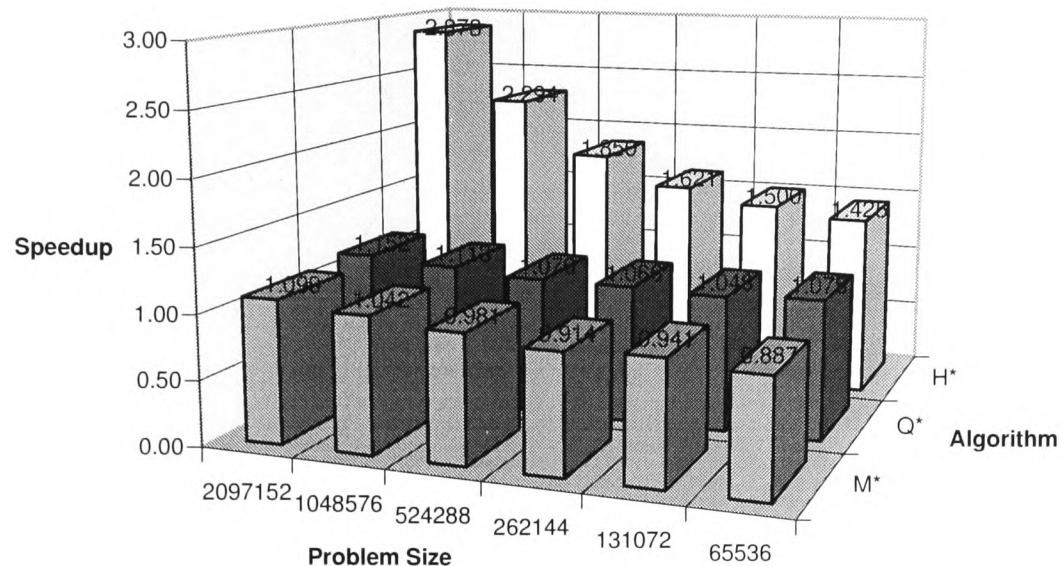


Figure 2.49: Speedup on a SGI Power Challenge for randomized merge sort (M^*), quick sort (Q^*) and heap sort (H^*) with respect to merge sort (M), quick sort (Q) and heap sort (H) respectively – uniform distribution (U).

distinct we can make them so by appending to them their index. We consider, in this section, the problem of *selecting* from sequence X the C -th smallest item – say, $e = x_{\pi(C)}$, where $\langle x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)} \rangle$ is the sequence X in non-decreasing order.

2.8.2 Cost Model

The performance of the BSP selection algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations. Furthermore, constant time c_{sel} is charged for each comparison followed by a counter increment – within the context of this section we denote c_{sel} by c_* .

Optimal $\Theta(n)$ time sequential selection algorithms are well-known [54, 133]. In the following sections, we design, analyze and implement a simple optimal BSP selection algorithm – optimality is realized for an ample range of the BSP parameters.

2.8.3 Related Work and New Results

There has been a significant amount of previous work – mainly on the PRAM model – on parallel selection (e.g., [6, 49, 187, 208] on the PRAM model, [27] on the BSP* model,²² [175] on the hyper-cube model). The exhibit in table 2.7 summarizes the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality. To simplify the exposition, selection is assumed to transpire on an input set of uniformly distributed items.

With reference to table 2.7 we note that the optimal randomized algorithm of [187] employs a non-standard (stronger) PRAM model, and therefore, compares unfairly with other algorithms.

²²The BSP* is an augmentation of the standard BSP model that takes into explicit account the size of communicated messages. The standard BSP model may also take message size into consideration by absorbing message size, for small messages, into one of its parameters, L , and for large messages, into another parameter, g .

Refs	Computation	Communication	Constants	Implem.	Procs
[6]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg^2 n}\right)$
[187]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[208]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■ ■	$O\left(\frac{n}{\lg n \cdot \lg \lg n}\right)$
[49]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■ ■	$O\left(\frac{n}{\lg n \cdot \lg^* n}\right)$
[175]	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[27]	$O\left(\frac{n}{p}\right)$	$o\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg^4 n}\right)$
Sec 2.8.5	$c_* \cdot \frac{n+C}{p} + o\left(\frac{n}{p}\right)$	$o\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$

Table 2.7: Performance comparison of parallel selection algorithms – the cyan color is associated with randomized approaches.

The optimal BSP selection algorithm of section 2.8.5 improves upon previous approaches on the following particulars:

- (1) the asymptotic communication time requirements of [6, 27, 49, 175, 187, 208],²³
- (2) the multiplicative constant factor requirements of [6, 27, 49, 175, 187, 208],
- (3) the implementation effort requirements of [49, 208], and
- (4) the parallel slackness requirements (to achieve optimality) of [6, 27, 49, 208].

The principal contribution, however, of the selection algorithm of section 2.8.5 is attributed to the *simplicity* of the underlying BSP algorithm (refer to case 3), the *low communication cost* (refer to case 1), and the *small constant multiplicative factors* (refer to case 2). Additionally, our techniques provide *tighter probabilistic bounds* than previous randomized methods [27, 175, 187] and *one-optimality* for an ample range of the BSP parameters.

2.8.4 A Randomized Sorting Algorithm with $n^{1+\epsilon}$ Processors

In this section we present a simple randomized BSP algorithm for the problem of sorting n items with $p = n^{1+\epsilon}$ processors, for any constant $0 < \epsilon < 1$. The algorithm is a non-trivial generalization of an algorithm due to [167]; the fundamental difference between our randomized algorithm and the deterministic algorithm of [167] is that our methods are based on random-sampling techniques [75] and exclusively utilize broadcast, parallel-prefix and regular-routing operations. The algorithm is employed as an auxiliary routine for the randomized selection algorithm presented later in this work.

Informal Description of Algorithm The randomized sorting algorithm is an immediate iterative adaptation of the sorting algorithm of section 2.6.5 – with the utilization of the over-sampling ideas for the actualization of the iterative process.

The main result of this section is summarized in the following theorem.

²³The $o(n/p)$ term in the communication time requirements of [27] is asymptotically larger than the corresponding term in our algorithm.

Theorem 2.8.1 *For any constant $\rho > 0$ and $0 < \epsilon < 1$, there exists a randomized p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p such that $p = \Omega(n^{1+\epsilon})$, requires time [with probability $1 - O(n^{-\rho})$] $O(1/\epsilon \cdot \lg p)$ and $L \cdot O(1/\epsilon \cdot \lg p) + g \cdot O(1/\epsilon \cdot \lg p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.8.1 follows easily as a consequence of these results.

The general formulation of our algorithm is as follows. Initially, the p processors select a sample of size $\sqrt{p}$ and sort this sample by performing all possible $(\sqrt{p} \cdot (\sqrt{p} - 1))/2$ comparisons in constant time by utilizing all p processors. A parallel-prefix sum operation on the results of the comparisons determines the rank of each item in the final sorted sequence (for a detailed discussion of this method refer to section 2.6.5). The time of this sorting step is thus dominated by that of the parallel-prefix operation. Subsequently, a subset of the sample consisting of $k - 1 = o(\sqrt{p})$ items (splitters) is chosen. The input items are then partitioned into k buckets (as induced by the $k - 1$ splitters) by a search procedure that exclusively utilizes parallel-prefix and regular-routing operations. The k buckets are then sorted recursively by allocating the same number of processors to each bucket. The recursion terminates when the size of each bucket is at most the square root of the number of processors allocated to it; then, the algorithm previously employed for sample sorting is used to sort the items within each bucket.

In the following paragraphs we sketch a deterministic BSP algorithm for the following *searching* problem.

Problem Given a multi-set of n items $X = \langle x_1, x_2, \dots, x_n \rangle$ and a sorted sequence Y of k items, determine, for each $x \in X$, its rank in Y .

The following result suggests an efficient BSP algorithm for solving the searching problem with $p = n \cdot k^{1/h}$ processor, for some positive integer $h > 0$.

Proposition 2.8.1 *There exists a p -processor BSP algorithm for solving the searching problem on a multi-set of size n and a sorted sequence of size k , that for all integers n, k, p and m , such that $p = n \cdot k^{1/m}$ and $m \geq 1$, requires time*

$$C_{srch}(n, k, m, p) = m \cdot (O(1) + C_{ppf}(p))$$

and

$$M_{srch}(n, k, m, p) = m \cdot (L \cdot O(1) + g \cdot O(1) + 3 \cdot C_{bcast}(p) + C_{ppf}(p)),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{srch}(n, k, m, p) = C_{srch}(n, k, m, p) + M_{srch}(n, k, m, p)$.

Proof The algorithm to manage the search process employs a $k^{1/m}$ -ary balanced search tree on the sorted sequence of size k and derives from the ideas of section 2.6.5. The basic algorithmic idea is fairly simple; however, its technical details are rather complicated and omitted in this work. A full detailed account of the algorithm and its analysis can be found in [78]. $\square$

We now present the randomized BSP sorting algorithm for the problem of sorting n items with $p = n^{1+\epsilon}$ processors, for any constant $0 \leq \epsilon < 1$.

Proposition 2.8.2 *For any constant $\rho > 0$ and $0 < \epsilon < 1$, there exists a randomized p -processor BSP algorithm for sorting a multi-set of size n , that for all integers n and p , such that $p \geq n^{1+\epsilon}$, requires time [with probability $1 - O(n^{-\rho})$] –*

$$C_{poly_srt}(n, p) = O\left(\frac{1}{\epsilon}\right) \cdot \left(O(1) + C_{quad_srt}(n^{(1+\epsilon)/2}, p) + C_{srch}\left(n, n, \left\lceil \frac{2}{1+\epsilon} \right\rceil, p\right) + C_{ppf}(p)\right)$$

and

$$M_{poly_srt}(n, p) = O\left(\frac{1}{\epsilon}\right) \cdot \left(L \cdot O(1) + g \cdot O(1) + M_{quad_srt}(n^{(1+\epsilon)/2}, p) + M_{srch}\left(n, n, \left\lceil \frac{2}{1+\epsilon} \right\rceil, p\right) + M_{ppf}(p)\right)$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{poly_srt}(n, p) = C_{poly_srt}(n, p) + M_{poly_srt}(n, p)$.

Proof The algorithm to manage the sorting process employs the sorting and searching algorithms implied in propositions 2.6.2 and 2.8.1 respectively and is based on the random sampling techniques of section A.2.1. The basic algorithmic idea is fairly simple; however, its technical details are rather complicated and omitted in this work. A full detailed account of the algorithm and its analysis can be found in [78]. $\square$

Proof (of theorem 2.8.1) The theorem follows easily from proposition 2.8.2 by substituting for T_{quad_srt} , T_{srch} and T_{ppf} the corresponding expression in propositions 2.6.2, 2.8.1 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.8.5 An Optimal Randomized Selection Algorithm

In this section, we provide an optimal randomized BSP solution to the problem of selecting the C -th smallest item from a randomly distributed multi-set X that is based on *over-sampling* techniques [75, 183] – the required notation and concepts related to random over-sampling are presented in appendix A.2.2. The basic algorithmic idea derives from a selection algorithm of [70]. The algorithm achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal randomized selection algorithm is based on sampling techniques and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, each processor constructs (in a randomized way) a sub-sequence that is an appropriate sample of its input. Subsequently, the so constructed sub-sequences are united and sorted, by means of either the sorting algorithm of section 2.8.4 (for small sample sizes) or the optimal sorting algorithm of section 2.6.8.2 (for large sample sizes), and two appropriately chosen items are selected from the resulting sorted sequence and broadcast to all processors. The input items within these two items are selected and sorted, by means of either the sorting algorithm of section 2.8.4 (for small input sizes) or the optimal sorting algorithm of section 2.6.8.2 (for large input sizes). Subsequently, the sum and prefix sum of the number of input items within the two selected items is computed, by means of a parallel-prefix scheme, and communicated to

each processor. Finally, the required item is selected (the aforementioned selection operation is realized by employing the sum and prefix sum of the number of input items within the two selected items).

Our main result is summarized in the following theorem.

Theorem 2.8.2 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for determining the C -th smallest item of a multi-set of size n that for all integers n , C and p , such that $p = O(n)$ and $C \leq \lceil n/2 \rceil$, requires time [with probability $1 - O(n^{-\rho})$] $c_* \cdot (n + C)/p + O(\sqrt{(n/p) \cdot \lg n}) + (n^{2/3}/p) \cdot \lg^{4/3} n + \lg n$ and $L \cdot O(\lg p) + g \cdot O((n^{2/3}/p) \cdot \lg^{1/3} n + \lg n)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number technical results. Then, theorem 2.8.2 follows easily as a consequence of these results.

Optimality Claim 2.8.1 *Theorem 2.8.2 implies that there exists a p -processor BSP algorithm for determining the C -th smallest item of a multi-set of size n , that for all integers n , C , p , L and g such that $p = o(n/\lg n)$, $L = o(n/(p \cdot \lg p))$ and $g = o((n/\lg n)^{1/3} + n/(p \cdot \lg n))$, has $\pi = 1 + o(1)$ and $\mu = o(1)$.*

The general formulation of the algorithm is as follows. A random sample of the input is initially chosen; the iterative version of the algorithm, which improves the lower order terms of the non-iterative version, proceeds in phases (iterations) so that in each phase a new sample is selected that is a superset of the sample selected in the previous phase. Subsequently, the sample is sorted and two items within this sample are selected so that the C -th smallest item is straddled by these items. The so identified items are broadcast to all processors and the set of items between them is determined. Finally, by sorting this set the required statistic is trivially determined and broadcast to all processors. The procedure to manage the aforementioned scheme is presented in the following result.

Proposition 2.8.3 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for determining the C -th smallest item of a multi-set of size n that for all integers n , C and p , such that $p = O(n)$ and $C \leq \lceil n/2 \rceil$, requires time [with probability $1 - O(n^{-\rho})$]*

$$C_{sel}(n, C, p) = c_* \cdot \frac{n + C}{p} + O\left(\sqrt{\frac{C \cdot \lg n}{p}} + \frac{n^{\frac{2}{3}} \cdot \lg^{\frac{4}{3}} n}{p} + \lg n\right) + 2 \cdot T_{ppf}(p)$$

and

$$M_{sel}(n, C, p) = L \cdot O(\lg p) + g \cdot O\left(\frac{n^{\frac{2}{3}} \cdot \lg^{\frac{1}{3}} n}{p} + \lg n\right) + 3 \cdot M_{bcast}(p) + 2 \cdot M_{ppf}(p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{sel}(n, C, p) = C_{sel}(n, C, p) + M_{sel}(n, C, p)$.

Proof We assume that the input set is evenly and uniformly at random distributed among the p processors. If the input is not initially randomly distributed, then a pre-randomization phase is required for

optimal time bounds to be derived.²⁴

Procedure SELECT

begin

[**Stage 1**] A sample of size $2 \cdot \lambda - 1 = 2 \cdot n^{2/3} \cdot (3 \cdot \rho \cdot \log n)^{1/3} \cdot ((\sqrt{2 \cdot c} + \sqrt{2 \cdot (1 - c)})/2)^{2/3} - 1$ is selected, where $c = C/n$. The sample selection method is as follows.

- (1) Each processor selects approximately the same number of processor identifiers $\lceil (2 \cdot \lambda - 1)/p \rceil$ (or $\lfloor (2 \cdot \lambda - 1)/p \rfloor$) uniformly at random; a sample of size $2 \cdot \lambda - 1$, over all processors, is selected.
- (2) The processor identifiers are routed to the corresponding processors.
- (3) Each processor p_i , $0 \leq i < p$, that receives t_i identifiers selects among its items uniformly at random without replacement t_i items.
- (4) A parallel-prefix sum on the number t_i of items selected per processor is realized.
- (5) The result of the parallel-prefix sum is used to evenly distribute the selected sample among the processors.

[**Analysis of Stage 1**] Step 1 requires time $O(\lambda/p)$ for random processor identifier generation. We distinguish three cases in the analysis of step 2.

Case 1 If $\lceil (2 \cdot \lambda - 1)/p \rceil = \Omega(\lg n)$, then the inequality in theorem A.1.3 implies that each processor receives $O(\lambda/p)$ processor identifiers, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Case 2 If $\lceil (2 \cdot \lambda - 1)/p \rceil = O(\lg n)$, then the inequality in theorem A.1.2 implies that each processor receives $O(\lg n)$ processor identifiers, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Case 3 If $\lceil (2 \cdot \lambda - 1)/p \rceil = O(n^{-\zeta})$, for any constant $0 < \zeta < 1/3$, then the inequality in theorem A.1.2 implies that each processor receives $O(1)$ processor identifiers, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Cases 2 and 3 overlap for certain ranges of $\lceil 2\lambda/p \rceil$; case 3 is applied to such cases. Thus, this step requires communication time $L + g \cdot O(\max\{\lambda/p, \lg n\})$ if $p \leq n^{2/3+\zeta}$ and $L + g \cdot O(1)$ otherwise. Similarly, we conclude that step 3 requires time $O(\max\{\lambda/p, \lg n\})$ if $p \leq n^{2/3+\zeta}$ and $O(1)$ otherwise. Step 4 requires time $T_{ppf}(p)$ and step 5 can be shown (along the lines of step 2) to require communication time $L + g \cdot O(\max\{\lambda/p, \lg n\})$ if $p \leq n^{2/3+\zeta}$ and $L + g \cdot O(1)$ otherwise.

[**Stage 2**] The sample of size $\lceil 2 \cdot \lambda/p \rceil$ is sorted. Let the result of the sorting operation be sequence $Y = \langle y_1, y_2, \dots, y_{2 \cdot \lambda - 1} \rangle$.

[**Analysis of Stage 2**] This stage is realized by one of two sorting methods depending on the size of the sample. We distinguish two cases for the analysis of this stage.

²⁴For a random distribution and given $p = o(n/\lg n)$, the BSP algorithm requires, with high probability, $c_* \cdot (n + C)/p + o(n/p)$ and $g \cdot o(n/p)$ computation and communication time respectively. For a non-random distribution, the algorithm requires, with high probability, $2 \cdot (n/p) + o(n/p)$ and $g \cdot O(n/p)$ computation and communication time respectively. In the latter case, the algorithm is work-optimal in computation in the sense that the total computation time over all processors is still $c_* \cdot (n + C + o(n))$.

Case 1 If $p = \lambda^{1+\epsilon}$, for some constant $0 < \epsilon < 1$, then the algorithm implied in theorem 2.8.1 is employed. The time required for this case is therefore bounded above by $O(\lg \lambda)$ and $L \cdot O(\lg p) + g \cdot O(\lg p)$, for computation and communication respectively, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Case 2 Otherwise, the algorithm implied in theorem 2.6.7 is employed. The time required for this case is therefore bounded above by $O((\lambda/p') \cdot \lg \lambda)$ and $L \cdot O(\lg \lambda / \lg(\lambda/p')) + g \cdot O((\lambda/p') \cdot (\lg \lambda / \lg(\lambda/p')))$, for computation and communication respectively, where $p' = \min\{p, \lambda/2\}$.

[Stage 3] Let $\mu_0 = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot c \cdot \lambda \cdot \log n}$ and $\mu_2 = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1-c) \cdot \lambda \cdot \log n}$. If $c > 2 \cdot \rho \cdot \log n / \lambda$, then let $m = y_{2c\lambda - \mu_0/2}$ and $M = y_{2c\lambda + \mu_2/2}$; otherwise, let $m = -\infty$ and $M = y_{2c\lambda + \mu_2/2}$. Pivot items m and M are broadcast to all processors.

[Analysis of Stage 3] This stage takes time $O(1)$ and $T_{bcast}(p)$ for computation and communication respectively.

[Stage 4] Each processor p_i , $0 \leq i < p$, sets $l_i = r_i = 0$ and $R_i = \{\}$. Then, processor p_i , for each item x in its possession, compares x to M . If x is larger than M , then $r_i = r_i + 1$. If x is smaller than M , then x is compared to m . If x is smaller than m , then $l_i = l_i + 1$. Otherwise, $R_i = R_i \cup \{x\}$.

[Analysis of Stage 4] By way of claim A.2.4, at least $n - C - n \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1-c) \cdot \log n} / \sqrt{\lambda}$ items have value larger than M , with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$. Therefore, the number of items, over all processors, which are smaller than M is at most $(C + n \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1-c) \cdot \log n} / \sqrt{\lambda})$. Since these items are uniformly at random distributed among the processors, we distinguish cases similar to those of stage 1 to determine the number of such items residing in the local memory of each processor. We therefore obtain that each processor has in its local memory at most $C/p + O(\sqrt{C \cdot \lg n/p} + \lg n) + O(\max\{\lambda/p, \lg n\})$ such items if $p \leq n^{2/3+\zeta}$ and $C/p + O(\sqrt{C \cdot \lg n/p} + \lg n) + O(1)$ otherwise, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$. Thus, this step requires time $c_* \cdot (n + C)/p + O(\sqrt{C \cdot \lg n/p} + \lg n) + O(\max\{\lambda/p, \lg n\})$ if $p \leq n^{2/3+\zeta}$ and $c_* \cdot (n + C)/p + O(\sqrt{C \cdot \lg n/p} + \lg n) + O(1)$ otherwise.

[Stage 5] A parallel-prefix sum operation is realized on the l_i 's and r_i 's, $0 \leq i < p$.

[Analysis of Stage 5] This stage requires time $T_{ppf}(p)$.

[Stage 6] A broadcast operation is realized on $\sum_i l_i$ and $\sum_i r_i$, $0 \leq i < p$.

[Analysis of Stage 6] This stage takes time $T_{bcast}(p)$.

[Stage 7] Set $R = \cup_i R_i$, for all $0 \leq i < p$, is sorted and the $(C - \sum_i l_i)$ -th smallest item of R is broadcast to all processors.

[Analysis of Stage 7]

Case 1 If $c \geq 3 \cdot \rho \cdot \log n / \lambda$, then, for $n = N$, by way of claims A.2.1 and A.2.3, at least $C - n \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot c \cdot \log n} / \sqrt{\lambda}$ items and at most $C - 1$ items have value less than m , with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$. Similarly, by way of claims A.2.2 and A.2.4, at least $n - C - n \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1-c) \cdot \log n} / \sqrt{\lambda}$ and at most $n - C - 1$ items have value larger than M , with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Case 2 If $c < 3 \cdot \rho \cdot \log n / \lambda$, then, by way of claim A.2.4, at least $n - C - n \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1 - c) \cdot \log n / \lambda}$ items have value larger than M , with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

Therefore, the number of input items between m and M , or equivalently, the size of R is at most $2 \cdot \lambda$, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$. The results of stages 5 and 6 are then used to evenly distribute set R among the processors. Along the lines of the analysis of step 2 of stage 1, this distribution operation requires communication time $L + g \cdot O(\max \{\lambda/p, \lg n\})$ if $p \leq n^{2/3+\zeta}$ and $L + g \cdot O(1)$ otherwise. Finally, the broadcast operation requires time $T_{bcast}(p)$.

end

Analysis The correctness of the algorithm follows from its description and is omitted. By summing up the computation and communication requirements of procedure SELECT we obtain the desired time bound, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$.

The proposition follows. $\square$

Remark We note that if $C > \lceil n/2 \rceil$, by reversing the meaning of the inequalities in procedure SELECT, we can employ the same procedure to determine the $(n - C)$ -th largest item of the input set, which is the C -th smallest item of the input set.

Proof (of theorem 2.8.2) The theorem follows easily from proposition 2.8.3 by substituting for T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

In the data-structure algorithms we describe in later sections, it is sometimes desirable that h independent and disjoint selection operations are realized simultaneously. Each such selection operates on disjoint data sets of size n and the same statistic is involved in all operations. The probabilistic arguments involved in the analysis of the procedure implied in proposition 2.8.3 can be extended, due to the independence of the involved sub-problems, to handle simultaneously all h selection operations. This observation can be exploited to decrease the amount of parallel slackness required to achieve optimality by as much as h . Moreover, since the operations involved in the selection process include broadcast and parallel-prefix operations the selection process can be further improved by pipelining these operations. Hence, it is straightforward to verify the following analogue of proposition 2.8.3.

Proposition 2.8.4 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for determining the C -th smallest item of h multi-sets each of size n that for all integers n , C and p , such that $p = O(n)$ and $C \leq \lceil n/2 \rceil$, requires time [with probability $1 - O(n^{-\rho})$]*

$$C_{sel}^h(n, C, p) = c_* \cdot h \cdot \frac{n + C}{p} + O \left(h \cdot \sqrt{\frac{C \cdot \lg n}{h \cdot p}} + h \cdot \frac{n^{\frac{2}{3}} \cdot \lg^{\frac{4}{3}} n}{p} + \lg n \right) + 2 \cdot T_{ppf}^h(p)$$

and

$$M_{sel}^h(n, C, p) = L \cdot O(\lg p) + g \cdot O \left(h \cdot \frac{n^{\frac{2}{3}} \cdot \lg^{\frac{1}{3}} n}{p} + \lg n \right) + 3 \cdot M_{bcast}^{O(h)}(p) + 2 \cdot M_{ppf}^h(p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{sel}^h(n, C, p) = C_{sel}^h(n, C, p) + M_{sel}^h(n, C, p)$.

Proof The proof follows that of proposition 2.8.3 and is omitted. $\square$

2.8.5.1 Improving Randomized Selection

In order to improve the lower order terms in proposition 2.8.3 an iterative version of the selection procedure of the previous section has also been designed. The underlying algorithm is based on a bootstrapping technique of [70].

Informal Description of Algorithm The randomized selection algorithm is an immediate iterative adaptation of the selection algorithm of section 2.8.5 – with the utilization of the bootstrapping ideas of [70] for the actualization of the iterative process.

Our main result is summarized in the following theorem.

Theorem 2.8.3 *For any constant $\rho > 0$ and arbitrarily small constant $0 < \zeta < 1/2$, there exists a randomized p -processor BSP algorithm for determining the C -th smallest item of a multi-set of size n that for all integers n , C and p , such that $p = O(n)$ and $C \leq \lceil n/2 \rceil$, requires time [with probability $1 - O(n^{-\rho})$] $c_* \cdot (n + C)/p + O(\sqrt{(n/p) \cdot \lg n}) + (n^{1/2+\zeta}/p) \cdot \lg^{3/2} n + \lg n$ and $L \cdot O(\lg p) + g \cdot O((n^{1/2+\zeta}/p) \cdot \lg^{1/2} n + \lg n)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the following result. Then, theorem 2.8.3 follows easily as a consequence of this result.

Optimality Claim 2.8.2 *Theorem 2.8.3 implies that there exists a p -processor BSP algorithm for determining the C -th smallest item of a multi-set of size n , that for all integers n , C , p , L and g such that $p = o(n/\lg n)$, $L = o(n/(p \cdot \lg p))$ and $g = o((n/\lg^{-\zeta} n)^{1/2+\zeta} + n/(p \cdot \lg n))$, has $\pi = 1 + o(1)$ and $\mu = o(1)$.*

Proposition 2.8.5 *For any constant $\rho > 0$ and arbitrarily small constant $0 < \zeta < 1/2$, there exists a randomized p -processor BSP algorithm for determining the C -th smallest item of a multi-set of size n that for all integers n , C and p , such that $p \leq n$ and $C \leq \lceil n/2 \rceil$, requires time – with probability $1 - O(n^{-\rho})$ –*

$$C_{sel}(n, C, p) = c_* \cdot \frac{n + C}{p} + O\left(\sqrt{\frac{C \cdot \lg n}{p}} + \frac{n^{\frac{1}{2}+\zeta} \cdot \lg^{\frac{3}{2}} n}{p} + \lg n\right) + O\left(\frac{1}{\zeta}\right) \cdot (2 \cdot T_{ppf}(p))$$

and

$$M_{sel}(n, C, p) = L \cdot O\left(\frac{1}{\zeta} \cdot \lg p\right) + g \cdot O\left(\frac{n^{\frac{1}{2}+\zeta} \cdot \lg^{\frac{1}{2}} n}{p} + \lg n\right) + O\left(\frac{1}{\zeta}\right) \cdot (3 \cdot M_{bcast}(p) + 2 \cdot M_{ppf}(p)),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{sel}(n, C, p) = C_{sel}(n, C, p) + M_{sel}(n, C, p)$.

Proof The algorithm to manage the selection process employs a scheme similar to that of the selection algorithm implied in proposition 2.8.3 and is based on a bootstrapping technique of [70]. The basic algorithmic idea is fairly simple; however, its technical details are rather complicated and omitted in this work. A full detailed account of the algorithm and its analysis can be found in [78]. $\square$

Proof (of theorem 2.8.3) The theorem follows easily from proposition 2.8.5 by substituting for T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions.

$\square$

2.8.5.2 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the selection algorithm (SELECT) of the previous section.

Implementation The prototype adopted for the *BSPlib* selection procedure is outlined in figure 2.50.

```
void BSP_select (byte *src, int *trg, int n, int m, int sb, int (*cmp)()) ;
    src  - input array.
    trg  - rank of required statistic.
    n    - input size.
    m    - required statistic.
    sb   - item size.
    cmp  - generic compare function.
```

Figure 2.50: Procedure for BSP selection.

The aim of function (procedure) `BSP_select` is to provide a data-type independent high-performance *BSPlib* selection algorithm implementation.²⁵

Experimental Results The theoretical performance of the algorithm is summarized in theorem 2.8.2. To determine the practical significance of the aforementioned result we benchmarked our implementation (`BSP_select`) on two different machines with a varying number of processors and input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated). Figures 2.51 and 2.53 present the execution time of procedure `BSP_select` on the two machines. Similarly, figures 2.52 and 2.54 expose the efficiency of the corresponding timing results.

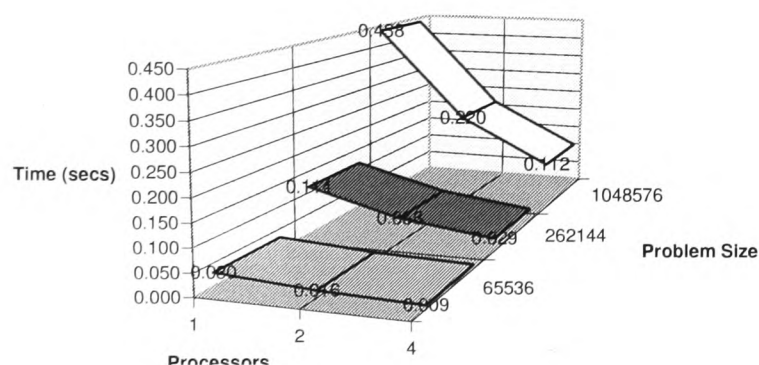


Figure 2.51: Execution time on a SGI Power Challenge for randomized selection (statistic $\lceil n/2 \rceil$) – uniform distribution (U).

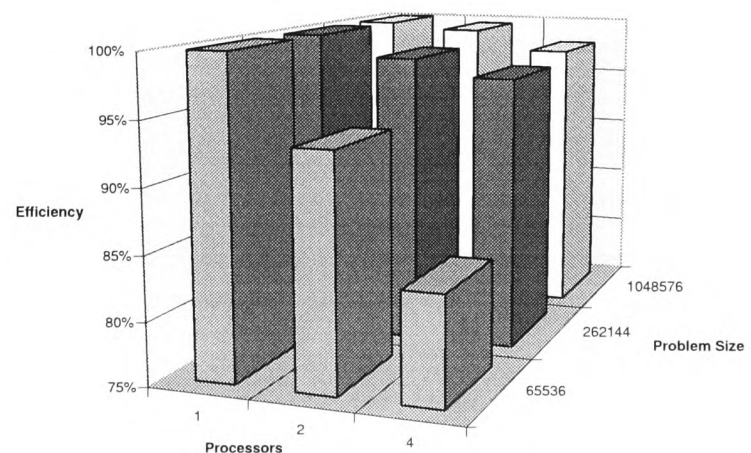


Figure 2.52: Efficiency on a SGI Power Challenge for randomized selection (statistic $\lceil n/2 \rceil$) – uniform distribution (U).

As shown in figures 2.51 and 2.53, the execution time of procedure `BSP_select` using a fixed number of processors scales with the input size (approximately as n). Accordingly, the execution time of procedure `BSP_select` for a fixed input size scales inversely with the number of processors (approximately as $1/p$).

²⁵The terms generic and data-type independent imply that procedure `BSP_select` can be employed to handle *any* data type. To this end, a comparison function `cmp` is used to facilitate the comparison of two elements e_1 and e_2 . In particular, the `cmp` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2 .

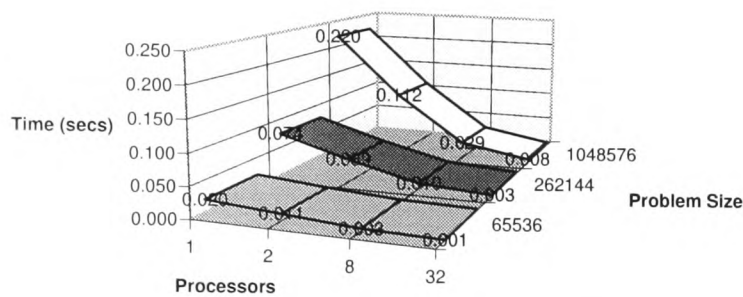


Figure 2.53: Execution time on a Cray T3D for randomized selection (statistic $\lceil n/2 \rceil$) – uniform distribution (U).

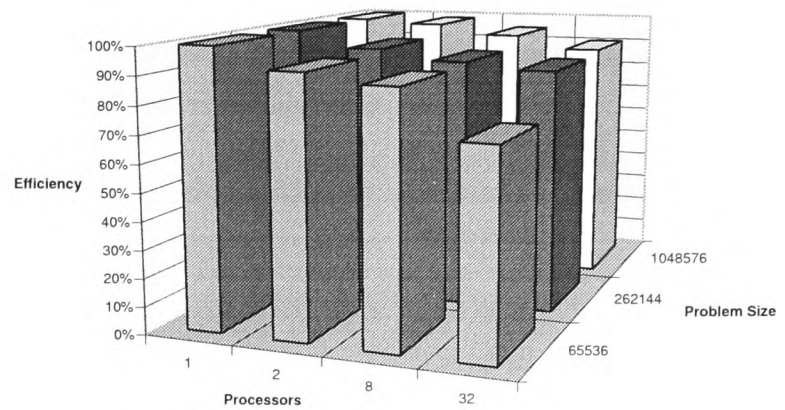


Figure 2.54: Efficiency on a Cray T3D for randomized selection (statistic $\lceil n/2 \rceil$) – uniform distribution (U).

Both experimental results are in accordance with the theoretical analysis of section 2.8.5. In terms of efficiency, procedure `BSP_select` utilizes approximately (within a 5%):

- 80%–90% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 95%–100% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),
- 60%–85% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 90%–100% of the available processor resources on the Cray T3D machine (for large problem sizes).

These results should be interpreted with caution on a twofold basis: (i) the random choices of the parallel and sequential selection algorithms might be different, and therefore, small inconsistencies in timing results are possible, and (ii) the timing results pertain only to the uniform distribution (we note that for non-uniform distributions the selection algorithm executes 1.3–1.6 times slower than reported in figures 2.51 and 2.53 – in accordance with the theoretical analysis – this slowdown is attributed to the fact that for non-uniform distributions the selection problem is communication bound, and therefore, efficiency close to one is not possible without the introduction of artificial conditions and premises).

2.9 Searching

In this section we design, analyze and implement BSP algorithms for a fundamental combinatorial operation, namely *searching*. In fact, we study a generalization of the searching operation [133], namely *multi-way searching* on complex data structures.²⁶ Parallel searching is a frequently recurring oper-

²⁶The implementation and utilization of data structures on distributed memory models such as the BSP rather than shared memory ones like the PRAM offers a serious challenge.

The problem of realizing such distributed data structures has been addressed in the past and several results have been obtained for particular network architectures (e.g, hyper-cubes, butterflies and meshes [17, 57, 178]). Unfortunately, architecture dependent features involved in these implementations, make the migration of these results to the more general setting of the BSP model non-trivial, and, in general, inefficient. In this work, we study communication efficient BSP methods for multi-way searching on a broad class of data structures that generalize, extend and improve previous approaches in several ways, as described in later sections.

ation in computer science and is fundamental for many parallel applications, such as implementing database primitives and parallel dictionaries, and improving the performance of combinatorial algorithms [25, 26, 54, 76, 80, 133]. The material of this section is based on the following papers:

- (1) “Communication efficient data structures on the BSP model with applications.” Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, May 1996 [76].
- (2) “Communication efficient data structures on the BSP model with applications in computational geometry.” In *Proceedings of Euro-Par’96 Parallel Processing, Lecture Notes in Computer Science* 1124, Springer-Verlag, August 1996 [80].
- (3) “Ordered h -level graphs on the BSP model.” Submitted for journal publication, 1997 [93].

2.9.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a non-decreasing sequence whose items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments. We consider, in this section, the problem of *multi-way searching* for the items in sequence X into complex graph structures – in particular, *ordered h -level graphs*. A detailed account of the definitions and notation pertaining ordered h -level graphs is diverged to section 2.9.4.1.

2.9.2 Cost Model

The performance of the BSP multi-way searching algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations. Furthermore, for some constant c_{srch} , time $c_{srch} \cdot n \cdot \lg m$, $O(n)$, and $o(n)$ is charged respectively for searching (for a single item) in a d -ary tree of size m , for integer-sorting a sequence (in the range $[0, 1, \dots, n^{O(1)}]$) of size n , and for sorting a sequence of size $\lceil \sqrt{n} \rceil$ – within the context of this section we denote c_{srch} by c_* .

Remark In the remainder of this work we refer to the aforementioned linear-time integer-sorting and sub-quadratic sorting algorithms as SEQ_RADIX_SORT and SEQ_SUBQUAD_SORT respectively. Moreover, we refer to the parallel version of algorithm SEQ_RADIX_SORT as RADIX_SORT (refer to section 2.4.4 for a detailed discussion of the parallel integer-sorting algorithm).

Optimal $\Theta(n \cdot \lg m)$ time sequential multi-way searching (on d -ary trees of size m) algorithms are well-known [133, 54]. In the following sections, we design and analyze optimal BSP multi-way searching (on a much wider class than d -ary trees) algorithms – optimality is realized for an ample range of the BSP parameters.

2.9.3 Related Work and New Results

There has been a notable amount of previous work on parallel multi-way searching (e.g., [170] on the PRAM model, [24, 25] on the BSP* model, [57] on the hyper-cube model, [178, 185] on the butterfly model). The exhibit in table 2.8 summarizes the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality. To

simplify the exposition, multi-way searching is assumed to transpire on d -ary trees of appropriate degree and of size $m \leq n$.

Refs.	Computation	Communication	Constants	Implem.	Procs
[170]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot (\lg m + \lg p)\right)$	■■■	■■	$O(n)$
[57]	$O\left(\frac{n}{p} \cdot (\lg m \cdot \lg n)\right)$	$O\left(\frac{n}{p} \cdot (\lg m \cdot \lg n)\right)$	■■	■■	$O(n)$
[178]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot (\lg m + \lg p)\right)$	■■	■■	$O\left(\frac{n}{\lg^2 n \cdot \lg \lg n}\right)$
[185]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot (\lg m + \lg p)\right)$	■■	■■■	$O(n)$
[24]	$2 \cdot c_* \cdot \frac{n}{p} \cdot \lg m + o\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■■■	$O\left(\frac{n}{\lg^2 n}\right)$
[25]	$c_* \cdot \frac{n}{p} \cdot \lg m + o\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■■■	$O\left(\frac{n}{\lg^2 n}\right)$
Sec 2.9.6	$c_* \cdot \frac{n}{p} \cdot \lg m + o\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■■	$O\left(\frac{n}{\lg n}\right)$

Table 2.8: Performance comparison of parallel multi-way searching algorithms – the cyan color is associated with randomized approaches.

With reference to table 2.8 we note that the algorithms of [57] and of section 2.9.6 can be employed to realize multi-way searching for a much wider class of data structures than d -ary trees (as opposed to the algorithms of [24, 25, 170, 178, 185] that can only be employed for d -ary trees).

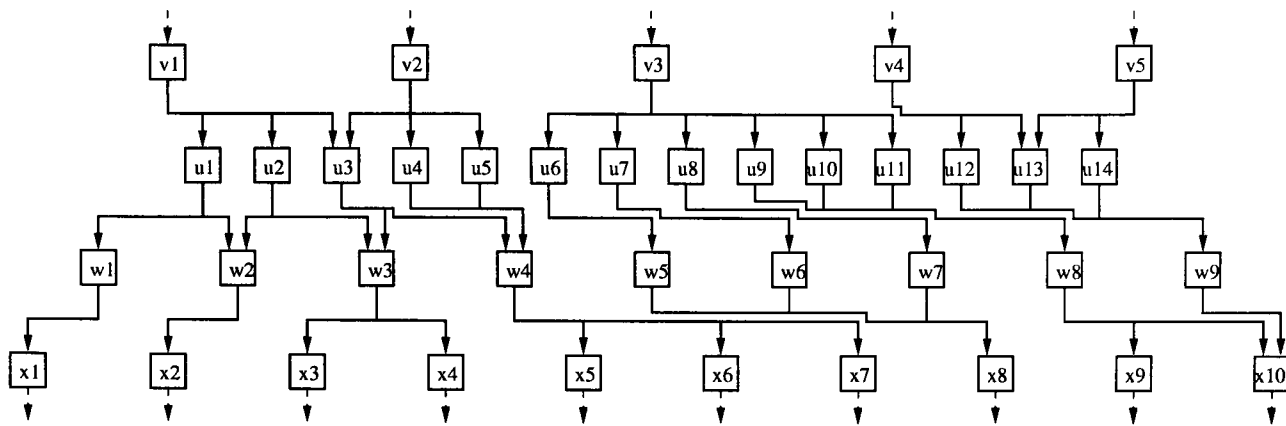
The optimal BSP multi-way searching algorithm of section 2.9.6 improves upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [57, 170, 178, 185] (if $m = n^{o(1)}$),
- (2) the asymptotic communication time requirements of [57, 170, 178, 185],
- (3) the multiplicative constant factor requirements of [24, 57, 170, 178, 185],
- (4) the implementation effort requirements of [24, 25, 185], and
- (5) the parallel slackness requirements (to achieve optimality) of [24, 25, 178].

The principal contribution, however, of the multi-way searching algorithm of section 2.9.6 is attributed to the *low communication cost* (refer to case 2), the *small constant multiplicative factors* (refer to case 3), and the *low amount of parallel slackness* required to achieve optimality (refer to case 5). Additionally, our techniques provide support for a much *wider class of data structures* than previous approaches [24, 25, 170, 178, 185] and *one-optimality* for an ample range of the BSP parameters.

2.9.4 Ordered h -level Graphs

In this section we study the computation and communication requirements of multi-way searching on a special class of graphs, namely *ordered h -level graphs* [57, 76, 80]. We propose multi-way searching as a general tool for the design, analysis and implementation of BSP algorithms. As noted in [57], multi-way searching allows elegant high-level design and analysis of several parallel techniques – using data structures similar to those of sequential models. In later sections, applications of multi-way searching (on ordered h -level graphs) to several fundamental computational problems are also presented.

Figure 2.55: An ordered h -level graph.

2.9.4.1 Preliminaries and Notation

In the remainder of this section, we formally define ordered h -level graphs and discuss some of their properties. The definitions and notation of this section derive from [57]. We extend, however, some of the notions introduced in [57] to include a more general class of ordered h -level graphs. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E . An *ordered h -level partitioning* of G is a partitioning of V into h disjoint sets $V_1, V_2, \dots, V_h$, together with an ordering of the vertices in each subset V_i , $1 \leq i \leq h$. For every h -level partitioning of G we define, for all $v \in V$, two parameters, namely $Level(v)$ and $LevelIndex(v)$ as follows.

- (1) $Level(v) = l$ if and only if $v \in V_l$, $1 \leq l \leq h$.
- (2) $LevelIndex(v) = \text{rank}(v : V_{Level(v)})$.

A directed graph $G = (V, E)$ is in the class of *ordered h -level graphs* if G has the following properties (refer to figure 2.55 for an illustration).

- (1) Every vertex v in V has out degree of at most d_{max} . In contrast to [57], we do not insist on constant degree graphs.
- (2) The number of sources d_s of G is at most d_{max} .
- (3) There exists an ordered h -level partitioning $V_1, V_2, \dots, V_h$, of G such that
 - every source of G is contained in V_1 ,
 - if (u, v) is an edge of G , then $Level(v) = Level(u) + 1$,
 - if (u, v) and $(\bar{u}, \bar{v})$ are two edges of G such that both $Level(u) = Level(\bar{u})$ and $LevelIndex(u) < LevelIndex(\bar{u})$, then $Level(v) = Level(\bar{v})$ and $LevelIndex(v) \leq LevelIndex(\bar{v})$.
- (4) The graph has no double edges.

As noticed in [57], any d -ary tree can be expressed as an ordered h -level graph. The following lemma captures many important facts about ordered h -level graphs.

Lemma 2.9.1 *Let $G = (V, E)$ be an ordered h -level graph. Then, the following statements hold.*

- (1) G is planar and acyclic.

$$(2) |E| \leq |V_1| + 2 \cdot |V_2 \cup \dots \cup V_{h-1}| + |V_h| - (h - 1).$$

Proof The planar and acyclic properties of G follow directly from the definition of ordered h -level graphs. In particular, the planarity of G derives from property 3 (case 3) of ordered h -level graphs, and similarly, the acyclic property of G derives from property 3 (case 2).

We use induction on the level number l of G to prove that $|E| \leq |V_1| + 2 \cdot |V_2 \cup \dots \cup V_{h-1}| + |V_h| - (h - 1)$. The claim is initially true, for when $l = 1$, E is empty, and therefore, $|E| = 0$.

Assume that the induction hypothesis holds up to level $l - 1$, $1 \leq l \leq h$. We next show that the claim holds for level l . We prove this by coloring the vertices of sets V_{l-1} , V_l and the edges between these two sets, and subsequently, by claiming that the number of colored edges is at most one less than the sum of the cardinalities of sets V_{l-1} and V_l . We assume that vertices within a set are colored from “left” to “right”, that is, from the vertex with the lowest *LevelIndex* to the vertex with the highest *LevelIndex*.

We use an inductive argument on the number of uncolored vertices in V_{l-1} . Let $u \in V_{l-1}$ be the vertex with the lowest *LevelIndex* among the yet uncolored vertices of V_{l-1} . If there is no edge $(u, v) \in E$, we color u and our claim holds, since we have colored one vertex and no edges. Otherwise let $v_{j+1}, v_{j+2}, \dots, v_{j+a} \in V_l$ be the vertices connected to u . By property 3, and the coloring so far, only the “rightmost” vertex v_{j+a} may have additional incoming edges from other uncolored vertices of V_{l-1} . Let the number of such additional edges be b . Denote by u_r the vertex with the highest *LevelIndex* among the vertices of V_{l-1} pointing to v_{j+a} . By property 3, only vertex u_r among the b vertices of V_{l-1} may be connected to other vertices in V_l . Let the out-degree of u_r be $c \geq 1$, and let $v_{j+a}, v_{j+a+1}, \dots, v_{j+a+c-1} \in V_l$ be the children of u_r . We color vertices $u, v_{j+1}, v_{j+2}, \dots, v_{j+a+c-2}$ (all but the “rightmost” children of u_r), and the b vertices of V_{l-1} connected to v_{j+a} (u_r included). We also color the a outgoing edges of u , the b incoming edges of v_{j+a} , and the remaining $c - 1$ outgoing edges of u_r . This process colors $a + b + c - 1$ vertices and $a + b + c - 1$ edges. If the “rightmost” child of u_r is not connected to some other yet uncolored vertex of V_{l-1} , then we color it and our claim holds, since we have colored $a + b + c$ vertices and $a + b + c - 1$ edges. We proceed as above until all vertices of V_{l-1} and V_l are colored. We can deduce that at the end of the coloring process, the number of colored vertices will be at least one more than the number of colored edges.

We thus conclude that the number of edges pointing to vertices in V_l is at most $|V_{l-1} \cup V_l| - 1$. By the induction hypothesis, the number of edges up to level $l - 1$ is at most $|V_1| + 2 \cdot |V_2 \cup \dots \cup V_{l-2}| + |V_{l-1}| - (l - 2)$, and thus, the number of edges up to level l is at most $|V_1| + 2 \cdot |V_2 \cup \dots \cup V_{l-1}| + |V_l| - (l - 1)$, and the lemma follows. $\square$

2.9.5 Multi-Way Searching on Ordered h -Level Graphs

Let $G = (V, E)$ be an ordered h -level graph of maximum out-degree d_{max} , with vertex set $V = V_1 \cup V_2 \cup \dots \cup V_h$ and edge set E , where $|V| = m$. Moreover, let U be a universe of possible search queries on G . A *search path* for a query $q \in U$ is a sequence $Path(q) = \langle v_1, v_2, \dots, v_h \rangle$ of h vertices of G , $v_i \in V_i$, $1 \leq i \leq h$, prescribed by a *next* function $f : U \times (V \cup \{start\}) \rightarrow \{1, 2, \dots, d_{max}\}$ as follows:

- $f(q, start) = Index(v_1)$,
- $f(q, v_i) = Index(v_{i+1})$, $1 \leq i \leq h - 1$,

where $(v_i, v_{i+1}) \in E$ and for every $v \in V$, $Index(v) = \sum_{i=1}^{Level(v)-1} |V_i| + LevelIndex(v)$. For example, if G is a d -ary search tree, then for every query q , $f(q, start)$ evaluates to the index of (or pointer to) the root

of the tree, and for every vertex v , $f(q, v)$ evaluates to the index of (or pointer to) the next vertex to be visited.

Given an ordered h -level graph G , we define the *search process* for a query q to be a process divided into h phases such that in phase l , $1 \leq l \leq h$, the search path $\langle v_1, v_2, \dots, v_l \rangle$ of q has been determined and there exists a processor which contains a description of both the query q and the vertex v_l . We do not assume that the search path is given in advance, only that it is constructed during the search process by successive applications of the *next* function f . We note that, in contrast to other methods (e.g., [24, 25, 170, 178, 185]), our method has the advantage of constructing the entire search path for each query q , and therefore, can be applied to the solution of a class of geometric problems such as those presented in [57]. The *multi-way search* problem is the following one.

Problem Given a set of n queries $Q = \{q_1, q_2, \dots, q_n\}$ and an ordered h -level graph G , determine, for each $q \in Q$, the search processes induced by q .

2.9.6 An Optimal Multi-Way Searching Algorithm on Ordered h -level Graphs

In this section we provide an optimal BSP solution to the multi-way searching problem on ordered h -level graphs that is based on a communication efficient *load-balancing* scheme. The algorithm achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic searching algorithm (on ordered h -level graphs) is based on pipelining techniques and is iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. In each iteration of the algorithm, processors obtain the required edge information (in order to advance the queries from one level of the graph to the next). Subsequently, processors determine, by utilizing the previously obtained edge information, the vertex destination of each query in their possession and re-group the queries with respect to their destination vertex. Finally, after a pre-specified number of iterations (equal to the number of levels in the graph) the iterative process results in all queries being at the last level of the graph.

Our main result is summarized in the following theorem.

Theorem 2.9.1 *Let $G = (V, E)$ be an ordered h -level graph, with minimum out-degree d_{min} and maximum out-degree d_{max} , where $|V| = m$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n , p , d_{min} and d_{max} , and any δ such that $p = o(n/\lg n)$, $d_{max} = o(n/p)$ and $\delta = O(\sqrt{d_{min}})$, requires time $\sum_{l=1}^h (\tau_l + O(1)) \cdot (n/p + 2 \cdot m/(\delta \cdot p) + o(n/p))$ and $L \cdot h \cdot O(\lg p) + g \cdot h \cdot O(\delta \cdot (n/p) + m/(\delta \cdot p))$, for computation and communication respectively, where τ_l is the time required to evaluate the next function at level l . The algorithm requires $O(\delta \cdot (n/p) + m/p)$ space per processor.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.9.1 follows easily as a consequence of these results.

Consider an ordered h -level graph $G = (V, E)$, of maximum out-degree d_{max} , with vertex set $V = V_1 \cup V_2 \cup \dots \cup V_h$ and edge set E , where $|V| = m$. Let U be a universe of possible search queries on G and let $f : U \times (V \cup \{start\}) \rightarrow \{1, 2, \dots, d_{max}\}$ be the *next* function. In the remainder of this section,

we consider a communication efficient algorithm for the solution of the multi-way searching problem for a set $Q = \{q_1, q_2, \dots, q_n\} \subseteq U$ of n queries.

The general formulation of our technique is as follows. The algorithm computes the n search processes for all n queries in a sequence of phases. The number of phases corresponds to the number of levels (height) of the underlying ordered h -level graph. At each phase l , $1 \leq l \leq h$, the queries advance through the graph along their search paths from level $l - 1$ to level l . We observe that the first phase is different from subsequent phases, since it has to manage an arbitrary permutation of the queries, whereas each subsequent phase may utilize the processing performed in the previous stage to improve the performance of the algorithm. At each phase l , for each query $q \in Q$, the algorithm consists of the following two stages.

Algorithm MWAY_SEARCH

begin

[Stage 1] Let v_{l-1} be the $(l - 1)$ -th vertex in the search path of q (if $l = 1$, then $v_0 = start$). Evaluate $f(q, v_{l-1})$.

[Stage 2] Obtain the next vertex v_l in the search path of q .

end

The algorithm terminates within h phases. Some additional notation, describing the allocation of the queries among the processors, is required before proceeding to the details of the algorithm. Let v be a vertex of V . We define the set of queries visiting v to be the *thread* at vertex v . We refer to this thread as a thread of level $Level(v)$. A thread might be distributed over a number of consecutive processors; such a thread is called *shared*, as opposed to *exclusive*. The processors that hold a shared thread form a *group*. The first processor of each group is called the *group leader*.

In order to achieve communication efficiency we maintain the following distribution (*load-balancing properties*) of the underlying ordered h -level graph G among the processors.

- (1) Each vertex $v \in V$, has out-degree of at most d_{max} , where $d_{max} = o(n/p)$. Additionally, each vertex v maintains the information pertaining to its children (*NextInfo*) in the form of edges.
- (2) Each edge $e \in E$, maintains (i) a pointer to the vertex v that e points to, (ii) the key associated with v , and (iii) the out-degree of v .
- (3) *start* is maintained as a vertex with children the sources of G .
- (4) G is *orderly* and *evenly* distributed among the processors; that is, all outgoing edges of a given vertex are stored in consecutive memory locations of a single processor, and no processor maintains more than $\lceil |E|/p \rceil + d_{max}$ edges.

The procedure to manage the multi-way search process, for an arbitrary phase l , is summarized in the following proposition.

Proposition 2.9.1 *Let $G = (V, E)$ be an ordered h -level graph, with minimum out-degree d_{min} and maximum out-degree d_{max} , where $|V| = m$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n , p , d_{min} and d_{max} , and any δ such that $p = O(n)$, $d_{max} = o(n/p)$ and $\delta = O(\sqrt{d_{min}})$, requires time*

$$C_{mway}(n, m, \delta, p) = \tau_l \cdot \left(\left\lceil \frac{n}{p} \right\rceil + \left\lceil \frac{2 \cdot m}{\delta \cdot p} \right\rceil + o\left(\frac{n}{p}\right) \right) + O\left(\frac{n}{p} + \frac{m}{\delta \cdot p}\right) + C_{rdx_srt}\left(n + \frac{2 \cdot m + o(n)}{\delta}, o\left(\frac{n}{p}\right), p\right) + C_{ppf}(p)$$

and

$$M_{mway}(n, m, \delta, p) = L \cdot O(1) + g \cdot O\left(\delta \cdot \frac{n}{p} + \frac{m}{\delta \cdot p}\right) + M_{rdx_srt}\left(n + \frac{2 \cdot m + o(n)}{\delta}, o\left(\frac{n}{p}\right), p\right) + M_{bcst}^{o(n/p)}(p) + M_{ppf}(p),$$

for computation and communication respectively, where τ_l is the time required to evaluate the next function at level l . The overall time complexity of the algorithm is denoted by $T_{mway}(n, m, \delta, p) = C_{mway}(n, m, \delta, p) + M_{mway}(n, m, \delta, p)$. The algorithm requires $O(\delta \cdot (n/p) + m/p)$ space per processor.

Proof Let $Q = \{q_1, q_2, \dots, q_n\}$ be the input set (of queries). We assume that Q is evenly but otherwise arbitrarily distributed among the processors. Additionally, we assume that the ordered h -level graph G satisfies load-balancing properties 1–4. As part of the input, for phase l , we also assume the following *invariants*.

- (1) The queries are sorted with respect to the *LevelIndex* of their associated vertex of level $l - 1$.
- (2) All shared threads at level $l - 1$ are identified and characterized by their group and group leader.

In the first phase there exists a single thread associated with vertex *start* of out-degree $d_s \leq d_{max}$. The procedure for level l , $1 \leq l \leq h$, ensures that, for each query $q \in Q$, with search path $\langle v_1, v_2, \dots, v_h \rangle$, there exists a processor which contains a description of both the query q and the vertex v_l . In the remainder of this proof, we use the term *NextInfo*, for a particular thread $\bar{Q}$, to describe the edges emanating from the vertex associated with $\bar{Q}$. Informally, *NextInfo* relates to the information required to evaluate the *next* function.

Procedure MWAY_SEARCH

begin

[**Stage 1**] Each processor obtains, for each thread in its possession, the associated *NextInfo* information.

[**Analysis of Stage 1**] Let $\bar{s}$ be the size of some thread $\bar{Q}$ formed in the previous phase or the single thread associated with vertex *start*, and let $\bar{d}$ be the out-degree of the vertex associated with $\bar{Q}$ (if $l = 1$, then $\bar{s} = n$). We note that $\bar{d}$ is known by load-balancing property 2.

This stage is realized by one of several methods depending on the size and distribution of $\bar{Q}$. Informally, the motivation behind realizing different methods for obtaining *NextInfo* is to minimize the communication requirements of the procedure. To this end, when the size of a thread is larger than the size of the associated *NextInfo* (cases 1–3 below), the latter information is read from the appropriate processor. Otherwise (case 4 below), the thread is sent to the processor maintaining the associated *NextInfo*. In particular, if $\bar{Q}$ resides on some processor $\bar{p}$, *NextInfo* resides on some processor $\hat{p}$ and $\delta = O(\sqrt{\bar{d}})$, processor $\bar{p}$ proceeds as follows:

- (1) read *NextInfo* if $\bar{s} \geq d_{max}/\delta$ and $\bar{Q}$ is exclusive.
- (2) read *NextInfo* if $\bar{s} \geq d_{max}/\delta$, $\bar{Q}$ is shared, and $\bar{p}$ is a group leader.
- (3) read *NextInfo* if $\bar{s} < d_{max}/\delta$ and $\bar{s} \geq \bar{d}/\delta$.
- (4) send $\bar{Q}$ to $\hat{p}$ if $\bar{s} < \bar{d}/\delta$.
- (5) remain idle otherwise.

When this stage is initiated each processor holds at most $\lceil n/p \rceil$ queries, and by way of lemma 2.9.1 and load-balancing property 4, maintains at most $\lceil 2 \cdot m/p \rceil + d_{max}$ edges. By way of cases 1–3 above, each processor reads information of size at most $\delta \cdot (n/p) + 2 \cdot d_{max}$, where the $2 \cdot d_{max}$ term is attributed to shared threads. Additionally, by way of case 4 above, each processor receives a set Q^* of queries, of size at most $s^* = (\lceil 2 \cdot m/p \rceil + d_{max})/\delta$. Hence, this step requires $L + g \cdot O(\delta \cdot (n/p) + (m/p + d_{max})/\delta)$ communication time.

If $\bar{Q}$ is shared and processor $\bar{p}$ is the group leader for $\bar{Q}$, then processor $\bar{p}$ broadcasts the information it read in the previous step to all the processors in its group. By proposition 2.1.2, this step requires communication time $M_{broadcast}^{O(d_{max})}(p)$. At the end of this stage each processor holds at most $\lceil n/p \rceil + s^*$ queries.

[Stage 2] Each processor computes, for each query $q \in Q$ in its possession, the *Index* of the next vertex v_l in the search path of q by evaluating the *next* function

$$\begin{aligned} f(q, start) & \quad \text{if } l = 1, \\ f(q, v_{l-1}) & \quad \text{otherwise.} \end{aligned}$$

[Analysis of Stage 2] When this stage is executed every processor holds at most $\lceil n/p \rceil + s^*$ queries, and therefore, the time required by this stage is bounded above by $\tau_l \cdot (\lceil n/p \rceil + s^*)$, where τ_l is the time required to evaluate the *next* function f at level l .

[Stage 3–5] If $l < h$, in stages 3–5, the necessary invariants for phase $l + 1$ are updated.

[Stage 3] The queries within each thread are sorted with respect to the *LevelIndex* of their associated vertex of the current level [giving invariant 1].

[Analysis of Stage 3] Let $\bar{s}$ be the size of some thread $\bar{Q}$ formed in the previous phase or the single thread at the beginning of the algorithm, and let $\bar{d}$ be the out-degree of the vertex associated with $\bar{Q}$ (if $l = 1$, then $\bar{s} = n$). The sorting operation is performed by one of several methods depending on the size and distribution of $\bar{Q}$. In particular, the algorithm employs

$$\begin{aligned} \text{SEQ_RADIX_SORT} & \quad \text{if } \bar{s} \geq \sqrt{\bar{d}/\delta} \text{ and } \bar{Q} \text{ is exclusive,} \\ \text{RADIX_SORT} & \quad \text{if } \bar{s} \geq \sqrt{\bar{d}/\delta} \text{ and } \bar{Q} \text{ is shared,} \\ \text{SEQ_SUBQUAD_SORT} & \quad \text{if } \bar{s} < \sqrt{\bar{d}/\delta}. \end{aligned}$$

In case 3, if $\bar{Q}$ is shared, then the lower-indexed of the two processors sharing $\bar{Q}$ performs procedure SEQ_SUBQUAD_SORT.

When this stage is executed the overall size of all threads of size greater than $\sqrt{\bar{d}/\delta}$ on any one processor is at most $\lceil n/p \rceil + s^*$. By way of algorithms SEQ_RADIX_SORT and RADIX_SORT (refer to section 2.9.2) we have that all such threads can be sorted in time $O(n/p) + C_{rdx_srt}(n + p \cdot s^*, d_{max}, p)$ and $M_{rdx_srt}(n + p \cdot s^*, d_{max}, p)$ for computation and communication respectively. The $O(n/p)$ term is attributed to sequential radix sort – each exclusive thread implies radix sorting of at least $\sqrt{\bar{d}/\delta}$ integers in the range $[0, 1, \dots, \bar{d} - 1]$, for a total time of $O(\lg \bar{d} / \lg \sqrt{\bar{d}/\delta} \cdot (n/p)) = O(n/p)$, since $\delta = O(\sqrt{\bar{d}})$.

By the definition of SEQ_SUBQUAD_SORT (refer to section 2.9.2), each thread of size less than $\sqrt{\bar{d}/\delta}$ can be sorted in $o(\bar{d}/\delta)$ time. The total time required to sort all such threads $\bar{Q}_j$ of size $\bar{s}_j < \sqrt{\bar{d}_j/\delta}$ on any particular processor is $o(\sum_j \bar{s}_j^2) \leq o(\sum_j \bar{d}_j/\delta)$, where $\bar{d}_j$ is the out-degree of the vertex associated with thread $\bar{Q}_j$. By lemma 2.9.1 and load-balancing property 4, $\sum_j \bar{d}_j \leq \lceil 2 \cdot m/p \rceil + d_{max}$, and therefore, $\sum_j \bar{s}_j^2 \leq (\lceil 2 \cdot m/p \rceil + d_{max})/\delta$. Thus, the total time required to sort all such threads is bounded above by $o((m/p + d_{max})/\delta)$ and $L + g \cdot o(\sqrt{d_{max}}/\delta)$ for computation and communication respectively.

[Stage 4] Each processor returns (to the original processor) the threads that it received in stage 1.

[Analysis of Stage 4] Since the total size of all the threads that any single processor sent or received in stage 1 is at most $\lceil n/p \rceil$ and s^* respectively, this stage takes time $L + g \cdot O(\max\{n/p, s^*\})$.

[Stage 5] The threads at level l are determined and characterized by group and group leader [giving invariant 2].

[Analysis of Stage 5] When this stage is executed, the queries are sorted with respect to their *LevelIndex*, and therefore, each processor can determine all of its exclusive threads by simply scanning the queries in its local memory. Moreover, if there are shared threads, all of them can be identified and characterized by communicating a constant number of messages and performing a parallel-prefix operation. In particular, the prefix operator takes two operands, namely $(shared_thread_i, processor_{i'})$ and $(shared_thread_j, processor_{j'})$, and returns (for $i < j$):

$$\begin{aligned} & (shared_thread_i, processor_{i'}) && \text{if } shared_thread_i = shared_thread_j, \\ & (shared_thread_j, processor_{j'}) && \text{if } shared_thread_i \neq shared_thread_j. \end{aligned}$$

After completion of the parallel prefix operation each processor knows for each of its shared threads its group and group leader – the group leader of some shared thread i is determined by the second argument of the pair returned by the parallel-prefix operation. When this stage is initiated every processor holds at most $\lceil n/p \rceil$ queries and at most two shared threads. Thus, the cost of this stage is bounded above by $O(n/p) + C_{ppf}(p)$ and $L + g \cdot O(1) + M_{ppf}(p)$, for computation and communication respectively.

end

Analysis The correctness of the algorithm follows from its description. By summing up the computation and communication requirements of procedure MWAY_SEARCH we obtain the desired time bound.

The proposition follows. $\square$

Proof (of theorem 2.9.1) The theorem follows easily from proposition 2.9.1 by substituting for T_{rdx_srt} , $T_{broadcast}$ and T_{ppf} the corresponding expressions in propositions 2.4.1, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.9.6.1 The Case $m \leq n$ through Load-Balancing

In the description of procedure MWAY_SEARCH of section 2.9.6, we employed the parameter δ to express a trade-off between computation and communication time. In particular, we obtained a $2 \cdot m/(\delta \cdot p)$ term in the computation requirements of the algorithm at the expense of a $\delta \cdot (n/p)$ term in the communication and space requirements of the algorithm. In the remainder of this section we outline an algorithm that disposes of the aforementioned terms by employing a load-balancing scheme.

Informal Description of Algorithm The optimal deterministic searching algorithm (on ordered h -level graphs) is an immediate adaptation of the searching algorithm of section 2.9.6 – with the exception that load-balancing techniques are utilized to improve and generalize the algorithm.

Our main result is summarized in the following theorem.

Theorem 2.9.2 *Let $G = (V, E)$ be an ordered h -level graph, with maximum out-degree d_{max} , where $|V| = m$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n , m , p and d_{max} such that $m \leq n$, $p = o(n/\lg n)$ and $d_{max} = o(n/p)$, requires time $\sum_l (\tau_l + O(1)) \cdot (n/p)$ and $L \cdot h \cdot O(\lg p) + g \cdot h \cdot O(n/p)$, for computation and communication respectively, where τ_l is the time required to evaluate the next function at level l .*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a technical result. Then, theorem 2.9.2 follows easily as a consequence of this result.

The main inefficiency of algorithm MWAY_SEARCH derives from case 4 of stage 1, where a set Q_i^* , of size $s_i^* \leq (\lceil 2 \cdot m/p \rceil + d_{max})/\delta$, is received by processor p_i . We assume that $s_i^* \leq n/p$, which is always true (asymptotically) for some $\delta < 3 \cdot (m/n)$, $m \leq n$ and $d_{max} = o(n/p)$. We next outline a scheme for load-balancing the queries in stage 1 of algorithm MWAY_SEARCH so that each processor in stages 2–5 operates on a set of queries of total size at most $\lceil n/p \rceil$.

Recall that the queries in Q_i^* need to be evaluated by the recipient processor p_i , and therefore, can not be reallocated in order to achieve load-balance. The remaining queries on processor p_i , say, Q_i , are distributed among the processors so that each processor receives $\lceil n/p \rceil - s_i^*$ queries for a total of at most $\lceil n/p \rceil$. The load-balancing scheme is realized by a series of parallel-prefix and regular routing operations. Initially, the queries in $\cup_i Q_i$ are evenly distributed among the processors. This step can be effected by realizing a parallel-prefix sum operation on the size of the Q_i 's followed by a routing operation of size at most $\lceil n/p \rceil$. Let $\lceil n/p \rceil - s_i^*$ denote the *capacity* of processor p_i . A parallel-prefix sum operation is realized on the capacities in order to determine the final destination of the queries in $\cup_i Q_i$. The load-balancing operation is completed by realizing a routing operation of size at most $\lceil n/p \rceil$. The total time complexity of the aforementioned load-balancing scheme is bounded above by $O(n/p) + 2 \cdot C_{ppf}(p)$ and $L \cdot 2 + g \cdot O(n/p) + 2 \cdot M_{ppf}(p)$, for computation and communication respectively. We therefore derive the following analogue of proposition 2.9.1.

Proposition 2.9.2 *Let $G = (V, E)$ be an ordered h -level graph, with maximum out-degree d_{max} , where $|V| = m$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n, m, p and d_{max} such that $m \leq n$, $p = O(n)$ and $d_{max} = o(n/p)$, requires time*

$$C_{mway}(n, p) = \tau_l \cdot \left(\left\lceil \frac{n}{p} \right\rceil + o\left(\frac{n}{p}\right) \right) + O\left(\frac{n}{p}\right) + C_{rdx_srt}\left(n, o\left(\frac{n}{p}\right), p\right) + 3 \cdot C_{ppf}(p)$$

and

$$M_{mway}(n, p) = L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + M_{rdx_srt}\left(n, o\left(\frac{n}{p}\right), p\right) + M_{bcast}^{o(n/p)}(p) + 3 \cdot M_{ppf}(p),$$

for computation and communication respectively, where τ_l is the time required to evaluate the next function at level l . The overall time complexity of the algorithm is denoted by $T_{mway}(n, p) = C_{mway}(n, p) + M_{mway}(n, p)$.

Remark Even though the aforementioned load-balancing scheme improves asymptotically the performance of algorithm MWAY_SEARCH, in most practical situations an appropriately chosen δ would result in superior performance.

Proof (of theorem 2.9.2) The theorem follows easily from proposition 2.9.2 by substituting for T_{rdx_srt} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.4.1, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.9.6.2 The Case $m > n$ through Randomization

Consider an ordered h -level graph $G = (V, E)$, of maximum out-degree d_{max} , with vertex set $V = V_1 \cup V_2 \cup \dots \cup V_h$ and edge set E , where $|V| = m$. For the case $m > n$, the deterministic algorithm described in section 2.9.6 proves to be inefficient, since its running time depends on m . In this section, we remove all dependencies on m in the time bound of procedure MWAY_SEARCH of section 2.9.6 by introducing randomization. In the remainder of this section, we sketch the details of the randomized algorithm.

Informal Description of Algorithm The optimal randomized searching algorithm (on ordered h -level graphs) is an immediate adaptation of the searching algorithm of section 2.9.6.1 – with the exception that randomized techniques are utilized to improve and generalize the algorithm.

Our main result is summarized in the following theorem.

Theorem 2.9.3 *Let $G = (V, E)$ be an ordered h -level graph, with maximum out-degree d_{max} and number of possible out-degree values r , where $|V| = m$. Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n, p, d_{max} and r such that $p = O(n/(r \cdot d_{max}^2 \cdot \lg n))$ and $d_{max} = o(n/p)$, requires time [with probability $1 - O(h \cdot n^{-\rho \cdot n/(p \cdot r \cdot d_{max}^2 \cdot \lg n)})$] $\sum_l (\tau_l + O(1)) \cdot (n/p)$ and $L \cdot h \cdot O(\lg p) + g \cdot h \cdot O(n/p)$, for computation and communication respectively, where τ_l is the time required to evaluate the next function at level l .*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.9.3 follows easily as a consequence of these results.

In the randomized version, the data structure representing the ordered h -level graph is maintained in such a way that a vertex and its associated information (number of edges and edge pointers) is stored at some processor chosen uniformly at random. All information dependent on the storage address of this vertex is modified accordingly. Given this random distribution of vertices to processors, the implementation details of procedure MWAY_SEARCH remain almost identical. We assume that $p \leq n/(r \cdot d_{max}^2 \cdot \lg n)$, where r is the number of possible out-degree values of the underlying graph (e.g., for a balanced d -ary tree $r = 1$, since the only possible out-degree of an internal vertex is d). We note that in general, $r \leq d_{max}$.

The notable change in the randomized version is identified in the implementation of stage 1 of algorithm MWAY_SEARCH; a minor modification is also required in the implementation of stage 3. Stages 2, 4–5 of algorithm MWAY_SEARCH remain unmodified. In particular, we examine two cases in the analysis of stage 1 as described below. As the parameter δ of section 2.9.6 is irrelevant in our analysis, we fix $\delta = 1$.

Case 1 If a thread $\bar{Q}$ is of size $\bar{s}$ such that $\bar{d} \leq \bar{s}$, where $\bar{d}$ is the out-degree of the vertex associated with $\bar{Q}$, then a “read *NextInfo*” operation is realized, similar to cases 1–3 of stage 1 of algorithm MWAY_SEARCH. Each such thread $\bar{Q}$ receives information of size $\bar{s}$. Let the total size of all such threads, over all processors, be denoted by $S_1 \leq n$.

We next bound the number of queries that are sent to and received from any processor. Clearly, a processor can receive information of size at most $\lceil n/p \rceil$. In the remainder of the analysis of this case, we derive an upper bound on the size of information sent by any processor. Let r_{ik} be the number of threads of processor p_i such that the vertex associated with each such thread has out-degree $k = \bar{d}$, where $k \leq \bar{s}$. Then, $\sum_i r_{ik}$ is the number of such threads, over all p processors, whose associated vertices have out-degree k . Since such vertices are uniformly at random distributed among the p processors, we claim, by way of the inequality in theorem A.1.3, that with probability $1 - O(n^{-\rho \cdot n/(p \cdot r \cdot d_{max}^2 \cdot \lg n)})$, for any constant $\rho > 0$, each processor receives $O(\max\{\sum_i r_{ik}/p, n/(p \cdot r \cdot d_{max}^2)\})$ “read *NextInfo*” requests. For each such request, the processor needs to route information of size k . Summing over all r possible values of k , we claim that each processor routes information of size $O(\max\{\sum_k k \cdot (\sum_i r_{ik})/p, \sum_k k \cdot (n/(p \cdot r \cdot d_{max}^2))\})$. By definition we have that $\sum_{i,k} (k \cdot r_{ik}) \leq S_1$. Moreover, since $\bar{d} \leq d_{max}$, we have that $\sum_k k \leq r \cdot d_{max}$, and therefore, we obtain that a relation of size $O(\max\{S_1/p, n/(p \cdot d_{max})\})$ is routed. Thus, since $p \leq n/(\omega_n \cdot r \cdot d_{max}^2 \cdot \lg n)$ and $S_1 \leq n$, we have that a relation of size $O(n/p)$ is realized in stage 1 (cases 1–3).

Case 2 If a thread $\bar{Q}$ is of size $\bar{s}$ such that $\bar{s} < \bar{d}$, where $\bar{d}$ is the out-degree of the vertex associated with thread $\bar{Q}$, then a “send $\bar{Q}$ to $\hat{p}$ ” operation is realized, similar to case 4 of algorithm MWAY_SEARCH. Let the total size of all such threads, over all processors, be denoted by $S_2 \leq n$. It is clear that $S_2 = n - S_1$.

Each such thread $\bar{Q}$ sends information of size $\bar{s}$. We next bound the number of queries that are sent to and received from any processor. Clearly, a processor can send information of size at most $\lceil n/p \rceil$ (all the queries in its possession). In the remainder of the analysis of this case, we derive an upper bound on the size of information received by any processor. Let r_{ijk} be the number of threads of processor p_i of size $j = \bar{s}$ such that the vertex associated with each such thread has out-degree $k = \bar{d}$,

where $j < k$. Then, $\sum_i r_{ijk}$ is the number of such threads, over all p processors, of size j , whose associated vertices are of out-degree k . Since such vertices are uniformly at random distributed among the p processors, we claim, by way of the inequality in theorem A.1.3, that with probability $1 - O(n^{-\rho \cdot n / (p \cdot r \cdot d_{max}^2 \cdot \lg n)})$, each processor receives $O(\max \{j \cdot \sum_i r_{ijk} / p, j \cdot n / (p \cdot r \cdot d_{max}^2)\})$ queries. Summing over all r possible values of k and all d_{max} possible thread sizes j , we claim that each processor routes information of size at most $O(\max \{\sum_{j,k} j \cdot (\sum_i r_{ijk} / p), \sum_{j,k} j \cdot (n / (p \cdot r \cdot d_{max}^2))\})$. By a simple combinatorial argument, we have that $\sum_{i,j,k} (j \cdot r_{ijk}) \leq S_2$. Moreover, since $\bar{s} < \bar{d} \leq d_{max}$, we have that $\sum_{j,k} j \leq r \cdot d_{max}^2$, and therefore, we obtain that a relation of size $O(\max \{S_2/p, n/p\})$ is routed. Thus, since $p \leq n / (\omega_n \cdot r \cdot d_{max}^2 \cdot \lg n)$ and $S_2 \leq n$, we have that a relation of size at $O(n/p)$ is realized in stage 1 (case 4).

By way of the analysis of cases 1–2 above, at the end of stage 1 of the randomized algorithm MWAY_SEARCH, each processor maintains $\lceil n/p \rceil + O(n/p)$ queries. We can therefore load-balance these queries, using a scheme similar to that of sub-section 2.9.6.1, and subsequently, proceed as in stages 2–5 of algorithm MWAY_SEARCH. It remains to show how to modify the sorting operation of stage 3 to cope with the fact that $m > n$.

Whereas procedures SEQ_RADIX_SORT and RADIX_SORT (within procedure MWAY_SEARCH) are oblivious to the size m of the underlying graph, the analysis of algorithm SEQ_SUBQUAD_SORT heavily depends on m . To remove the dependency of the latter algorithm on m we substitute algorithm SEQ_SUBQUAD_SORT by two applications of SEQ_RADIX_SORT as follows.

Let $\bar{Q}$ be a thread and let $Index_{min}$ be the minimum *LevelIndex* over all queries in $\bar{Q}$ in the current level. We define the *Index* of a query q in $\bar{Q}$ to be the *LevelIndex* of the vertex associated with q minus the $Index_{min}$ of $\bar{Q}$; clearly, *Index* is in the range $[0, 1, \dots, d_{max} - 1]$. Let $\bar{S}$ be the total size of all threads $\bar{Q}_j$ of size $\bar{s}_j < \sqrt{d_j}$ on any particular processor. Each processor employs algorithm SEQ_RADIX_SORT to sort the queries in $\cup_j \bar{Q}_j$ with respect to their *Index*, maintaining for each query its thread number. Thus, by way of algorithm SEQ_RADIX_SORT (refer to section 2.9.2), we have that the queries in all threads of size $\bar{s}_j < \sqrt{d_j}$ can be sorted in time $O(\lceil \lg d_{max} / \lg \bar{S} \rceil \cdot \bar{S})$. Finally, each processor employs once more algorithm SEQ_RADIX_SORT to sort the queries with respect to their thread number. This latter operation results in globally sorting the queries in $\cup_j \bar{Q}_j$ with respect to the *LevelIndex* of their associated vertex. It remains to establish the time complexity of the latter sorting operation. By way of algorithm SEQ_RADIX_SORT and the fact that the number of queries is greater than the number of threads, we have that this step requires time $O(\bar{S})$. Since $\bar{S} \leq \lceil n/p \rceil$ and $d_{max} = o(n/p)$ we conclude that the overall time required by this process is therefore bounded above by $O(n/p)$.

This result completes the analysis of the randomized version of algorithm MWAY_SEARCH. We have therefore established the following analogue of proposition 2.9.2.

Proposition 2.9.3 *Let $G = (V, E)$ be an ordered h -level graph, with maximum out-degree d_{max} and number of possible out-degree values r , where $|V| = m$. Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n, p, d_{max} and r such that $p = O(n / (r \cdot d_{max}^2 \cdot \lg n))$ and $d_{max} = o(n/p)$, requires time [with probability $1 - O(n^{-\rho \cdot n / (p \cdot r \cdot d_{max}^2 \cdot \lg n)})$]*

$$C_{mway}(n, p) = \tau_l \cdot \left(\left\lceil \frac{n}{p} \right\rceil + o\left(\frac{n}{p}\right) \right) + O\left(\frac{n}{p}\right) + C_{rdx_srt} \left(n, o\left(\frac{n}{p}\right), p \right) + 3 \cdot C_{ppf}(p)$$

and

$$M_{mway}(n, p) = L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + M_{rdx_srt}\left(n, o\left(\frac{n}{p}\right), p\right) + M_{bcast}^{o(n/p)}(p) + 3 \cdot M_{ppf}(p),$$

for computation and communication respectively, where τ_l is the time required to evaluate the *next* function at level l . The overall time complexity of the algorithm is denoted by $T_{mway}(n, p) = C_{mway}(n, p) + M_{mway}(n, p)$.

Proof (of theorem 2.9.3) The theorem follows easily from proposition 2.9.3 by substituting for T_{rdx_srt} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.4.1, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.9.6.3 Improving Multi-Way Searching

We observe that not only can the bounds given in the previous sections be expressed in a variety of ways, but also algorithm MWAY_SEARCH leaves room for several improvements. As stated, the algorithm makes the implicit assumption that the time τ_l required to evaluate the *next* function f at some level l , $1 \leq l \leq h$, is uniform over all vertices of level l . Even though this assumption is valid in most practical situations (e.g., fixed degree ordered h -level graphs) there might be cases where the time required to evaluate f at level l varies considerably amongst the vertices of level l . We address this problem, by presenting a simple vertex-decomposition scheme that normalizes the time required to evaluate f over all vertices of the same level.

We also observe that algorithm MWAY_SEARCH is limited by the fact that it requires g to be upper bounded by $o((\sum_l \tau_l)/h)$ for the communication requirements to be within the lower order terms of the computation requirements (refer to theorem 2.9.2). This bound implies that the performance of algorithm MWAY_SEARCH is closely linked to the actual time τ_l required to evaluate the *next* function f at level l . For example, for $\tau_l = O(1)$ (e.g., constant degree ordered h -level graphs) the algorithm requires $g = o(1)$ in order to achieve comparable computation and communication requirements. Intuitively, the higher τ_l is, the better the performance of the algorithm. As τ_l is in general some non-decreasing function of the out-degree of the vertices at level l , the overall performance of algorithm MWAY_SEARCH can be improved by increasing the out-degree of the underlying ordered h -level graph. To this end, we present a rather natural level-composition scheme that coalesces a number of consecutive levels into a single level, and consequently, increases the out-degree of the underlying ordered h -level graph.

Modular Decomposition of Ordered h -Level Graphs In this section we outline a simple vertex-decomposition scheme that normalizes the time τ_l required to evaluate the *next* function f over all vertices of the same level. Let $G = (V, E)$ be an ordered h -level graph with vertex set $V = V_1 \cup V_2 \cup \dots \cup V_h$ and edge set E , where $|V| = m$. Consider an arbitrary level l of G and let $\tau_{l_{min}}$ and $\tau_{l_{max}}$ denote respectively the minimum and maximum time required to evaluate f over all vertices at level l . Similarly, let $d_{l_{min}}$ and $d_{l_{max}}$ denote respectively the minimum and maximum out-degree over all vertices at level l . In most practical situations, $\tau_{l_{min}}$ and $\tau_{l_{max}}$ are some non-decreasing functions of $d_{l_{min}}$ and $d_{l_{max}}$ respectively. For example,

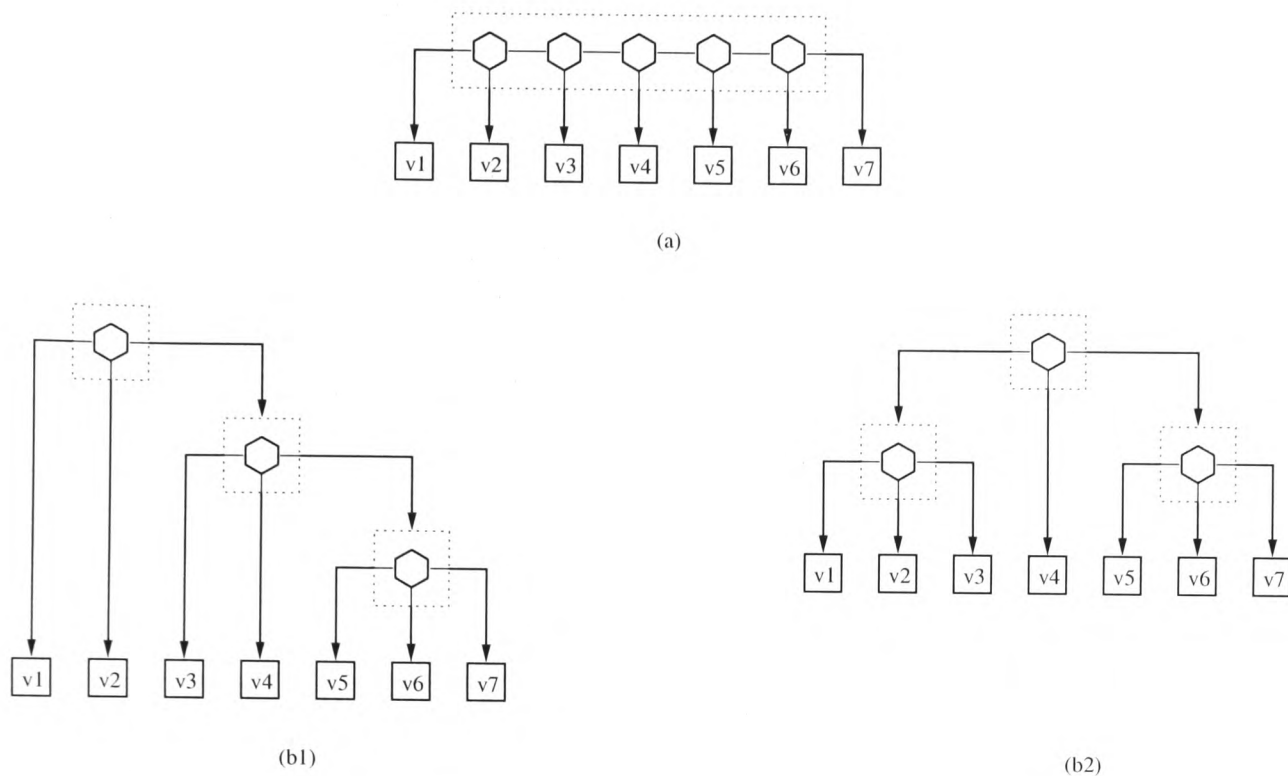


Figure 2.56: Modular decomposition of the vertex of out-degree 7 (case (a)) into the sub-vertices of out-degree 3 (cases (b1) and (b2)). The multi-level sub-vertex structure of case (b1) supports a linear-search *next* function and the multi-level vertex structure of case (b2) supports a binary-search *next* function. Solid-line squares and dotted-line rectangles represent vertices, solid-line arrows represent edges, and solid-line hexagons represent the keys associated with edges.

$$\tau_{l_{\min}} = \begin{cases} O(d_{l_{\min}}) & \text{if } f \text{ is a linear-search routine,} \\ O(\lg d_{l_{\min}}) & \text{if } f \text{ is a binary-search routine.} \end{cases}$$

Hence, the time required to evaluate the *next* function f over all vertices at level l can be normalized by *modularly decomposing* each vertex v_j in V_l into an $r_{l_j}^*$ -level structure, consisting of sub-vertices of minimum out-degree $d_{l_{\min}}^* = d_{l_{\min}}$ and maximum out-degree $d_{l_{\max}}^*$, such that $\tau_{l_{\max}}^* = (1 + o(1)) \cdot \tau_{l_{\min}}^*$. The aforementioned decomposition scheme guarantees that the time $\tau_l^* = \tau_{l_{\max}}^*$ required to evaluate the *next* function f is $(1 + o(1)) \cdot \tau_{l_{\min}}^*$, and therefore, the time normalization process is completed. If a vertex is decomposed into an r^* -level sub-vertex structure, then we say that the *decomposition factor* is $r^* \leq d_{\max}$ (for example, the decomposition factor of cases (b1) and (b2) of figure 2.56 is 3 and 2 respectively).

Remark The decomposition procedure is application specific and closely related to the *next* function f . We do not make any claims on how such a decomposition can be derived in general; in the remainder of this section we assume that such a decomposition procedure is available.

The adoption of such a vertex-decomposition scheme does not affect asymptotically the time complexity of our algorithm, it does, however, introduce a fundamental complication: by decomposing vertices into multi-level structures, property 3 (cases 2 and 3) of ordered h -level graphs is violated.

In order to overcome these complications we extend the notion of ordered h -level graphs to that of $r_{\max}^*$ -ordered h^* -level graphs that can accommodate the multi-level structures introduced by modularly decomposing the vertices of the underlying ordered h -level graph. In particular, a directed graph $G^* = (V^*, E^*)$ is called an $r_{\max}^*$ -ordered h^* -level graph if it has the following properties.

- (1) Every vertex v in V^* has out-degree of at most $d_{\max}^*$.

- (2) The number of sources d_s^* of G^* is at most d_{max}^* .
- (3) The maximum decomposition factor of G^* is r_{max}^* .
- (4) There exists an ordered h^* -level partitioning $V_1^*, V_2^*, \dots, V_{h^*}^*$ of G^* such that
 - every source of G^* is contained in V_1^* ,
 - if (u, v) is an edge of G^* , then $Level(u) < Level(v) \leq Level(u) + r_{max}^*$,
 - if (u, v) and $(\bar{u}, \bar{v})$ are two edges of G^* , such that both $Level(u) = Level(\bar{u})$ and $LevelIndex(u) < LevelIndex(\bar{u})$, then $Level(\bar{v}) - (r_{max}^* - 1) \leq Level(v) \leq Level(\bar{v}) + (r_{max}^* - 1)$, and if $Level(v) = Level(\bar{v})$, then also $LevelIndex(v) \leq LevelIndex(\bar{v})$.
- (5) Graph G^* has no double edges.

The parameter r_{max}^* in the above definition reflects the maximum decomposition factor over all vertices of the underlying ordered h -level graph; if $r_{max}^* = 1$, then the definitions of ordered h -level graphs and r_{max}^* -ordered h^* -level graphs coincide.

Informal Description of Algorithm The optimal deterministic searching algorithm (on r_{max}^* -ordered h^* -level graphs) is an immediate adaptation of the searching algorithm of section 2.9.6.1 – with the exception that non-trivial algorithmic techniques are employed to generalize the algorithm.

Our main result on r_{max}^* -ordered h^* -level graphs is summarized in the following theorem.

Theorem 2.9.4 *Let $G^* = (V^*, E^*)$ be an r_{max}^* -ordered h^* -level graph, with maximum out-degree d_{max}^* , where $|V^*| = m^*$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n, m^*, p, r_{max}^* and d_{max}^* such that $m^* \leq n$, $p = o(n/\lg n)$, $r_{max}^* = O(n/p)$ and $d_{max}^* = o(n/p)$, requires time $\sum_l (\tau_l^* + O(1)) \cdot (n/p)$ and $L \cdot h \cdot O(\lg p) + g \cdot h \cdot O(n/p)$, for computation and communication respectively, where τ_l^* is the time required to evaluate the next function at level l .*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.9.4 follows easily as a consequence of these results.

The procedure to manage multi-way searching on r_{max}^* -ordered h^* -level graphs is similar to procedure MWAY_SEARCH of section 2.9.6. The n search processes for all n queries are realized in a sequence of phases. The number of phases is upper bounded by the number of levels h^* of the underlying r_{max}^* -ordered h^* -level graph. At each phase l , the queries advance jointly through the graph along their search paths. We observe that, r_{max}^* -ordered h^* -level graphs, in contrast to ordered h -level graphs, allow queries to proceed to different levels of the graph. Therefore, the multi-way searching algorithm for r_{max}^* -ordered h^* -level graphs has to cope with the requirement of queries being processed on different levels of the underlying graph.

To this end, we extend the invariants assumed to hold for algorithm MWAY_SEARCH to accommodate the notion of queries being sorted with respect to the *Level* of their associated vertex – and still preserve within the *Level* the ordering with respect to *LevelIndex*. Since queries on different levels can proceed independently we can employ algorithm MWAY_SEARCH to realize searching on r_{max}^* -ordered h^* -level graphs.

It remains to show how the ordering of queries with respect to the *Level* of their associated vertex can be effected. We observe that, by way of property 4 (case 3) of r_{max}^* -ordered h^* -level graphs, the queries at some level l^* , $1 \leq l^* \leq h^*$, can proceed to at most r_{max}^* distinct consecutive levels of the underlying graph. Similarly, by way of property 4 (case 2), the queries from at most r_{max}^* distinct consecutive levels can converge to the same level l^* . Thus, no two queries that are r_{max}^* levels apart at phase $l-1$ of the search procedure can be on the same level at phase l . We next sketch the algorithmic scheme utilized to obtain the queries at phase l sorted with respect to the *Level* of their associated vertex.

Let the queries at phase $l-1$ be sorted with respect to the *Level* of their associated vertex. Denote by $k_{(l-1)_{min}}$ and $k_{(l-1)_{max}}$ the minimum and maximum *Level* of all vertices visited by queries at phase $l-1$. The queries are partitioned into $k = \lceil (k_{(l-1)_{max}} - k_{(l-1)_{min}}) / r_{max}^* \rceil$ buckets $H_1, H_2, \dots, H_k$, such that H_j , $1 \leq j \leq k$, consists of the queries that at phase $l-1$ visit some vertex with *Level* in $[i, \dots, i']$, where

$$[i, \dots, i'] = \begin{cases} [k_{(l-1)_{min}}, \dots, k_{(l-1)_{min}} + r_{max}^*] & \text{if } j = 1, \\ [k_{(l-1)_{min}} + (j-1) \cdot r_{max}^* + 1, \dots, k_{(l-1)_{min}} + j \cdot r_{max}^*] & \text{if } 1 < j < k, \\ [k_{(l-1)_{min}} + (k-1) \cdot r_{max}^* + 1, \dots, k_{(l-1)_{max}}] & \text{if } j = k. \end{cases}$$

The determination of $k_{(l-1)_{min}}$ and $k_{(l-1)_{max}}$ takes time $T_{ppf}(p)$. The partitioning of queries into buckets can be performed by way of a procedure similar to that of stage 5 of algorithm MWAY_SEARCH, and therefore, requires time $O(n/p) + C_{ppf}(p)$ and $L + g \cdot O(1) + M_{ppf}(p)$, for computation and communication respectively.

The queries within $H_j \cup H_{j+1}$, for all odd j , $1 \leq j \leq k-1$, are sorted independently with respect to the *Level* of their associated vertex at phase l . The sorting operation is realized by one of several methods depending on the size s_j of $H_j \cup H_{j+1}$. In particular, the algorithm employs

$$\begin{array}{ll} \text{SEQ_RADIX_SORT} & \text{if } s_j \geq (r_{max}^*)^\alpha \text{ and } H_j \cup H_{j+1} \text{ is exclusive,} \\ \text{RADIX_SORT} & \text{if } s_j \geq (r_{max}^*)^\alpha \text{ and } H_j \cup H_{j+1} \text{ is shared,} \\ \text{SEQ_SUBQUAD_SORT} & \text{if } s_j < (r_{max}^*)^\alpha, \end{array}$$

for some constant $0 < \alpha \leq 1/2$ to be specified later. In case 3, if $H_j \cup H_{j+1}$ is shared, then the lower-indexed of the two processors sharing $H_j \cup H_{j+1}$ performs procedure SEQ_SUBQUAD_SORT.

Recall that the queries at some level l^* can proceed to at most r_{max}^* distinct consecutive levels of the underlying graph, and the queries from at most r_{max}^* distinct consecutive levels can converge to the same level l^* . Thus, by way of algorithms SEQ_RADIX_SORT and RADIX_SORT (refer to section 2.9.2) we have that all queries in buckets of size greater than or equal to $(r_{max}^*)^\alpha$, can be sorted in time $O(n/p) + C_{rdx_srt}(n, 2 \cdot r_{max}^*, p)$ and $M_{rdx_srt}(n, 2 \cdot r_{max}^*, p)$ for computation and communication respectively. The $O(n/p)$ term is attributed to sequential radix sort – each exclusive bucket implies radix sorting of at least $(r_{max}^*)^\alpha$ integers in the range $[0, 1, \dots, 2 \cdot r_{max}^* - 1]$, for a total time of $O((\lg(2 \cdot r_{max}^*) / \lg((r_{max}^*)^\alpha)) \cdot (n/p)) = O(n/p)$, since α is a constant.

By the definition of SEQ_SUBQUAD_SORT (refer to section 2.9.2), each bucket of size less than $(r_{max}^*)^\alpha$ can be sorted in $o((r_{max}^*)^{2 \cdot \alpha})$ time. The total number of such buckets is at most

$$\left\lceil \frac{k_{(l-1)_{max}} - k_{(l-1)_{min}}}{r_{max}^*} \right\rceil \leq \left\lceil \frac{h^*}{r_{max}^*} \right\rceil \leq \frac{m^*}{r_{max}^*} + 1.$$

These buckets are evenly distributed amongst the processors by employing the load-balancing scheme of section 2.1.2 in time $T_{lbal}(n, \lceil n/p \rceil, p)$. Then, the time required to sort all such buckets of size $s < (r_{max}^*)^\alpha$ is $\lceil (m^*/r_{max}^* + 1)/p \rceil \cdot o((r_{max}^*)^{2\cdot\alpha}) = o((m^* \cdot (r_{max}^*)^{2\cdot\alpha-1})/p + (r_{max}^*)^{2\cdot\alpha})$.

The queries in buckets H_j and H_{j+2} are ordered with respect to each other, and therefore, by sorting the queries within $H_j \cup H_{j+1}$, for all even $2 \leq j \leq k-1$, the queries at phase l are globally sorted with respect to the *Level* of their associated vertex. Hence, we conclude that the aforementioned scheme requires overall time $O(n/p) + o((m^* \cdot (r_{max}^*)^{2\cdot\alpha-1})/p + (r_{max}^*)^{2\cdot\alpha}) + C_{rdx_srt}(n, 2 \cdot r_{max}^*, p) + C_{lbal}(n, \lceil n/p \rceil, p) + C_{ppf}(p)$ and $L \cdot 2 + g \cdot O(r_{max}^*)^\alpha + M_{rdx_srt}(n, 2 \cdot r_{max}^*, p) + M_{lbal}(n, \lceil n/p \rceil, p) + M_{ppf}(p)$ for computation and communication respectively. By substituting $\alpha = 1/2$ in the above analysis we obtain the following analogue of proposition 2.9.2.

Proposition 2.9.4 *Let $G^* = (V^*, E^*)$ be an r_{max}^* -ordered h^* -level graph, with maximum out-degree d_{max}^* , where $|V^*| = m^*$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n, m, p, r_{max}^* and d_{max}^* such that $m^* \leq n, p = O(n), r_{max}^* = O(n/p)$ and $d_{max}^* = o(n/p)$, requires time*

$$C_{mway^*}(n, p) = \tau_l^* \cdot \left(\left\lceil \frac{n}{p} \right\rceil + o\left(\frac{n}{p}\right) \right) + O\left(\frac{n}{p}\right) + 2 \cdot C_{rdx_srt}\left(n, O\left(\frac{n}{p}\right), p\right) + C_{lbal}\left(n, \left\lceil \frac{n}{p} \right\rceil, p\right) + 4 \cdot C_{ppf}(p)$$

and

$$M_{mway^*}(n, p) = L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + 2 \cdot M_{rdx_srt}\left(n, O\left(\frac{n}{p}\right), p\right) + M_{lbal}\left(n, \left\lceil \frac{n}{p} \right\rceil, p\right) + M_{bcst}^{o(n/p)}(p) + 4 \cdot M_{ppf}(p),$$

for computation and communication respectively, where τ_l^* is the time required to evaluate the next function at level l . The overall time complexity of the algorithm is denoted by $T_{mway^*}(n, p) = C_{mway^*}(n, p) + M_{mway^*}(n, p)$.

Remark In proposition 2.9.4, by abuse of notation, we refer to τ_l^* as the time required to evaluate f at level l , rather than at phase l . This is to be interpreted as $\tau_l^* = \max_i \{\tau_i^*\}$, where i ranges over all levels participating at phase l .

Proof (of theorem 2.9.4) The theorem follows easily from proposition 2.9.4 by substituting for T_{rdx_srt} , T_{lbal} , T_{bcst} and T_{ppf} the corresponding expressions in propositions 2.4.1, 2.1.5, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

Structural Composition of Ordered h -Level Graphs In this section we outline a simple level-composition scheme that coalesces a number of consecutive levels into a single level, and consequently, increases the out-degree of vertices. Let $G = (V, E)$ be an ordered h -level graph with vertex set $V = V_1 \cup V_2 \cup \dots \cup V_h$ and edge set E , where $|V| = m$. Consider an arbitrary level l of G and let τ_l denote the time required to evaluate the next function f at level l . As noted in sub-section 2.9.6.3, the performance of procedure MWAY_SEARCH is closely linked to the actual time τ_l required to evaluate f ; the higher τ_l is, the better the performance of the algorithm.

As τ_l is in general some non-decreasing function of the out-degree of the vertices at level l , the performance of algorithm MWAY_SEARCH can be improved by increasing the out-degree of the underlying

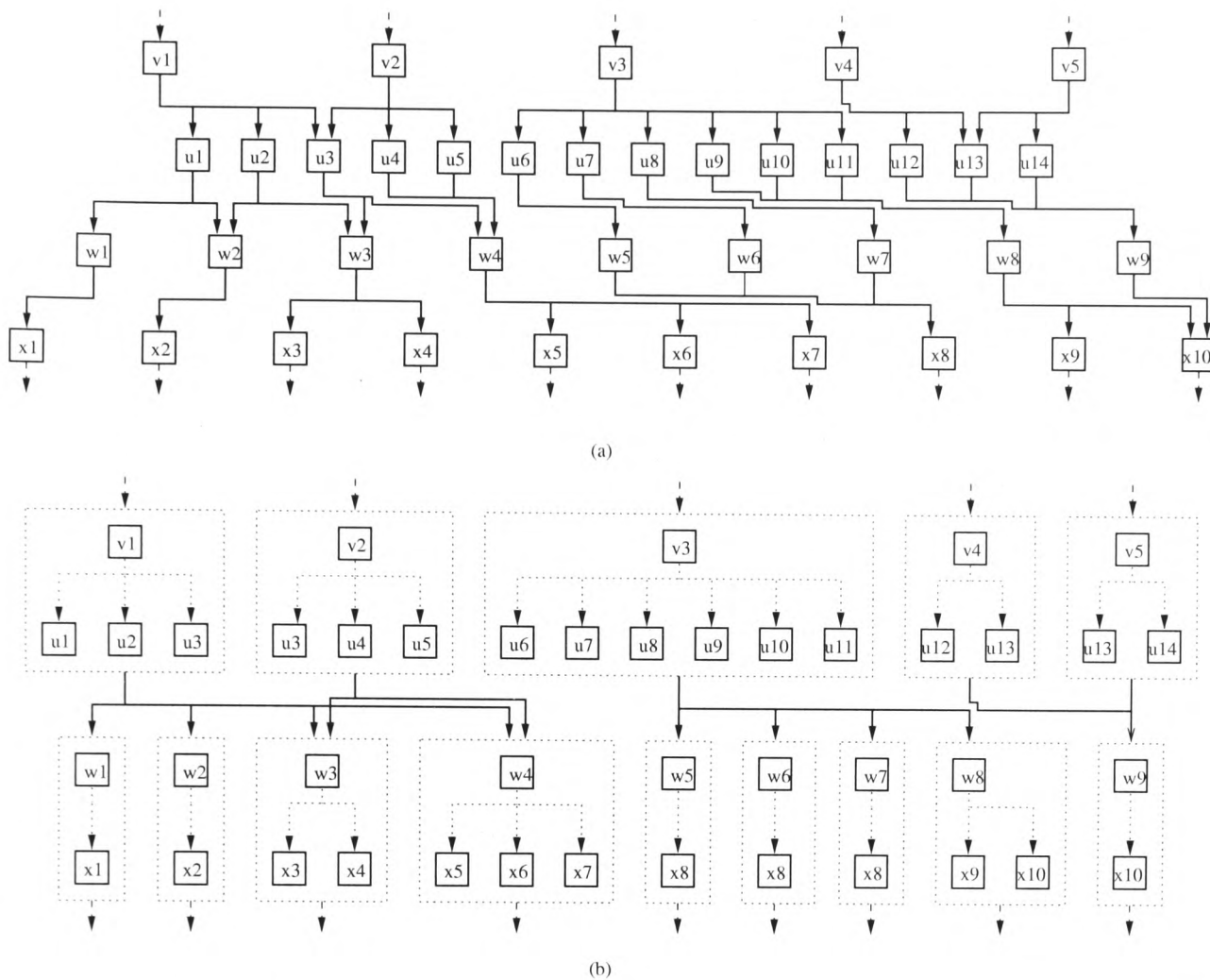


Figure 2.57: Structural composition of the one-level vertices of case (a) into the two-level vertices of case (b). *Solid-line squares and dotted-line rectangles represent respectively vertices and multi-level vertices, solid-line arrows and dotted-line arrows represent respectively edges and internal multi-level vertex edges.*

ordered h -level graph – by *structurally composing* the descendants of each vertex in V_l , up to some level $l' \geq l$, into a single multi-level vertex structure. If the vertices and their descendants are composed into R^* -level vertex structures, then we say that the *composition factor* is R^* (for example, the composition factor of figure 2.57 is 2).

Remark The composition procedure is application independent. We do not, however, make any claims on how such a composition can be derived in general; in the remainder of this section we assume that such a composition procedure is available.

Since in the aforementioned composition scheme each vertex in levels $l + 1$ through l' needs to be replicated as many times as its in-degree, we obtain that the resulting vertex structure corresponding to levels l through l' has size at most $\sum_j \sum_{k=i}^j |V_k|$, $l \leq j \leq l'$, which is at most $R^* \cdot \sum_k |V_k|$, where $R^* = l' - l + 1$. We thus obtain (in the worst case) an R^* -fold increase in the space requirements. We note, however, that in most practical situations the increase in memory requirements is negligible (e.g., d -ary trees do not involve an increment in space requirements).

The aforementioned composition scheme guarantees that the *next* function f at level l^* requires evaluation time $\tau_{l^*}^* = \sum_{l^*} \tau_{l^*}$, $l \leq l^* \leq l + R^* - 1$. Thus, the time required to evaluate the *next* function f at level l^* has increased, and simultaneously, the number of levels in the underlying graph has decreased. It is easy to verify that the adoption of such a level-composition scheme does not affect asymptotically the time complexity of our algorithm, it does, however, introduce a fundamental complication: by composing

a vertex and its descendants into a single multi-level vertex structure, property 3 (case 3) of ordered h -level graphs is violated.

In order to overcome this complication we extend the notion of ordered h -level graphs to that of R_{max}^* -ordered h^* -level graphs that can accommodate the multi-level vertex structures introduced by structurally composing the levels of the underlying ordered h -level graph. In particular, a directed graph $G^* = (V^*, E^*)$ is called an R_{max}^* -ordered h^* -level graph if it has the following properties.

- (1) Every vertex v in V^* has out-degree of at most d_{max}^* and size at most s_{max}^* , where the size of a vertex is the size of its internal structure (vertices and edges).
- (2) The number of sources d_s^* of G^* is at most d_{max}^* .
- (3) The maximum composition factor of G^* is R_{max}^* .
- (4) There exists an ordered h^* -level partitioning $V_1^*, V_2^*, \dots, V_{h^*}^*$ of G^* , such that
 - every source of G^* is contained in V_1^* ,
 - if (u, v) is an edge of G^* , then $Level(v) = Level(u) + 1$,
 - if (u, v) and $(\bar{u}, \bar{v})$ are two edges of G^* , such that both $Level(u) = Level(\bar{u})$ and $LevelIndex(u) < LevelIndex(\bar{u})$, then $Level(v) = Level(\bar{v})$ and $LevelIndex(v) \leq LevelIndex(\bar{v}) + (d_{max}^* - 1) \cdot \min\{1, R_{max}^* - 1\}$.
- (5) Graph G^* has no double edges.

The parameter R_{max}^* in the above definition reflects the maximum composition factor over all levels of the underlying ordered h -level graph; if $R_{max}^* = 1$, then the definitions of ordered h -level graphs and R_{max}^* -ordered h^* -level graphs coincide.

Informal Description of Algorithm The optimal deterministic searching algorithm (on R_{max}^* -ordered h^* -level graphs) is an immediate adaptation of the searching algorithm of section 2.9.6.1 – with the exception that non-trivial algorithmic techniques are employed to generalize the algorithm.

Our main result on R_{max}^* -ordered h^* -level graphs is summarized in the following theorem.

Theorem 2.9.5 *Let $G^* = (V^*, E^*)$ be an R_{max}^* -ordered h^* -level graph, with maximum out-degree d_{max}^* and maximum vertex size s_{max}^* , where $|V^*| = m^*$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n, m^*, p, d_{max}^* and s_{max}^* such that $m^* \leq n$, $p = o(n/\lg n)$, $d_{max}^* = o(n/p)$ and $s_{max}^* = o(n/p)$, requires time $\sum_l (\tau_l^* + O(1)) \cdot (n/p)$ and $L \cdot h \cdot O(\lg p) + g \cdot h \cdot O(n/p)$, for computation and communication respectively, where τ_l^* is the time required to evaluate the next function at level l^* . The algorithm requires $O(n/p + ((s_{max}^* + d_{max}^*) \cdot m^*)/p)$ space per processor.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.9.5 follows easily as a consequence of these results.

The procedure to manage multi-way searching on R_{max}^* -ordered h^* -level graphs is similar to procedure MWAY-SEARCH of section 2.9.6. The n search processes for all n queries are realized in a sequence of phases. The number of phases corresponds to the number of levels h^* of the underlying R_{max}^* -ordered h^* -level graph. At each phase l , the queries advance jointly through the graph along their search paths

from level $l^* - 1$ to level l^* . We observe that, for R_{max}^* -ordered h^* -level graphs, in contrast to ordered h -level graphs, sorting the queries within each thread with respect to the *LevelIndex* of their associated vertex is not sufficient for maintaining the queries globally sorted. Therefore, the multi-way searching algorithm for R_{max}^* -ordered h^* -level graphs has to cope with this complication.

To this end, we extend the scope of stage 3 of algorithm MWAY_SEARCH to handle sorting of queries with respect to the *LevelIndex* of their associated vertex in the underlying R_{max}^* -ordered h^* -level graph. We next sketch the algorithmic scheme utilized to obtain the queries at phase l sorted with respect to the *LevelIndex* of their associated vertex.

Let the queries at phase $l - 1$ be ordered with respect to the *LevelIndex* of their associated vertex. Denote by $K_{l_{min}}$ and $K_{l_{max}}$ the minimum and maximum *LevelIndex* over all vertices visited by queries at phase l . The queries are partitioned into $K = \lceil (K_{l_{max}} - K_{l_{min}}) / d_{max}^* \rceil$ buckets $H_1, H_2, \dots, H_K$, such that H_j , $1 \leq j \leq K$, consists of queries that at phase l visit some vertex with *LevelIndex* in $[i, \dots, i']$, where

$$[i, \dots, i'] = \begin{cases} [K_{l_{min}} + (j - 1) \cdot d_{max}^*, \dots, K_{l_{min}} + (j + 1) \cdot d_{max}^*] & \text{if } 1 \leq j < K, \\ [K_{l_{min}} + (K - 2) \cdot d_{max}^*, \dots, K_{l_{max}}] & \text{if } j = K. \end{cases}$$

We observe that the bucketing scheme used in the previous section (modular decomposition) does not work in this setting as the assignment of queries to buckets is realized with respect to the ordering of queries at phase l and not $l - 1$ (at phase $l - 1$ the queries are sorted with respect to *LevelIndex*).

By way of property 4 (case 3), the *LevelIndex* of any two queries that are not ordered with respect to *LevelIndex* at phase l may differ by at most d_{max}^* . This fact is exploited in the assignment of queries to the aforementioned “overlapped” buckets as follows. The queries are scanned from “left” to “right”, and query q is assigned to H_j or H_{j+1} depending on whether some query q' that belongs to bucket H_{j+1} but not H_j has already been encountered; if such a query q' has been encountered, then q is assigned to bucket H_{j+1} , otherwise, to bucket H_j .

The determination of $K_{l_{min}}$ and $K_{l_{max}}$ takes time $T_{ppf}(p)$. The partitioning of queries into buckets can be performed by way of a procedure similar to that of stage 5 of algorithm MWAY_SEARCH, and therefore, requires time $O(n/p) + C_{ppf}(p)$ and $L + g \cdot O(1) + M_{ppf}(p)$, for computation and communication respectively.

The queries within $H_j \cup H_{j+1}$, for all odd j , $1 \leq j \leq K - 1$, are sorted independently with respect to the *LevelIndex* of their associated vertex at phase l , and subsequently, the queries within $H_j \cup H_{j+1}$, for all even j , $2 \leq j \leq K - 1$, are sorted independently. The queries in buckets H_j and H_{j+2} are sorted with respect to each other, and therefore, by way of the aforementioned sorting operations, the queries at phase l are globally sorted with respect to the *LevelIndex* of their associated vertex. These sorting operations can be realized by employing an algorithmic scheme similar to that of the previous sub-section. Hence, we conclude that the aforementioned scheme requires overall time $O(n/p) + o((m^* \cdot (d_{max}^*)^{2 \cdot \alpha - 1})/p + (d_{max}^*)^{2 \cdot \alpha}) + C_{rdx_srt}(n, 2 \cdot d_{max}^*, p) + C_{lbal}(n, \lceil n/p \rceil, p) + C_{ppf}(p)$ and $L \cdot 2 + g \cdot O(d_{max}^*)^\alpha + M_{rdx_srt}(n, 2 \cdot d_{max}^*, p) + M_{lbal}(n, \lceil n/p \rceil, p) + M_{ppf}(p)$ for computation and communication respectively. By substituting $\alpha = 1/2$ in the above analysis we obtain the following analogue of proposition 2.9.2.

Proposition 2.9.5 *Let $G^* = (V^*, E^*)$ be an R_{max}^* -ordered h^* -level graph, with maximum out-degree d_{max} and maximum vertex size s_{max}^* , where $|V^*| = m^*$. Then, there exists a p -processor BSP algorithm for solving the multi-way searching problem on n queries, that for all integers n , m , p , d_{max}^* and s_{max}^* such that $m^* \leq n$, $p = O(n)$, $d_{max}^* = o(n/p)$ and $s_{max}^* = o(n/p)$, requires time*

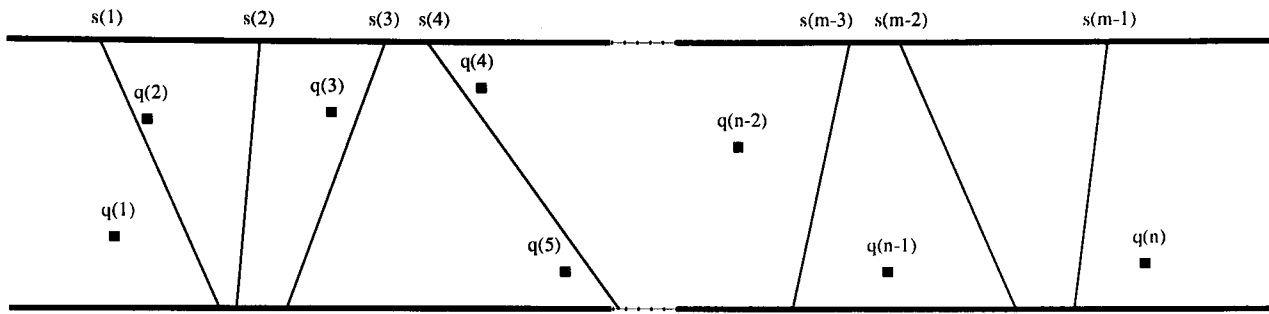


Figure 2.58: The multi-way point-location problem.

$$C_{Mway^*}(n, p) = \tau_{l^*}^* \cdot \left(\left\lceil \frac{n}{p} \right\rceil + o\left(\frac{n}{p}\right) \right) + O\left(\frac{n}{p}\right) + 2 \cdot C_{rdx_srt}\left(n, o\left(\frac{n}{p}\right), p\right) + C_{lbal}\left(n, \left\lceil \frac{n}{p} \right\rceil, p\right) + 4 \cdot C_{ppf}(p)$$

and

$$M_{Mway^*}(n, p) = L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + 2 \cdot M_{rdx_srt}\left(n, o\left(\frac{n}{p}\right), p\right) + M_{lbal}\left(n, \left\lceil \frac{n}{p} \right\rceil, p\right) + M_{bcast}^{o(n/p)}(p) + 4 \cdot M_{ppf}(p),$$

for computation and communication respectively, where $\tau_{l^*}^*$ is the time required to evaluate the next function at level l^* . The overall time complexity of the algorithm is denoted by $T_{Mway^*}(n, p) = C_{Mway^*}(n, p) + M_{Mway^*}(n, p)$. The algorithm requires $O(n/p + ((s_{max}^* + d_{max}^*) \cdot m^*)/p)$ space per processor.

Proof (of theorem 2.9.5) The theorem follows easily from proposition 2.9.5 by substituting for T_{rdx_srt} , T_{lbal} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.4.1, 2.1.2, 2.1.5 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

2.9.6.4 One-Optimal Applications of Multi-Way Searching in Computational Geometry Algorithms

By substituting appropriate values for the parameters in the results of the previous sections a multiplicity of optimality bounds on the runtime are derived. As noted in [75], the best optimality bounds are obtained by selecting the parameters with regard to the actual values of n , p , L and g . By making, however, more generic selections for the underlying parameters and worst-case bounds for p , L and g , more general results in terms of n can be obtained. In the following sections, we illustrate this claim by presenting a variety of one-optimal results in computational geometry that are derived as direct applications of theorems 2.9.2, 2.9.4 and 2.9.5.

Multi-Way Point Location through Multi-Way Searching In this section we apply the methods we developed in section 2.9.6 to outline an efficient BSP solution to an important computational geometry problem.

Problem Given a set of n points (queries) $X = \{x_1, x_2, \dots, x_n\}$ and an ordered partitioning Y of a two-dimensional horizontal strip V by $m - 1$ non-intersecting line segments $S = \langle s_1, s_2, \dots, s_{m-1} \rangle$ into m slabs, determine, for each $x \in X$, the vertical slab it belongs to.

The solution to this problem, which we call *multi-way point location* (refer to figure 2.58 for an illustration), is central to several other geometric problems [172] (e.g., planar point location, plane sweep, and segment trees). In the remainder of this section we assume $n = m$, the case $n \neq m$ can be derived in a similar way.

The algorithm for solving the multi-way point-location problem consists of two phases. In the first phase an ordered h -level graph G is constructed based on the ordered partitioning Y of the two-dimensional horizontal strip V . In particular, a balanced d -ary tree is constructed on the n vertical slabs, where $d = \lceil (n/p)^\alpha \rceil$ for some positive constant $\alpha < 1$. In the second phase we utilize algorithm `MWAY_SEARCH`, to determine, for each $x \in X$, the slab it belongs to. The following result shows how to efficiently implement the first phase.

Lemma 2.9.2 *For any constant α , $0 < \alpha < 1$, there exists a p -processor BSP algorithm for constructing a balanced d -ary tree T on n ordered items, that for all positive integers n , p and d such that $p = O(n)$ and $d = \lceil (n/p)^\alpha \rceil$ requires time $O(n/p) + O(d \cdot \log_d p)$ and $L \cdot \lceil \log_d p \rceil + g \cdot O(d \cdot \log_d p)$, for computation and communication respectively.*

Proof Let l be an arbitrary level of T . We define every d -th item of level l of T to be a *pivot* item. The construction of the balanced tree proceeds from the leaves to the root as follows. For each level l of T , level $l - 1$ is constructed by only considering the pivot keys of level l . Since the height of T is $\lceil \log_d n \rceil$ the construction takes $\lceil \log_d n \rceil$ phases, of which $\lceil \log_d p \rceil$ phases are non-local (require inter-processor communication). Each such phase requires time $O(d)$ and $L + g \cdot O(d)$ for computation and communication respectively. Since the overall cost of local computations is bounded above by $O(n/p)$, the lemma follows. $\square$

The so constructed balanced tree T has n leaves and out-degree $d = \lceil (n/p)^\alpha \rceil$, and therefore, its height is at most $\lceil \lg n / (\alpha \cdot \lg(n/p)) \rceil$. Moreover, since the in-degree of any vertex $v \in T$ is at most one and the number of internal vertices is at most $n - 1$, the number of edges in T is bounded above by $2 \cdot n - 1$. Combining these observations with lemma 2.9.2 and theorem 2.9.2 we obtain the following result which summarizes the discussion of this section – note that sequentially, multi-way point location requires time $\Theta(n \cdot \lg n)$.²⁷

Optimality Claim 2.9.1 *Let Y be an ordered partitioning of a two-dimensional horizontal strip V , where $|Y| = m$. Then, the aforementioned discussion implies that there exists a p -processor BSP algorithm for solving the multi-way point-location problem induced by n points, that for all positive integers n , m , p and L such that $m \leq n$, $p = o(n / \lg n)$, $L = o((n/p) \cdot (\lg(n/p) / \lg p))$ and $g = o(\lg(n/p))$, has $\pi = 1 + o(1)$ and $\mu = o(1)$.*

Range Searching through Multi-Way Search In the following discussion we apply the methods we developed in section 2.9.6 and sub-section 2.9.6.3 to outline an efficient BSP solution to an important computational geometry problem.

Problem Given a set of n range queries $Q = \{q_1, q_2, \dots, q_n\}$ and a set $\bar{X}$ of m two-dimensional points, determine, for each $q \in Q$, the subset of $\bar{X}$ contained in the query domain of q .

The solution to this problem, which we call *range searching*, is of fundamental importance in computational geometry [172]. In the remainder of this section we assume $n = m$, the case $n \neq m$ can be derived

²⁷The sequential solution to multi-way point-location problem entails, for each $x \in X$, a *binary-search* process on the m slabs; clearly, this method requires optimal time $\Theta(n \cdot \lg n)$.

in a similar way. We also assume that range queries are rectangles and that $\bar{X}$ consists of two-dimensional points, the generalization to higher dimensions can be realized easily.

The algorithm for solving the range-searching problem is an adaptation on the BSP model of the “range tree method” [32, 190]. The n abscise on the x -axis of set $\bar{X}$ are assumed to be normalized to the integers in $[0, 1, \dots, n-1]$ by their rank. These n abscise induce $n-1$ elementary intervals $[i, i+1]$, $0 \leq i < n-1$. The range tree method partitions the interval $[0, n-1]$ into elementary intervals by way of the *segment tree* structure [172]. The segment tree – a rooted binary tree of depth $O(\lg n)$ – is a data structure that allows the partitioning of an arbitrary interval, whose extremes belong to the set of n given abscise, into its constituent elementary intervals. In particular, the segment tree corresponding to interval $[l, u]$ is recursively built as follows: (i) it consists of a root with parameters l and u , and if $u - l > 1$, (ii) of a left segment tree corresponding to interval $[l, \lfloor (l+u)/2 \rfloor]$ and a right segment tree corresponding to interval $[\lfloor (l+u)/2 \rfloor, \dots, u]$ (we note that the segment tree construction can be optimally realized by way of the tree construction algorithm of the previous sub-section).

The *range tree* T is then constructed as follows: (i) it consists of a primary segment tree $\bar{T}$ corresponding to the x -coordinates of the points in $\bar{X}$, and (ii) for each node $v \in \bar{T}$, of a standard threaded binary tree on the y -coordinates of the points corresponding to the interval associated with node v [172]. In view of these properties, the use of the range tree in range searching is straightforward. The segment tree is employed for searching the x -coordinate. This search process identifies for each range query a unique set of nodes (the *allocation nodes*) of size at most $2 \cdot \lg n - 2$ [172]. Then, the threaded binary trees associated with these allocation nodes are employed for searching the y -coordinate. An illustration of the range-tree method is presented in figure 2.59 (see also [172]).

The aforementioned search process on range tree T resembles the search process on ordered h -level graphs since both $\bar{T}$ and its associated threaded binaries trees can be expressed as ordered h -level graphs. By employing the structural composition scheme of section 2.9.6.3 with $R_{max}^* = \alpha \cdot \log(n/p)$, the maximum out-degree $d_{max} = 2$ of $\bar{T}$ and its associated threaded binaries trees can be increased from constant to $2^{R_{max}^*} = (n/p)^\alpha$, for any constant $0 < \alpha < 1$. Combining these observations with theorem 2.9.5 we obtain the following result, which summarizes the discussion of this section – note that sequentially, range-searching requires time $\Theta(n \cdot \lg n)$ [172].²⁸

Optimality Claim 2.9.2 *Let $\bar{X}$ be a set of two-dimensional points (along with their corresponding range tree T), where $|T| = m$. Then, the aforementioned discussion implies there exists a p -processor BSP algorithm for solving the range-searching problem induced by n range queries, that for all positive integers n , m , p and L satisfying $m \leq n$, $p = o(n/\lg n)$, $L = o((n/p) \cdot (\lg(n/p)/\lg p))$ and $g = o(\lg(n/p))$, has $\pi = 1 + o(1)$ and $\mu = o(1)$.*

Planar Subdivision Point Location through Multi-Way Search In the subsequent discussion we apply the methods we developed in sections 2.9.6 and 2.9.6.3 to demonstrate an efficient BSP solution to yet another important computational geometry problem.

Problem Given a set of n points (queries) $X = \{x_1, x_2, \dots, x_n\}$ and an m -vertex connected planar straight-line graph G of vertex degree ≥ 2 , determine, for each $x \in X$, the planar subdivision it belongs to.

²⁸The sequential range-tree method we presented in this section requires time $\Theta(n \cdot \lg^2 n)$; however, in [147, 210] it is shown that a straightforward variant of the aforementioned method requires optimal time $O(n \cdot \lg n)$. The latter variant is assumed in optimality claim 2.9.2.

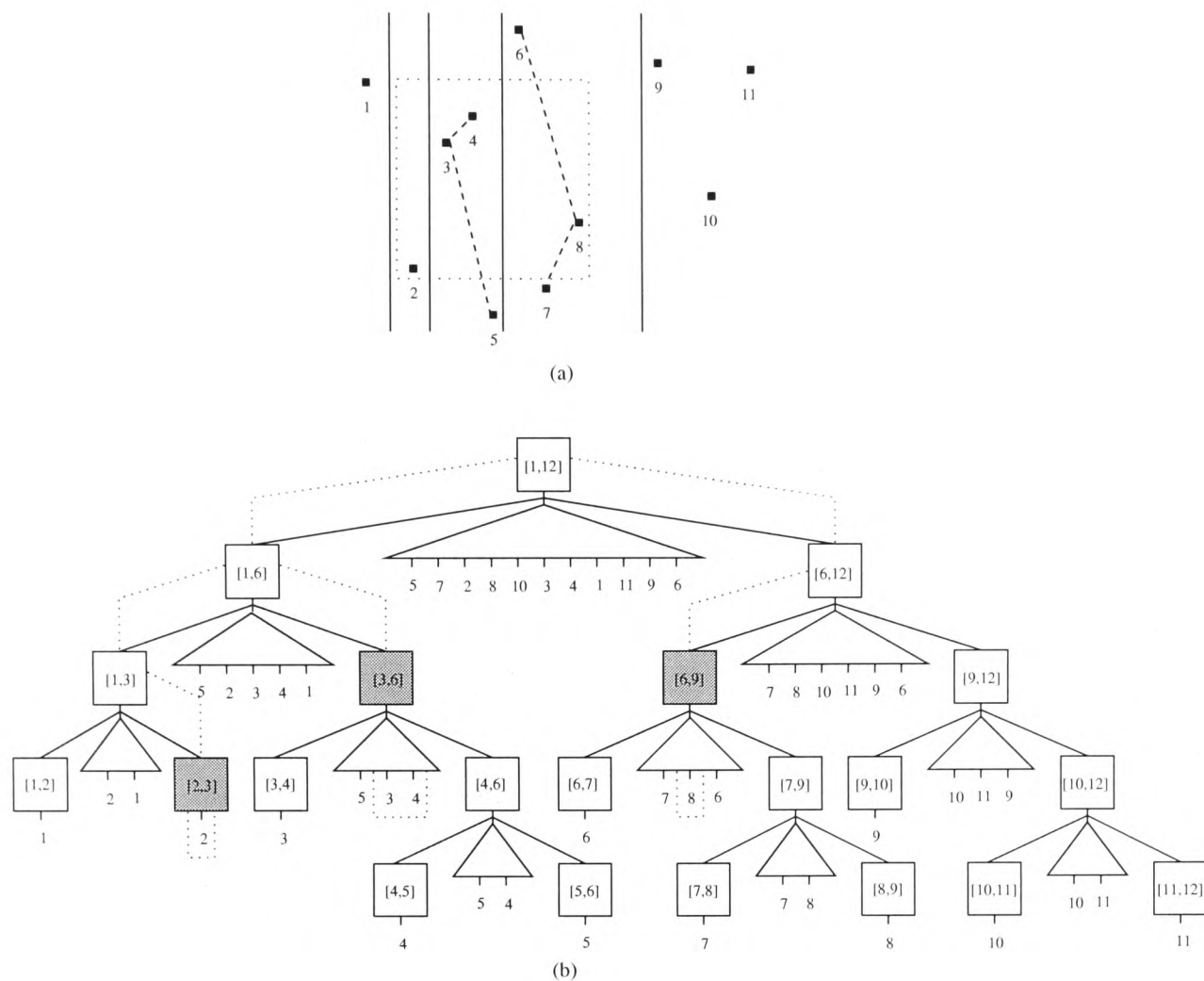


Figure 2.59: A single range query (a) and its associated search process (b).

The solution to this problem, which we call *planar subdivision point location*, is of fundamental importance in computational geometry [172]. In the remainder of this section we assume $n = m$, the case $n \neq m$ can be derived in a similar way.

The algorithm for solving the planar subdivision point-location problem is an adaptation on the BSP model of the “triangulation refinement method” [129]. The n -vertex connected planar straight-line graph G is assumed to be a triangulation (all bounded regions are triangles). In addition – for technical reasons – it is convenient to assume that the triangulation is inscribed within a triangular boundary (refer to figure 2.60 for an illustration).

Let G be such an n -vertex triangulation and consider the sequence of triangulations $\langle S_1, S_2, \dots, S_h \rangle$, where $S_h = G$ and S_{l-1} is obtained from S_l by (i) removing the set of non-adjacent non-boundary vertices of degree less than or equal to $d_{max} = 12$ of S_{l-1} and their incident edges, and (ii) re-triangulating the polygons arising from the removal of vertices and edges.

Along the lines of [172], we note that a triangle r_j may appear in many triangulations; however, r_j is conventionally said to belong to triangulation S_{l-1} (denoted by $r_j \in S_{l-1}$) if r_j is generated in step (ii) of the construction of S_{l-1} . Consider the *directed acyclic graph* T , whose nodes represent triangles and whose topology is defined as follows. There is an out-going edge from triangle r_k to triangle r_j , if during the construction of S_{l-1} the following conditions are true: (i) r_j is eliminated from S_l in step (i) of the construction of S_{l-1} , (ii) r_k is generated in step (ii) of the construction of S_{l-1} , and (iii) $r_k \cap r_j \neq \{\}$. Then, in [172], it is shown that the following properties hold for such an n -vertex directed acyclic graph T .

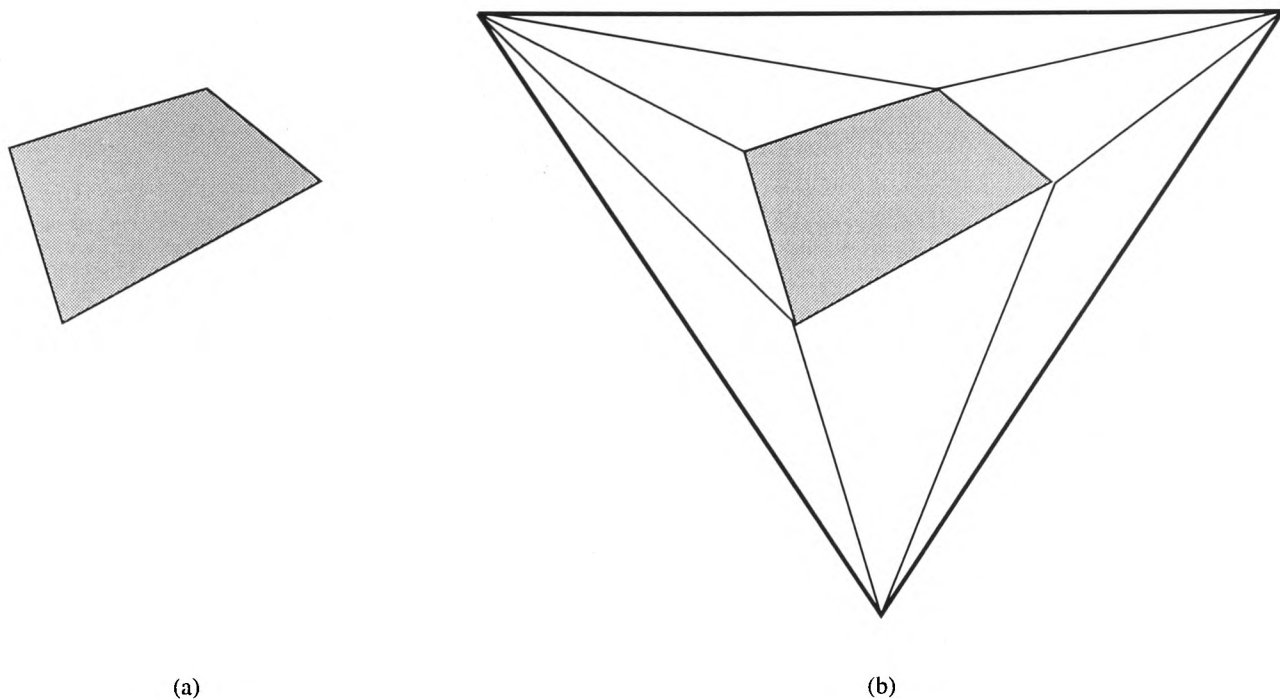


Figure 2.60: A given triangulation (a) and the same triangulation within an inscribing triangle (b).

- (1) $|S_h| = 2 \cdot n - 5$.
- (2) $|S_{l-1}| \leq \alpha \cdot |S_l|$, $2 \leq l \leq h$, for some constant $0 < \alpha < 1$.
- (3) Each triangle $r \in S_{l-1}$ intersects at most $d_{max} = 12$ triangles in S_l , and vice versa.

A sequence of triangulations along with their corresponding acyclic directed graph T is illustrated in figure 2.61 (see also [172]); the encircled vertices are those which are removed at each step.

As noted in [172], once T is available, it is straightforward to identify how point location proceeds. The underlying primitive operation is “inclusion in triangle”, which is obviously performed in constant time. The n search processes for all n points (queries) are realized in a sequence of phases. The number of phases corresponds to the number of levels of the underlying directed acyclic graph T . At each phase l , $1 \leq l \leq h$, the points advance through the graph along their search paths from level $l-1$ to level l , that is, once a point (query) has reached a triangle (node) in the path, test for inclusion in all of its descendants is performed; since the point is included in exactly one descendant, this advances the search process by one level.

The aforementioned search process on directed acyclic graph T resembles the search process on ordered h -level graphs. The fundamental difference, however, is identified in the fact that T does not satisfy property 3 (case 3) of ordered h -level graphs. In section 2.9.6.3 we outlined a scheme that relaxes property 3 (case 3) of ordered h -level graphs to property 4 (case 3) of R_{max}^* -ordered h^* -level graphs. The latter graph resembles closer the structure of T , but still, fails short of capturing its overall nature. In particular, directed acyclic graph T satisfies the following analogue of property 4 (case 3) of R_{max}^* -ordered h^* -level graphs:

- if (u, v) and $(\bar{u}, \bar{v})$ are two edges of T , such that $Level(u) = Level(\bar{u})$ and $Level_Index(u) < Level_Index(\bar{u})$, then $Level(v) = Level(\bar{v})$ and $Level_Index(v) \leq Level_Index(\bar{v}) + |S_{Level(u)}|$,

where $|S_{Level(u)}| \leq \alpha^{h-Level(u)} \cdot (2 \cdot n - 5)$. In view of this property, it is not difficult to verify that the scheme proposed in section 2.9.6.3 for R_{max}^* -ordered h^* -level graphs can be efficiently adapted to directed acyclic graphs; the additional time requirements for this scheme can be shown to be upper bounded by

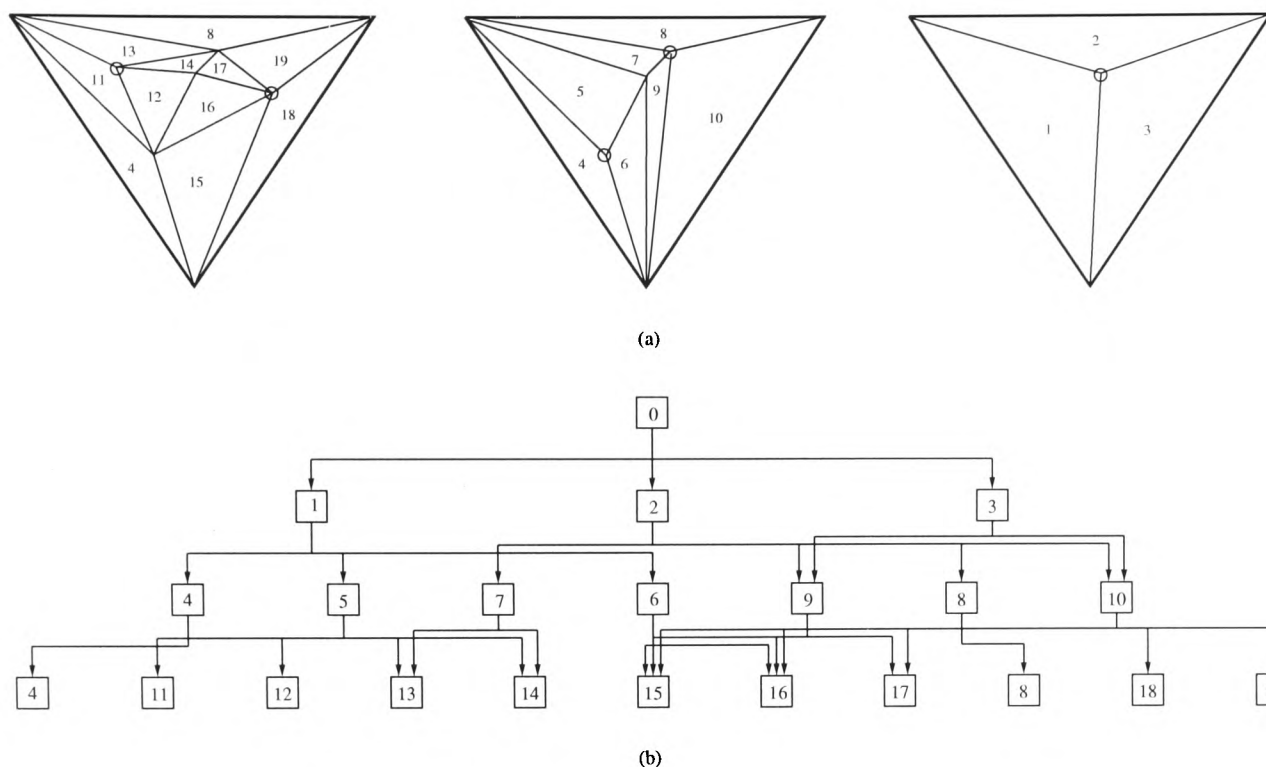


Figure 2.61: A sequence of triangulations (a) and the corresponding acyclic graph (b); the encircled vertices are those which are removed at each phase.

$$\sum_{l=1}^h T_{rdx_srt}(n, \alpha^{l-1} \cdot (2 \cdot n - 5), p).$$

By employing the structural composition scheme of section 2.9.6.3 with $R_{max}^* = \alpha \cdot \log_{12}(n/p)$, the maximum out-degree $d_{max} = 12$ of T can be increased from constant to $2^{R_{max}^*} \geq (n/p)^{\alpha/\lg 12}$, for any constant $0 < \alpha < 1$.²⁹ Additionally, if the node sizes in the resulting structure vary in size, then the modular decomposition scheme of section 2.9.6.3 can be employed to smooth out this variation. Combining the aforementioned result and these observations with theorems 2.9.4 and 2.9.5 we obtain the following result, which summarizes the discussion of this section – note that sequentially, planar-subdivision point location requires time $\Theta(n \cdot \lg n)$.

Optimality Claim 2.9.3 *Let G be a connected planar straight-line graph (given as a triangulation inscribed in a triangular boundary along with its directed acyclic graph T), where $|T| \leq n$. Then, the aforementioned discussion implies there exists a p -processor BSP algorithm for solving the planar subdivision point-location problem induced by the n points, that for all positive integers n , m , p and L satisfying $m \leq n$, $p = o(n/\lg n)$, $L = o((n/p) \cdot (\lg^2(n/p)/(\lg n \cdot \lg p)))$ and $g = o(\lg^2(n/p)/\lg n)$, has $\pi = 1 + o(1)$ and $\mu = o(1)$.*

2.9.6.5 Further Applications of Multi-Way Searching in Computational Geometry Algorithms

In [57] it is shown that the following problems admit efficient solutions on the hyper-cube model by way of multi-way searching: next element search, trapezoidal decomposition and triangulation [57]. It is our belief

²⁹The minimum out-degree of T is considered to be at least 2, since nodes with out-degree less than 2 are either leaf nodes or point to themselves, and therefore, can be safely ignored from further consideration.

that by employing the multi-way searching algorithm of section 2.9.6 it is possible to derive communication efficient BSP solutions for the aforementioned problems – there is no fundamental impediment known that would preclude this derivation done by communication efficient BSP algorithms.³⁰

2.10 Applications of Fundamental Algorithms to Computational Geometry Algorithms

In this section we apply the techniques we developed in this chapter to design, analyze and implement simple – yet optimally efficient – BSP algorithms for a variety of fundamental computational geometry operations, namely *convex hulls*, *geometric searching* and *visibility*. Computational geometry [172] has always attracted a great deal of research interest both in sequential [54, 172, 203] and parallel computation [3, 9, 18, 59, 119, 184] because of its many important applications. Parallel computational geometry is a branch of computer science concerned with the study of designing, analyzing and implementing efficient – practical being of great importance nowadays – parallel algorithms to treat computational problems dealing with collections of Euclidean objects. This wealthy class of problems arises naturally, not only in the obvious application areas such as computer graphics, computer aided design (CAD), image analysis, pattern recognition and classification, robotics, but also in statistics, management science, operations research, and database search [32, 59, 119, 172, 184, 203]. The material of this section is based on the following papers:

- (1) “Portable and scaleable algorithm design on the IBM SP2: The bulk-synchronous parallel paradigm.” In *Proceedings of the Sup-Eur’96 Conference*, Krakow, Poland, September 1996 [11].
- (2) “Direct bulk-synchronous parallel algorithms in computational geometry.” Technical Report PRG-TR-10-96, Computing Laboratory, Oxford University, May 1996 [196]. Also as *Qualifying Dissertation to Support an Application for Transfer of Status from a Probationary Research Student to a Student for the Degree of Doctor of Philosophy at the University of Oxford*, September 1995.

2.10.1 Preliminaries and Notation

Let $U = \langle u_1, u_2, \dots, u_n \rangle$ be a sequence of points in d -dimensional Euclidean space. Without loss of generality, we assume that no three points appearing in sequence U are co-linear (degenerate) – if the points are degenerate we can remove the degeneracy by adopting techniques for coping with degeneracies (e.g., [66, 68]). We consider, in this section, the problem of computing the *convex hull*, *dominance counting* and *lower envelope* of sequence U . In the forthcoming discussion the following notation is used.

- By E^d we denote the d -dimensional Euclidean space, with metric $(\sum_{i=1}^d x_i^2)^{\frac{1}{2}}$, and by $u = (x_1, x_2, \dots, x_d)$ we denote a *point (vector)* u in E^d .
- The combination $a \cdot u_1 + (1 - a) \cdot u_2$, for some real $0 \leq a \leq 1$, describes a *straight line segment*. The straight line segment joining vectors u_1 and u_2 is denoted by $\overline{u_1 u_2}$.

³⁰Unfortunately, the underlying details of these solutions are rather complicated and technical, and therefore, beyond the scope of this work.

- A domain D of vectors in E^d is *convex*, if for any pair of vectors (u_1, u_2) in D , the straight line segment $\overline{u_1 u_2}$ belongs entirely in D [172].
- The *convex hull* of a set of vectors in E^d is defined as the smallest convex domain in E^d containing the entire set.
- A collection of vectors V (*vertex set*) and straight line segments E (*edge set*) describes a *graph* G . A graph is *planar* if the set of edges and vertices composing the graph belong in E^2 without crossing – the special case where all its faces are triangles, defines a *triangulation*.

The restriction to Euclidean geometry is suggested by the fact that the vast majority of applications are formulated in Euclidean space, however, such a restriction is unessential for all of the applications to be considered in the remainder of this section.

2.10.2 Cost Model

The performance of the BSP computational geometry algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, each (ii) operation over an associative operator in parallel-prefix computations, and each (iii) operation over Euclidean objects in $E^{O(1)}$. Furthermore, for some constants c_{srt} , c_{chull} , c_{dcnt} , c_{lenv} , time $c_{srt} \cdot n \cdot \lg n$, $c_{chull} \cdot n \cdot \lg n$, $c_{dcnt} \cdot n \cdot \lg n$, and $c_{lenv} \cdot n \cdot \lg n$ is charged respectively for solving the sorting, convex-hull, dominance-counting and lower-envelope problems for a sequence of points in E^2 of size n – within the context of each separate section we denote c_{chull} , c_{dcnt} and c_{lenv} by c_* .

Optimal $\Theta(n \cdot \lg n)$ time sequential convex-hull, dominance-counting and lower-envelope algorithms are well-known [172]. In the following sections, we design, analyze and implement simple optimal BSP algorithms for the aforementioned problems – optimality is realized for an ample range of the BSP parameters.

2.10.3 Related Work and New Results

There has been a notable amount of previous work – mainly on the PRAM model – on parallel convex hull computations (e.g., [6, 161, 168] on the PRAM model, [58] on an abstract parallel communication model, [161] with [145] on the hyper-cube model, [184] on the butterfly model, [45] on the cube-connected-cycles model). The exhibit in table 2.9 summarizes the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality.

With reference to table 2.9 we note that the randomized butterfly algorithm of [183] does not migrate efficiently to the BSP model as a result of its inherent requirement for a non-bulk communication pattern (butterfly style communication) – as opposed to a bulk communication pattern (BSP style communication).

The optimal BSP convex-hull construction algorithms of section 2.10.4 and sub-section 2.10.7.2 improve upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [45],
- (2) the asymptotic communication time requirements of [6, 45, 161, 168, 184],

Refs	Computation	Communication	Implem.	Constant	Procs
[168]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■	$O(\frac{n}{\lg n})$
[45]	$O\left(\frac{n}{p} \cdot \lg^2 n\right)$	$O\left(\frac{n}{p} \cdot \lg n \cdot \lg p\right)$	■ ■	■ ■	$O(n)$
[6]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■	$O(n^\epsilon)$
[161]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■ ■	■ ■ ■ ■	$O(n)$
[184]	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■ ■ ■	$O(n)$
Sec 2.10.4	$(c_* + c_{srt}) \cdot \frac{n}{p} \cdot \lg n + o\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p}\right)$	■	■ ■	$O(\sqrt{n})$
Sec 2.10.7.2	$O\left(\frac{n}{p} \cdot \lg n\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg n}{\lg(n/p)}\right)$	■ ■ ■	■ ■ ■	$O(n)$

Table 2.9: Performance comparison of parallel convex-hull algorithms (for any constant $0 < \epsilon < 1$) – the cyan color is associated with randomized approaches.

- (3) the multiplicative constant factor requirements of [6, 45, 161, 168, 184] (algorithm of section 2.10.4),
- (4) the implementation effort requirements of [161, 184], and
- (5) the parallel slackness requirements (to achieve optimality) of [6, 168] (algorithm of subsection 2.10.7.2).

The principal contribution, however, of the convex-hull construction algorithm of section 2.10.4 is attributed to the *low communication cost* (refer to case 2) and the *small constant multiplicative factors* (refer to case 3). Additionally, our techniques provide *one-optimality* for an ample range of the BSP parameters. Accordingly, the main contribution of the convex-hull construction algorithm of section 2.10.7.2 is attributed to the *low communication cost* (refer to case 2) and the *low amount of parallel slackness* required to achieve optimality (refer to case 5).

Remark The comments that apply to the dominance-counting and lower-envelope algorithms of sections 2.10.5 and 2.10.6 are similar to those of the convex-hull algorithm of section 2.10.4, and therefore, have been omitted from the discussion of this section.

2.10.4 An Optimal Convex-Hull Construction Algorithm

The problem of computing the convex hull of a point-set [9, 18, 119, 172] is not only fundamental to many practical applications, but is also a mean for the solution of apparently unrelated issues arising in computational geometry. The importance of efficient convex-hull algorithms is apparent in many applications; for example, in computer graphics, computer-aided design, image processing, robotics, pattern recognition and statistics [119, 172]. The *convex-hull* problem may be concisely defined as follows.

Problem Given a set of n two-dimensional points $U = \{u_1, u_2, \dots, u_n\}$, determine the smallest convex domain containing the entire set and give a complete description of its boundary.

Unfortunately, this simple definition is not of a constructive nature. Thus, appropriate notions must be identified that are conducive to algorithm development.

In the remainder of this section we present a deterministic BSP algorithm for the problem of computing the convex hull of n two-dimensional points. The algorithm is based on an efficient deterministic parti-

tioning scheme [3, 58, 100] and achieves optimality in computation for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic convex-hull construction algorithm is based on the fundamental algorithms of the previous sections and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, processors sort (with respect to the x -coordinate), by means of some optimal deterministic sorting algorithm, the points in their possession and each processor constructs, by means of some optimal sequential convex-hull construction algorithm, the convex hull of its points. Subsequently, each processor computes, by means of an optimal binary-like search process, the upper and lower lines of tangency between its convex hull and the convex hulls of all other processors. Finally, each processor computes its (possibly empty) contribution to the convex hull (the aforementioned computation is realized by employing the upper and lower lines of tangency).

Our main result is summarized in the following theorem.

Theorem 2.10.1 *There exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points that for all integers n and p such that $p = o(\sqrt{n/\lg n})$, requires time $C_{srt}(n, p) + (c_* + o(1)) \cdot ((n/p) \cdot \lg(n/p))$ and $M_{srt}(n, p) + L \cdot O(1) + g \cdot O(n/p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.10.1 follows easily as a consequence of these results.

Optimality Claim 2.10.1 *Theorem 2.10.1 implies that there exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points, that for all integers n , p , L and g such that $p = o(\sqrt{n/\lg n})$, $L = o((n/p) \cdot \lg n)$ and $g = o(\lg n)$, has $\pi = 1 + c_{srt}/c_* + o(1)$ and $\mu = o(1)$.*

Lemma 2.10.1 *Let $\{U_1, U_2, \dots, U_k\}$ be a set of k disjoint two-dimensional point sets. Then, the convex hull resulting from the union of $CH(U_1), CH(U_2), \dots, CH(U_k)$ is equivalent to the convex hull of the union of $U_1, U_2, \dots, U_k$; that is, $CH(U_1) \cup CH(U_2) \cup \dots \cup CH(U_i) = CH(U_1 \cup U_2 \cup \dots \cup U_i)$.*

Proof The correctness of the above lemma follows trivially from the convex-hull definition. $\square$

The aforementioned result implies that the convex hull of $U_1 \cup U_2 \cup \dots \cup U_k$ is invariant under permutations of $U_1, U_2, \dots, U_k$ and possesses the inclusion property. The procedure to manage the convex-hull computation is presented in the following result.

Proposition 2.10.1 *There exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points, that for any integers n and p such that $p = O(n)$, requires time,*

$$C_{c_hull}(n, p) = c_* \cdot \frac{n}{p} \cdot \lg \frac{n}{p} + C_{srt}(n, p) + O\left(\frac{n}{p} + p \cdot \left\lceil \frac{n}{p^3} \right\rceil \cdot \frac{\lg(n/p)}{\lg(\lceil n/p^3 \rceil + 1)}\right)$$

and

$$M_{c_hull}(n, p) = L \cdot O\left(\frac{\lg(n/p)}{\lg(\lceil n/p^3 \rceil + 1)}\right) + g \cdot O\left(p \cdot \left\lceil \frac{n}{p^3} \right\rceil \cdot \frac{\lg(n/p)}{\lg(\lceil n/p^3 \rceil + 1)}\right) + M_{srt}(n, p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{c_hull}(n, p) = C_{c_hull}(n, p) + M_{c_hull}(n, p)$.

Proof We assume that the input is evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm.

Procedure CONVEX_HULL

begin

[**Stage 1**] The input set is orderly (by x -coordinate) and evenly distributed amongst the processors. Let the result of this distribution be set V_i on processor p_i , $0 \leq i < p$.

[**Analysis of Stage 1**] The cost of this stage is that of parallel sorting by one of the algorithms of section 2.6. Thus, this stage requires time $T_{srt}(n, p)$.

[**Stage 2**] Each processor p_i , $0 \leq i < p$, solves the convex-hull problem for the points in V_i . Let the result of this computation be convex-hull chain $CH(V_i)$ on processor p_i .

[**Analysis of Stage 2**] When this step is executed, each processor holds at most $\lceil n/p \rceil$ points. Thus, this stage involves sequentially solving the convex-hull problem on $\lceil n/p \rceil$ points by some deterministic algorithm [172] and requires time $c_* \cdot \lceil n/p \rceil \cdot \lg \lceil n/p \rceil$.

[**Stage 3**] Each processor p_i , $0 \leq i < p$, computes the upper and lower lines of tangency between $CH(V_i)$ and $CH(V_j)$, for all $0 \leq j < p$.

[**Analysis of Stage 3**] By employing a procedure reminiscent to that of [169] – a binary search procedure that requires at each step a point, its successor and its predecessor – this stage can be shown to require time $O(p \cdot \lceil n/p^3 \rceil \cdot (\lg \lceil n/p \rceil / \lg (\lceil n/p^3 \rceil + 1)))$ and $L \cdot O(\lg \lceil n/p \rceil / \lg (\lceil n/p^3 \rceil + 1)) + g \cdot O(p \cdot \lceil n/p^3 \rceil \cdot (\lg \lceil n/p \rceil / \lg (\lceil n/p^3 \rceil + 1)))$, for computation and communication respectively.³¹

[**Stage 4**] Each processor p_i , $0 \leq i < p$, computes its (possibly empty) contribution to the convex hull.

[**Analysis of Stage 4**] The (possibly empty) contribution of each processor to the convex hull is determined by selecting the sequence of points on the upper chain between the leftmost point of tangency u_r of a right tangent v_r and the rightmost point of tangency u_l of a left tangent v_l [100] (the process for the lower chain is similar). As shown in figure 2.62 if the angle formed by v_l and v_r is less than π , none of the points between u_r and u_l is on the convex hull; otherwise, all the points between u_r and u_l are on the convex hull. This stage requires $O(n/p + p)$ time.

end

Analysis The correctness of the algorithm follows from its description and lemma 2.10.1, and is omitted. By summing up the computation and communication requirements of procedure CONVEX_HULL we obtain the stated time bound.

³¹The basic idea behind the procedure of stage 3 is for each processor (in each step) to read $\lceil n/p^3 \rceil$ evenly spaced points, along with their successors and predecessors, from each other processor. Then, the procedure of [169] can be shown to require $O(\lg \lceil n/p \rceil / \lg (\lceil n/p^3 \rceil + 1))$ steps, and the desired time bound follows trivially.

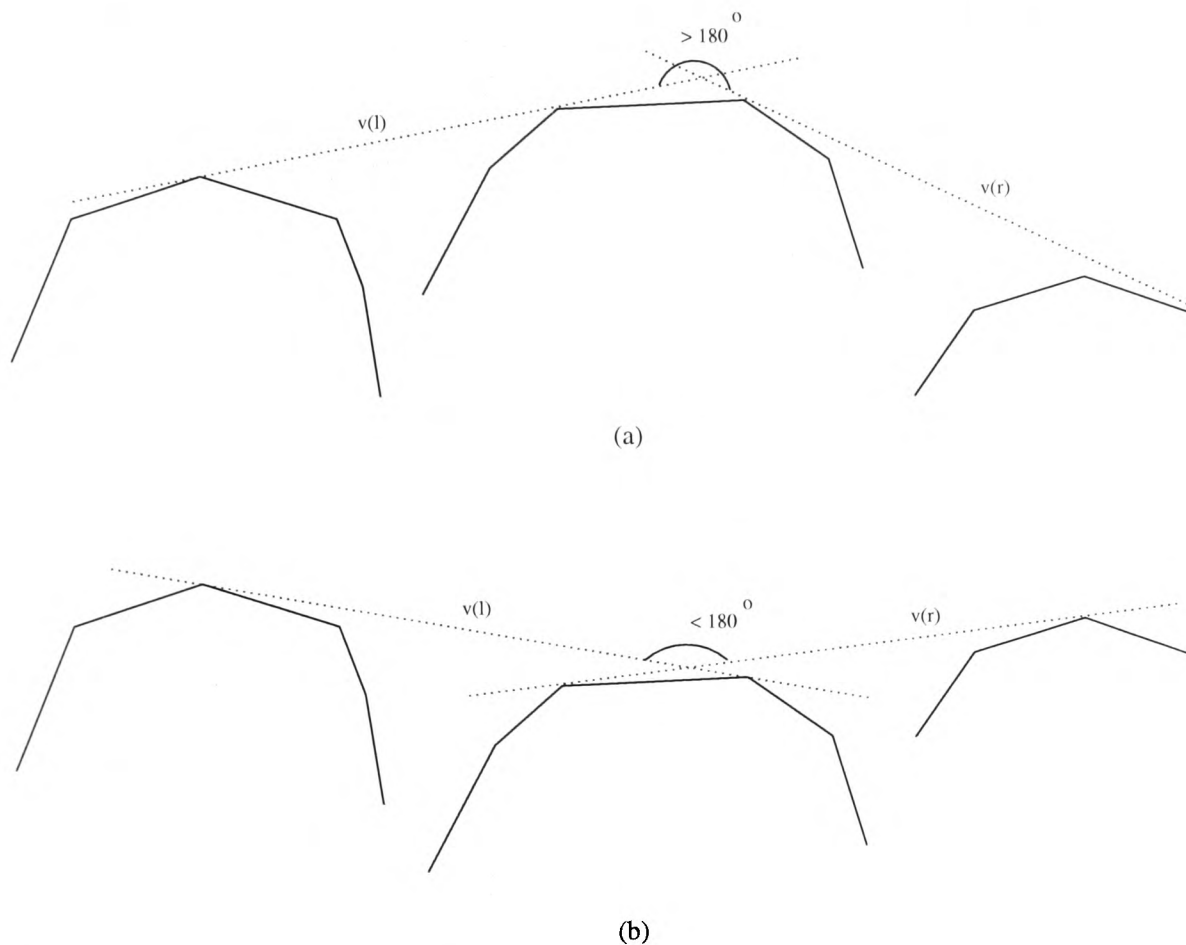


Figure 2.62: Two dimensional convex-hull merging. If the angle formed by v_l and v_r is less than π , none of the points between the tangency points of v_l and v_r is on the convex hull; otherwise, all the points are on the convex hull.

The proposition follows. $\square$

Proof (of theorem 2.10.1) The theorem follows easily from proposition 2.10.1. $\square$

2.10.4.1 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the convex-hull algorithm of section 2.10.4.

Implementation The prototype adopted for the *BSPlib* convex-hull procedure is outlined in figure 2.63.

```
void BSP_convex_hull (point *src, hull **trg, int n) ;
    src    - input point array.
    trg    - convex-hull structure.
    n      - input size.
```

Figure 2.63: Procedure for BSP convex-hull computation.

The aim of function (procedure) `BSP_convex_hull` is to provide a high-performance *BSPlib* convex-hull algorithm implementation.

Experimental Results The theoretical performance of the algorithm is summarized in theorem 2.10.1. To determine the practical significance of the aforementioned result we benchmarked our implementation (`BSP_convex_hull`) on two different machines with a varying number of processors and input sets

(consisting of uniformly distributed two-dimensional integer valued points – unless otherwise indicated). Figures 2.64 and 2.66 present the execution time of procedure `BSP_convex_hull` on the two machines. Similarly, figures 2.65 and 2.67 expose the efficiency of the corresponding timing results.

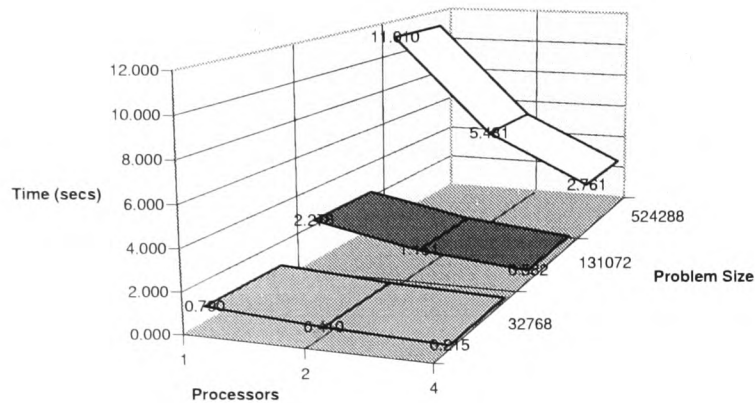


Figure 2.64: Execution time on a SGI Power Challenge for convex-hull computation – uniform distribution (U).

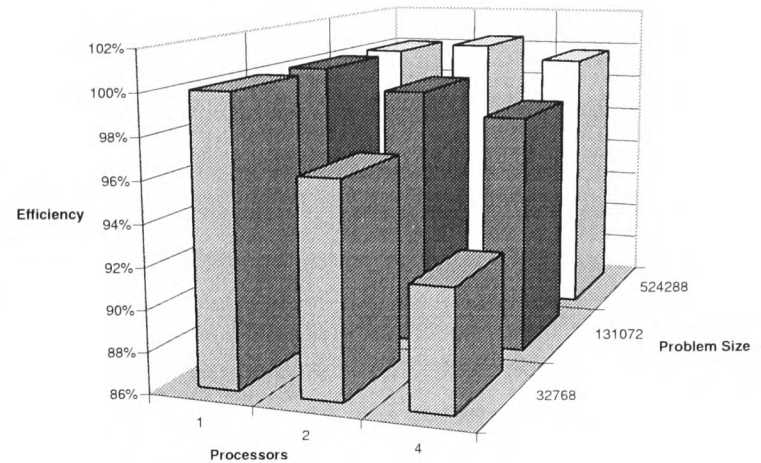


Figure 2.65: Efficiency on a SGI Power Challenge for convex-hull computation – uniform distribution (U).

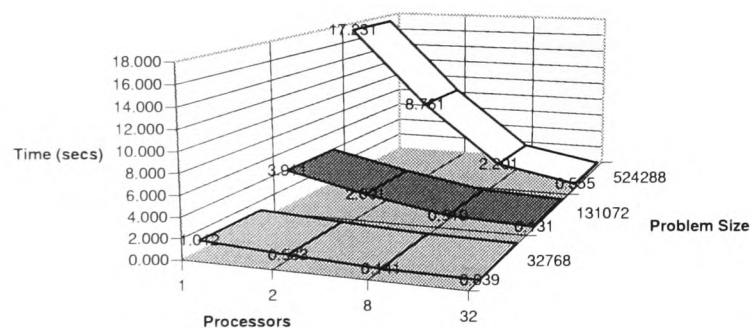


Figure 2.66: Execution time on a Cray T3D for convex-hull computation – uniform distribution (U).

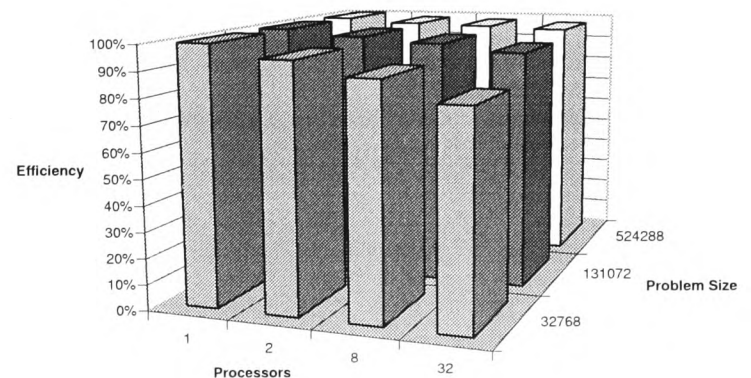


Figure 2.67: Efficiency on a Cray T3D for convex-hull computation – uniform distribution (U).

As shown in figures 2.64 and 2.66, the execution time of procedure `BSP_convex_hull` using a fixed number of processors scales with the input size (approximately as $n \cdot \lg n$). Accordingly, the execution time of procedure `BSP_convex_hull` for a fixed input size scales inversely with the number of processors (approximately as $1/p$). Both experimental results are in accordance with the theoretical analysis of section 2.10.4. In terms of efficiency, procedure `BSP_convex_hull` utilizes approximately (within a 5%):

- 90%–95% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 95%–100% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),
- 75%–100% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 95%–100% of the available processor resources on the Cray T3D machine (for large problem sizes).

These results should be interpreted with caution on the basis that the convex-hull algorithm of section 2.10.4 has been enhanced in the implementation by incorporating a heuristic point-elimination scheme [99] that improves the performance of the underlying convex-hull algorithm.

2.10.5 Optimal Geometric Searching and Dominance-Counting Algorithms

Geometric searching problems [9, 172] may be classified in two broad categories: *point-location* and *range-checking problems*. In point location problems we are given a partitioning of space P and an arbitrary point z , and we have to locate the relative position of z in P . On the other hand, in range checking problems we are given a partitioning of space (range) R and an arbitrary set of points n , and we have to determine how many points lie in R .

In the simplest case, searching is performed just once, in which case any preprocessing to speed up future queries would be wasteful. A one-time query is referred to as *single-shot* [172]. In many applications, however, numerous queries are repeated on the same set of data. Such queries are referred to as *repetitive mode queries* [172]. In this section we restrict ourselves to range searching single-shot queries; further results on point location and repetitive mode queries have been presented in the discussion of sub-section 2.9.6.4. The range searching problem we consider in this section, a variant of the *dominance-counting* problem, may be concisely be defined as follows.

Problem Given a *dominant*³² set of n two-dimensional points $U = \{u_1, u_2, \dots, u_n\}$ and a *dominated* set $\bar{U}$ of m two-dimensional points, determine, for each $u \in U$, the number of points in $\bar{U}$ that are dominated by u .

Unfortunately, this simple definition is not of a constructive nature. Thus, appropriate notions must be identified that are conducive to algorithm development.

In the remainder of this section we present a deterministic BSP algorithm for the problem of computing the dominance counting of n two-dimensional points. The algorithm is based on an efficient deterministic partitioning scheme [58] and achieves optimality in computation for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic dominance-counting algorithm is based on the fundamental algorithms of the previous sections and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, processors sort (with respect to the x -coordinate), by means of some optimal deterministic sorting algorithm, the items in their possession and each processor solves, by means of some optimal sequential dominance-counting algorithm, the dominance-counting problem for its items. Subsequently, processors sort (with respect to the y -coordinate), by means of some optimal deterministic sorting algorithm, the items in their possession and each processor solves, by means of some optimal sequential dominance-counting algorithm, the dominance-counting problem for its items. Additionally, each processor solves, by means of some optimal sequential dominance-counting algorithm, the dominance-counting problem for the sets resulting by the intersection of the two sorted sets (with respect to x - and y -coordinates). Finally, the dominance-counting problem solution derives by appropriately combining the aforementioned independent solutions.

Our main result is summarized in the following theorem.

³²The *dominance* is a relation as follows: two-dimensional point $u = (x_1, x_2)$ dominates two-dimensional point $\bar{u} = (\bar{x}_1, \bar{x}_2)$ if $x_1 > \bar{x}_1$ and $x_2 > \bar{x}_2$.

Theorem 2.10.2 *There exists a p -processor BSP algorithm for solving the dominance-counting problem on two sets consisting of n two-dimensional points that for all integers n and p such that $p = O(\sqrt{n})$, requires time $2 \cdot C_{srt}(n, p) + (3 \cdot c_* + o(1)) \cdot ((n/p) \cdot \lg(n/p))$ and $2 \cdot M_{srt}(n, p) + L \cdot O(1) + g \cdot O(p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.10.2 follows trivially as a consequence of these results.

Optimality Claim 2.10.2 *Theorem 2.10.2 implies that there exists a p -processor BSP algorithm for solving the dominance-counting problem on two sets consisting of n two-dimensional points, that for all integers n , p , L and g such that $p = O(\sqrt{n})$, $L = o((n/p) \cdot \lg n)$ and $g = o(\lg n)$, has $\pi = 3 + 2 \cdot (c_{srt}/c_*) + o(1)$ and $\mu = o(1)$.*

Lemma 2.10.2 ([58]) *Consider dominant and dominated sets U and $\bar{U}$ respectively. Let V and H be a vertical and a horizontal slab of $\bar{U}$, delimited from left and below by the lines v and h respectively. For every point $u \in U$ that belongs to the intersection I of V and H , we have that $D(u, \bar{U}) = D(u, V) + D(u, H) - D(u, I) + D(v \cup h, \bar{U})$, where $D(v \cup h, \bar{U})$ denotes the number of points in $\bar{U}$ that are both below h and to the left of v (or equivalently, the number of points in $\bar{U}$ dominated by point $v \cup h$).*

The procedure to manage the dominance computation is presented in the following result.

Proposition 2.10.2 *There exists a p -processor BSP algorithm for solving the dominance-counting problem on two sets consisting of n two-dimensional points, that for any integers n and p such that $p = O(n)$, requires time,*

$$C_{dom_cnt}(n, p) = 3 \cdot c_* \cdot \frac{n}{p} \cdot \lg \frac{n}{p} + 2 \cdot C_{srt}(n, p) + O\left(\frac{n}{p} + p\right)$$

and

$$M_{dom_cnt}(n, p) = L \cdot O(1) + g \cdot O(p) + 2 \cdot M_{srt}(n, p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{dom_cnt}(n, p) = C_{dom_cnt}(n, p) + M_{dom_cnt}(n, p)$.

Proof We assume that the input is evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm.

Procedure DOMINANCE_COUNT

begin

[Stage 1] The input set is orderly (by x -coordinate) and evenly distributed amongst the processors. Let the result of this distribution be set V_i on processor p_i , $0 \leq i < p$.

[Analysis of Stage 1] The cost of this stage is that of parallel sorting by one of the algorithms of section 2.6. Thus, this stage requires time $T_{srt}(n, p)$.

[**Stage 2**] Each processor p_i , $0 \leq i < p$, solves the dominance-counting problem for the points in V_i .

[**Analysis of Stage 2**] When this step is executed, each processor holds at most $\lceil n/p \rceil$ points. Thus, this stage involves sequentially solving the dominance-counting problem on $\lceil n/p \rceil$ points by some deterministic algorithm [172] and requires time $c_* \cdot \lceil n/p \rceil \cdot \lg \lceil n/p \rceil$.

[**Stage 3**] Each processor p_i , $0 \leq i < p$, computes the maximum x -coordinate of the points in its possession (vertical line v_i) and communicates v_i to all other processors.

[**Analysis of Stage 3**] This stage takes time $O(1)$ and $L+g \cdot O(p)$, for computation and communication respectively.

[**Stage 4**] The input set is orderly (by y -coordinate) and evenly distributed amongst the processors. Let the result of this distribution be set H_j on processor p_j , $0 \leq j < p$.

[**Analysis of Stage 4**] The cost of this stage is that of parallel sorting by one of the algorithms of section 2.6. Thus, this stage requires time $T_{srt}(n, p)$.

[**Stage 5**] Each processor p_j , $0 \leq j < p$, solves the dominance-counting problem for the points in H_j .

[**Analysis of Stage 5**] When this step is executed, each processor holds at most $\lceil n/p \rceil$ points. Thus, this stage involves sequentially solving the dominance-counting problem on $\lceil n/p \rceil$ points by some deterministic algorithm [172] and requires time $c_* \cdot \lceil n/p \rceil \cdot \lg \lceil n/p \rceil$.

[**Stage 6**] Each processor p_j , $0 \leq j < p$, determines $H_j \cup V_i$, for all $0 \leq i < p$.

[**Analysis of Stage 6**] The cost of this stage is that of merging H_j with $\{v_0, v_1, \dots, v_{p-1}\}$. Thus, this stage requires time $O(n/p + p)$.

[**Stage 7**] Each processor p_j , $0 \leq j < p$, solves the dominance-counting problem for the points in $H_j \cup V_i$, for all $0 \leq i < p$.

[**Analysis of Stage 7**] This stage involves sequentially solving p disjoint dominance-counting problems on a total of at most $\lceil n/p \rceil$ points by some deterministic algorithm [172], and therefore, requires time at most $c_* \cdot \lceil n/p \rceil \cdot \lg \lceil n/p \rceil$.

[**Stage 8**] Each processor p_j , $0 \leq j < p$, computes n_{ij} – the number of points in H_j that belong to the dominated set and they are to the left of v_i , for all $0 \leq i < p$ – and communicates n_{ij} to processor p_{i+1} , for all $0 \leq i < p - 1$.

[**Analysis of Stage 8**] The cost of this stage is that of merging H_j with $\{v_0, v_1, \dots, v_{p-1}\}$. Thus, this stage requires time $O(n/p + p)$ and $L + g \cdot O(p)$, for computation and communication respectively.

[**Stage 9**] Each processor p_j , $0 \leq j < p$, solves the dominance-counting problem for the points in H_j by employing the formulae of lemma 2.10.2.

[**Analysis of Stage 9**] This stage requires time $O(n/p)$.

end

Analysis The correctness of the algorithm follows from its description and lemma 2.10.2. By summing up the computation and communication requirements of procedure DOMINANCE_COUNT we obtain the stated time bound.

The proposition follows. $\square$

Proof (of theorem 2.10.2) The theorem follows easily from proposition 2.10.2. $\square$

2.10.5.1 Applications of Dominance Counting in Rectangle Inclusion

A range-searching problem that can be handled efficiently using dominance counting is the *rectangle-inclusion* problem.

Problem Given a set of n rectangular range queries $Q = \{q_1, q_2, \dots, q_n\}$ and a set $\bar{U}$ of m two-dimensional points, determine, for each $q \in Q$, the number of points in $\bar{U}$ contained in the query domain of q .

In the remainder of this section we present a deterministic BSP algorithm for the problem of computing the rectangle-inclusion of n rectangular range queries and n two-dimensional points. The algorithm is based on the dominance-counting algorithm of the previous section and achieves optimality in computation for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic rectangle-inclusion algorithm is an immediate application of the dominance-counting algorithm of section 2.10.5.

Our main result is summarized in the following theorem.

Theorem 2.10.3 *There exists a p -processor BSP algorithm for solving the rectangle-inclusion problem on n rectangle range queries and n two-dimensional points that for all integers n and p such that $p = O(n)$, requires time $C_{dom_cnt}(5 \cdot n, p) + O(n/p)$ and $M_{dom_cnt}(5 \cdot n, p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a technical result. Then, theorem 2.10.3 follows trivially as a consequence of this result.

Lemma 2.10.3 *The rectangle-inclusion problem on a set of n rectangular range queries Q and a set $\bar{U}$ of n two-dimensional points can be solved in $O(n/p)$ time, on the BSP model, once the dominance-counting problem has been solved for the points in dominant set Q (vertices of rectangles) and dominated set $\bar{U}$.*

Proof Let r be a rectangle with vertices (v_1, v_2, v_3, v_4) ordered counterclockwise starting from the upper right corner. The number of points contained in rectangle r is given by,

$$D(v_1, \bar{U}) - D(v_2, \bar{U}) - D(v_4, \bar{U}) + D(v_3, \bar{U}), \quad (2.16)$$

where $D(v, \bar{U})$ denotes the number of points in $\bar{U}$ dominated by point v . Hence, such a number can be computed in $O(1)$ sequential time for each rectangle. Since the number of rectangles is n , the rectangle

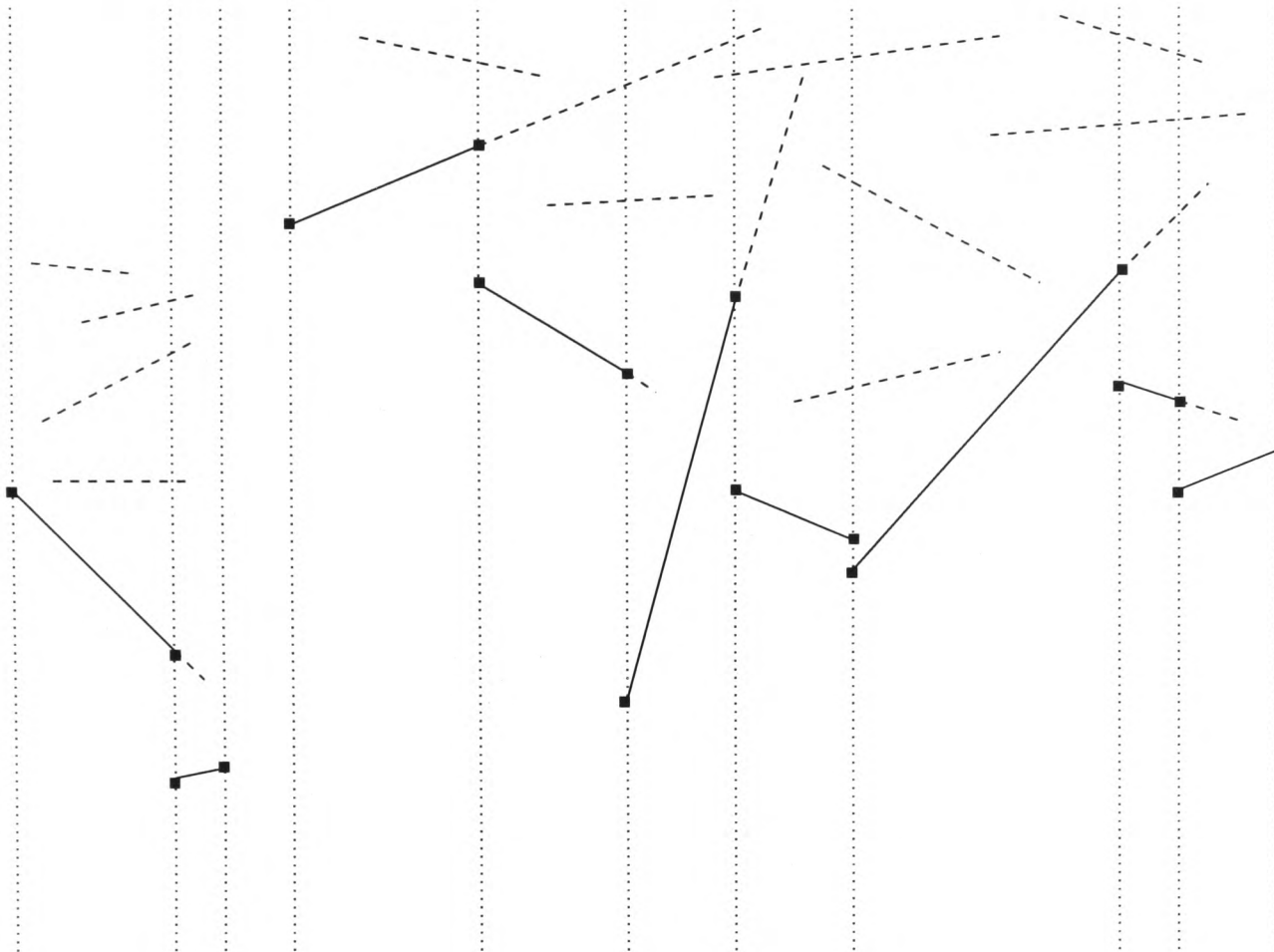


Figure 2.68: Two-dimensional lower envelope; visible portions of non-intersecting line segments from $-\infty$.

inclusion problem can be computed in $O(n/p)$ time, once the dominance-counting problem has been solved for the points in dominant set Q (vertices of rectangles) and dominated set $\bar{U}$. $\square$

Proof (of theorem 2.10.3) The theorem follows easily from lemma 2.10.3 and proposition 2.10.2. $\square$

2.10.6 Optimal Visibility and Lower-Envelope Construction Algorithms

Given a set S of “forbidden” curves and a point u in E^2 , a point v is *visible* from u if the line segment $\overline{uv}$ does not intersect any forbidden curve in S [9, 18, 119, 172]. Normally, visibility is defined with respect to a point u . In the plane, if S consists of line segments, the locus of points visible from u form a (possibly unbounded) star-shaped polygon, called the *visibility polygon* [9, 18, 119, 172]. Visibility problems occur in a multiplicity of applications in graphics, scene analysis and robotics. The visibility problem we consider in this section, a variant of the *lower-envelope* problem, may be concisely be defined as follows.

Problem Given a set of n non-intersecting line segments $S = \{s_1, s_2, \dots, s_n\}$ – except possibly at their endpoints – and a point v at negative infinity, determine, for each $s \in S$, the visible portion of s from p .

Unfortunately, this simple definition is not of a constructive nature. Thus, appropriate notions must be identified that are conducive to algorithm development.³³

In the remainder of this section we present a deterministic BSP algorithm for the problem of computing the lower envelope of n two-dimensional non-intersecting line segments. The algorithm is based on an

³³Obviously, an algorithm for the computation of the lower envelope of n non-intersecting line segments can be readily employed to handle the case that v is any finite point, that is, the view of segments being ordered by x -coordinate is replaced by the view that segments are ordered radially around v [18]. Therefore, it suffices to compute the lower envelope of a set of non-intersecting line segments to establish a method for determining the visibility from a given point.

efficient deterministic partitioning scheme [58] and achieves optimality in computation for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal deterministic lower-envelope algorithm is based on the fundamental algorithms of the previous sections and is non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Initially, each processor solves, by means of some optimal sequential lower-envelope algorithm, the lower-envelope problem for the items in its possession. Subsequently, processors sort (with respect to the x -coordinate), by means of some optimal deterministic sorting algorithm, the items in their possession and each processor solves, by means of some optimal sequential lower-envelope algorithm, the lower-envelope problem for its items along with at most p additional items belonging to the intersections of the initially constructed lower-envelopes and the p newly formed sets. Finally, the lower-envelope problem solution derives by appropriately combining the aforementioned independent solutions.

Our main result is summarized in the following theorem.

Theorem 2.10.4 *There exists a p -processor BSP algorithm for solving the lower-envelope problem on a set of n non-intersecting line segments that for all integers n and p such that $p = O(\sqrt{n})$, requires time $C_{srt}(2 \cdot n, p) + (3 \cdot c_* + o(1)) \cdot ((n/p + p/3) \cdot \lg(n/p))$ and $M_{srt}(2 \cdot n, p) + L \cdot O(1) + g \cdot O(p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.10.4 follows trivially as a consequence of these results.

Optimality Claim 2.10.3 *Theorem 2.10.4 implies that there exists a p -processor BSP algorithm for solving the lower-envelope problem on a set of n non-intersecting line segments, that for all integers n , p , L and g such that $p = O(\sqrt{n})$, $L = o((n/p) \cdot \lg n)$ and $g = o(\lg n)$, has $\pi = 3 + 2 \cdot (c_{srt}/c_*) + o(1)$ and $\mu = o(1)$.*

Lemma 2.10.4 *Let $\{S_1, S_2, \dots, S_k\}$ be a set of k disjoint non-intersecting line segment sets. Then, the lower envelope resulting from the union of $LE(S_1), LE(S_2), \dots, LE(S_k)$ is equivalent to the lower envelope of the union of $S_1, S_2, \dots, S_k$; that is, $LE(S_1) \cup LE(S_2) \cup \dots \cup LE(S_k) = LE(S_1 \cup S_2 \cup \dots \cup S_k)$.*

Proof The correctness of the above lemma follows trivially from the lower-envelope definition. $\square$

The aforementioned result implies that the lower envelope of $S_1 \cup S_2 \cup \dots \cup S_k$ is invariant under permutations of $S_1, S_2, \dots, S_k$, and possesses the inclusion property.

Lemma 2.10.5 *The lower envelope of a set S of n non-intersecting line segments is x -monotonic – a line orthogonal to the x -axis intersects at most one segment in S , except at endpoints – and consists of at most $2 \cdot n - 1$ non-intersecting line segments.*

Proof The x -monotonicity of the lower envelope follows directly from its definition. Furthermore, it is straightforward to verify that $|LE(S)| \leq 2 \cdot n - 1$. $\square$

The procedure to manage the lower envelope is presented in the following result.

Proposition 2.10.3 *There exists a p -processor BSP algorithm for solving the lower-envelope problem on a set of n non-intersecting line segments, that for any integers n and p such that $p = O(n)$, requires time,*

$$C_{low_env}(n, p) = 3 \cdot c_* \cdot \left(\frac{n}{p} + \frac{p}{3} \right) \cdot \lg \left(\frac{n}{p} + p \right) + C_{srt}(2 \cdot n, p) + O \left(\frac{n}{p} + p \right)$$

and

$$M_{low_env}(n, p) = L \cdot O(1) + g \cdot O(p) + M_{srt}(2 \cdot n, p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{low_env}(n, p) = C_{low_env}(n, p) + M_{low_env}(n, p)$.

Proof We assume that the input is evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm.

Procedure LOWER_ENVELOPE

begin

[**Stage 1**] Each processor p_i , $0 \leq i < p$, solves the lower-envelope problem for the line segments in its possession. Let the result of this computation be x -monotonic chain $LE(S_i)$ on processor p_i .

[**Analysis of Stage 1**] When this step is executed, each processor holds at most $\lceil n/p \rceil$ points. Thus, this stage involves sequentially solving the lower-envelope problem on $\lceil n/p \rceil$ non-intersecting line segments by some deterministic algorithm [172], and therefore, requires time $c_* \cdot \lceil n/p \rceil \cdot \lg \lceil n/p \rceil$. By way of lemma 2.10.5, $|\cup_i LE(S_i)| \leq 2 \cdot n - 1$.

[**Stage 2**] Set $\cup_i LE(S_i)$, $0 \leq i < p$, is orderly (by x -coordinate of the right endpoint) and evenly distributed amongst the processors. Let the result of this distribution be set L_i on processor p_i , $0 \leq i < p$.

[**Analysis of Stage 2**] The cost of this stage is that of parallel sorting by one of the algorithms of section 2.6. Thus, this stage requires time $T_{srt}(2 \cdot n, p)$.

[**Stage 3**] Each processor p_i , $0 \leq i < p$, computes the maximum x -coordinate (of the right endpoint) of the segments in its possession (vertical line v_i) and communicates v_i to all other processors.

[**Analysis of Stage 3**] This stage takes time $O(1)$ and $L + g \cdot O(p)$, for computation and communication respectively.

[**Stage 4**] Each processor p_j , $0 \leq j < p$, communicates segment $s \in LE(S_j)$ to processor p_i , for all $0 \leq i < p$, if s intersects v_i . Let R_i be the set of segments communicated to processor p_i . Then, by way of lemma 2.10.5, $|R_i| \leq p$.

[**Analysis of Stage 4**] The cost of this stage is that of merging $LE(S_j)$ with $\{v_0, v_1, \dots, v_{p-1}\}$. Thus, this stage requires time $O(n/p + p)$ and $L + g \cdot O(p)$, for computation and communication respectively.

[**Stage 5**] Each processor p_i , $0 \leq i < p$, solves the lower-envelope problem for the line segments in $LE(L_i) \cup R_i$.

[Analysis of Stage 5] When this step is executed, each processor holds at most $2 \cdot \lceil n/p \rceil + p$ points. Thus, this stage involves sequentially solving the lower-envelope problem on $2 \cdot \lceil n/p \rceil + p$ non-intersecting line segments by some deterministic algorithm [172] and requires time $c_* \cdot (2 \cdot \lceil n/p \rceil + p) \cdot \lg(2 \cdot \lceil n/p \rceil + p)$.

end

Analysis The correctness of the algorithm follows from its description and lemmas 2.10.4 and 2.10.5. In particular, after the completion of stage 2 the segments in $\cup_i L_i$, $0 \leq i < p$, are ordered by the x -coordinate of their right endpoints, and hence, the lower envelope $LE(L_i)$ of some processor p_i can be affected by those segments in processors $p_{i+1}, p_{i+2}, \dots, p_{p-1}$ that have their left endpoints to the left of the rightmost endpoint in L_i . Obviously, since after the completion of stage 1 the resulting chains are x -monotonic (refer to lemma 2.10.5) there can be at most p such segments – one per x -monotonic chain. Stage 4 identifies those segments, and therefore, in stage 5 each processor correctly computes the lower envelope of its share of the global problem. By summing up the computation and communication requirements of procedure LOWER_ENVELOPE we obtain the stated time bound.

Remark The same algorithmic idea can be employed to solve the problem for possibly intersecting line segments, by replacing stages 1 and 5 of the above algorithm by a sequential lower-envelope algorithm that can handle intersecting line segments [110]. However, since the size of the resulting problem is $\Omega(n)$, some extra storage is (possibly) necessary – the size of the lower envelope for possibly intersecting line segments is $O(n \cdot \alpha(n))$, where $\alpha(\cdot)$ is the slow growing inverse Ackermann function [58].

The proposition follows. $\square$

Proof (of theorem 2.10.4) The theorem follows easily from proposition 2.10.3. $\square$

2.10.7 Improving Computational Geometry Algorithms

The computational geometry methods we have presented in this section require polynomial amount of parallel slackness to achieve optimality – typically $p = O(\sqrt{n})$. In this section we argue that by employing recursive (or iterative) techniques the algorithms achieve optimality for values of p much closer to n . Unfortunately, the details underlying these techniques are problem-bound, and therefore, there is no (trivial) generalization. The main idea behind these recursive techniques can be summarized as follows. The global problem is partitioned into a suitable number of disjoint sub-problems and each of these sub-problems is solved recursively (in parallel). Then, by combining the solutions of the disjoint sub-problems each processor determines its share of the global solution. The above is necessarily an oversimplification. The actual algorithms do more than just those simplistic steps. The major challenge lies in formulating partitioning and combining strategies that ensure optimality. In the remainder of this section we outline such a scheme for the convex-hull problem. Similar ideas can be employed to reduce the amount of parallel slackness required to achieve optimality for the other computational geometry problems presented in this section.

2.10.7.1 Improving Convex-Hull Construction I

The algorithmic scheme that follows derives from the ideas of section 2.10.4. In particular, we outline an iterative version of the algorithm of section 2.10.4. The notable change in the iterative version is

identified in the implementation of stage 3 of algorithm CONVEX_HULL (refer to proposition 2.10.1); a minor modification is also required in the implementation of stage 4.

Informal Description of Algorithm The optimal deterministic convex-hull construction algorithm is an immediate iterative adaptation of the convex-hull construction algorithm of section 2.10.4.

Our main result is summarized in the following theorem.

Theorem 2.10.5 *For arbitrarily small constant $\zeta > 0$ and any $\alpha > 0$, there exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points that for all integers n and p such that $p = O(n \cdot (\lg(n/p)/\lg^2 n)^{1+\alpha})$, requires time $C_{srt}(n, p) + (c_* + o(1)) \cdot ((n/p) \cdot \lg(n/p))$ and $M_{srt}(n, p) + L \cdot O(\lg^2 n / \lg(n/p)) + g \cdot O(n/p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.10.5 follows easily as a consequence of these results.

Optimality Claim 2.10.4 *Theorem 2.10.5 implies that there exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points, that for all integers n, p, L and g such that $p = O(n \cdot (\lg(n/p)/\lg^2 n)^{1+\alpha})$, $L = o((n/p) \cdot (\lg(n/p)/\lg n))$ and $g = o(\lg n)$, has $\pi = 1 + c_{srt}/c_* + o(1)$ and $\mu = o(1)$.*

The details of the algorithm follow. Stages 1–2 of algorithm CONVEX_HULL remain unmodified. For some constant $0 < \epsilon < 1$, in stage 3, the upper and lower lines of tangency between at most $\lceil (n/p)^\epsilon \rceil$ consecutive convex hulls are computed. In stage 4, the contribution of these $\lceil (n/p)^\epsilon \rceil$ consecutive convex hulls to the resulting convex hull is computed. It is easy to verify that the desired convex hull is computed within $\lceil \lg p / \lg \lceil (n/p)^\epsilon \rceil \rceil$ iterations of stages 3–4. It remains to establish the time complexity of stages 3–4. Along the lines of the analysis of algorithm CONVEX_HULL, stage 3 can be easily shown to require time

$$\sum_{i=1}^{\lceil \lg p / \lg \lceil (n/p)^\epsilon \rceil \rceil} \left(O \left(\left(\frac{n}{p} \right)^\epsilon \cdot \lg \left(\frac{n}{p} \right)^{1+(i-1) \cdot \epsilon} \right) \right) = O \left(\left(\frac{n}{p} \right)^\epsilon \cdot \frac{\lg^2 n}{\lg(n/p)} \right)$$

and

$$\sum_{i=1}^{\lceil \lg p / \lg \lceil (n/p)^\epsilon \rceil \rceil} \left(L \cdot O \left(\lg \left(\frac{n}{p} \right)^{1+(i-1) \cdot \epsilon} \right) + g \cdot O \left(\left(\frac{n}{p} \right)^\epsilon \cdot \lg \left(\frac{n}{p} \right)^{1+(i-1) \cdot \epsilon} \right) \right) = \\ L \cdot O \left(\frac{\lg^2 n}{\lg(n/p)} \right) + g \cdot O \left(\left(\frac{n}{p} \right)^\epsilon \cdot \frac{\lg^2 n}{\lg(n/p)} \right),$$

for computation and communication respectively. Similarly, stage 4 can be shown to require time $O(n/p)$ and $L \cdot O(1) + g \cdot O((n/p)^\epsilon)$ for computation and communication respectively. This result completes the analysis of the iterative version of algorithm CONVEX_HULL. We have therefore established the following analogue of proposition 2.10.1.

Proposition 2.10.4 *For any constant $0 < \epsilon < 1$, there exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points, that for any integers n and p such that $p = O(n)$, requires time,*

$$C_{c_hull}(n, p) = c_* \cdot \frac{n}{p} \cdot \lg \frac{n}{p} + C_{srt}(n, p) + O\left(\frac{n}{p} + \left(\frac{n}{p}\right)^\epsilon \cdot \frac{\lg^2 n}{\lg(n/p)}\right)$$

and

$$M_{c_hull}(n, p) = L \cdot O\left(\frac{\lg^2 n}{\lg(n/p)}\right) + g \cdot O\left(\left(\frac{n}{p}\right)^\epsilon \cdot \frac{\lg^2 n}{\lg(n/p)}\right) + M_{srt}(n, p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{c_hull}(n, p) = C_{c_hull}(n, p) + M_{c_hull}(n, p)$.

Proof (of theorem 2.10.5) The theorem follows easily from proposition 2.10.4. $\square$

2.10.7.2 Improving Convex-Hull Construction II

The algorithmic scheme that follows derives from the ideas of section 2.10.4. In particular, we outline a non-trivial recursive version of the algorithm of section 2.10.4. The notable change in the recursive version is identified in the implementation of stage 3 of algorithm CONVEX_HULL (refer to proposition 2.10.1); a minor modification is also required in the implementation of stage 4.

Informal Description of Algorithm The optimal deterministic convex-hull construction algorithm is an immediate recursive adaptation of the convex-hull construction algorithm of section 2.10.4 – with the exception that the simple binary-like search process (in the computation of the lines of tangency) is substituted by a non-trivial re-working of the algorithm of [15].

Our main result is summarized in the following theorem.

Theorem 2.10.6 *There exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points that for all integers n and p such that $p = O(n)$, requires time $C_{srt}(n, p) + O((n/p) \cdot \lg n)$ and $M_{srt}(n, p) + L \cdot O(\lg n / \lg(n/p)) + g \cdot O((n/p) \cdot (\lg n / \lg(n/p)))$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 2.10.6 follows easily as a consequence of these results.

The details of the algorithm follow. The basic idea is to partition – by way of sorting – the input into $\min\{p, \lceil n^{1/6} \rceil\}$ contiguous subsets, each consisting of at most $\max\{\lceil n/p \rceil, \lceil n^{5/6} \rceil\}$ points. The convex hull of each of these subsets is recursively computed (in parallel), and subsequently, the $n^{1/6}$ disjoint convex hulls are merged by employing a procedure similar to that of stages 3–4 of algorithm CONVEX_HULL (refer to section 2.10.4). In [15], it is shown that given two convex hulls of size $k^{O(1)}$ there exists a k -processor CREW PRAM algorithm for computing the upper and lower lines of tangency that requires time $O(1)$. In stage 3, we simulate the process of the algorithm of [15], by means of a constant number of broadcast, parallel-prefix and sorting operations – sorting is required to avoid concurrent read conflicts. Then, in stage 4, we complete the convex-hull computation by means of a parallel-prefix operation. It can be shown that each sorting operation involves at most $\lceil n^{1/3} \rceil$ items. The overall convex-hull merging process can be shown to require time

$$O\left(\frac{n}{p}\right) + O\left(C_{quad_srt}\left(\left\lceil n^{1/3} \right\rceil, p\right)\right) + O\left(C_{ppf}(p)\right)$$

and

$$L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + O\left(M_{quad_srt}\left(\left\lceil n^{1/3} \right\rceil, p\right)\right) + O\left(M_{bcast}^{\lceil n/p \rceil}(p)\right) + O\left(M_{ppf}(p)\right),$$

for computation and communication respectively, where T_{quad_srt} denotes the time complexity of the sorting algorithm of section 2.6.5.

This result completes the analysis of the recursive version of algorithm CONVEX_HULL. We have therefore established the following analogue of proposition 2.10.1.

Proposition 2.10.5 *There exists a p -processor BSP algorithm for solving the convex-hull problem on a set of n two-dimensional points, that for any integers n and p such that $p = O(n)$, requires time,*

$$\begin{aligned} C_{c_hull}(n, p) &= C_{c_hull}\left(\max\left\{\left\lceil \frac{n}{p} \right\rceil, \left\lceil n^{5/6} \right\rceil\right\}, \left\lfloor \frac{p}{\min\{p, \lceil n^{1/6} \rceil\}} \right\rfloor\right) + \\ &\quad O\left(\frac{n}{p}\right) + O\left(C_{quad_srt}\left(n^{1/3}, p\right)\right) + O\left(C_{ppf}(p)\right) \end{aligned}$$

and

$$\begin{aligned} M_{c_hull}(n, p) &= M_{c_hull}\left(\max\left\{\left\lceil \frac{n}{p} \right\rceil, \left\lceil n^{5/6} \right\rceil\right\}, \left\lfloor \frac{p}{\min\{p, \lceil n^{1/6} \rceil\}} \right\rfloor\right) + \\ &\quad L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + O\left(M_{quad_srt}\left(n^{1/3}, p\right)\right) + O\left(M_{bcast}^{\lceil n/p \rceil}(p)\right) + O\left(M_{ppf}(p)\right), \end{aligned}$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{c_hull}(n, p) = C_{c_hull}(n, p) + M_{c_hull}(n, p)$.

Proof (of theorem 2.10.6) The theorem follows easily from proposition 2.10.5 by solving the recurrence relation (with base case $p = 1$) and substituting for T_{quad_srt} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.6.2, 2.1.2 and 2.1.4, and letting $t = \lceil n/p \rceil$ in these expressions. $\square$

Remark Theorems 2.10.1 and 2.10.6 are asymptotically equivalent for the case $p = n^{1-\epsilon}$, for any positive constant $1/2 < \epsilon \leq 1$. The constant factors, however, associated with theorem 2.10.6 are considerably larger than those of theorem 2.10.1.

2.10.8 Other Optimal Computational Geometry Algorithms

A multiplicity of related computational problems may be tackled by employing techniques similar to those of sections 2.10.4, 2.10.5 and 2.10.6. The following problems have been already tackled and the results have been reported in [197]. In the remainder of this section we summarize the aforementioned results.

2.10.8.1 Intersection of Horizontal and Vertical Line Segments

Problem Given a *dominant* set of n horizontal line segments $S = \{s_1, s_2, \dots, s_n\}$ and a *dominated* set $\bar{S}$ of m vertical line segments, determine, for each $s \in S$, then number of segments in $\bar{S}$ that are intersected by s .

Theorem 2.10.7 *There exists a p -processor BSP algorithm for solving the intersection-of-horizontal-and-vertical-line-segments problem on a set of n horizontal and vertical line segments that for all integers n and p such that $p = O(n)$, requires time $C_{dom_cnt}(2 \cdot n, p) + O(n/p)$ and $M_{dom_cnt}(2 \cdot n, p)$, for computation and communication respectively.*

2.10.8.2 Three-Dimensional Maxima

Problem Given a set of n three-dimensional points $U = \{u_1, u_2, \dots, u_n\}$, determine, for each $u \in U$, the number of points in U that are dominated by u .

Theorem 2.10.8 *There exists a p -processor BSP algorithm for solving the three-dimensional maxima problem on n three-dimensional points that for all integers n and p such that $p = O(\sqrt{n})$, requires time $2 \cdot C_{srt}(n, p) + C_{dom_cnt}(n, p) + (3 \cdot c_* + o(1)) \cdot ((n/p + p/3) \cdot \lg(n/p))$ and $2 \cdot M_{srt}(n, p) + M_{dom_cnt}(n, p) + L \cdot O(1) + g \cdot O(p)$.*

2.10.8.3 All Nearest Neighbors

Problem Given a set of n two-dimensional points $U = \{u_1, u_2, \dots, u_n\}$, determine, for each $u \in U$, all the nearest neighbors of u .

Theorem 2.10.9 *There exists a p -processor BSP algorithm for solving the all-nearest-neighbors problem on n two-dimensional points that for all integers n and p such that $p = O(\sqrt{n})$, requires time $4 \cdot C_{srt}(n, p) + (2 \cdot c_* + o(1)) \cdot ((n/p + p/2) \cdot \lg(n/p))$ and $4 \cdot M_{srt}(n, p) + L \cdot O(1) + g \cdot O(p)$.*

2.10.8.4 All Furthest Neighbors

Problem Given a set of n two-dimensional points $U = \{u_1, u_2, \dots, u_n\}$, determine, for each $u \in U$, all the furthest neighbors of u .

Theorem 2.10.10 *There exists a p -processor BSP algorithm for solving the all-furthest-neighbors problem on n two-dimensional points that for all integers n and p such that $p = O(n)$, requires time $C_{c_hull}(n, p) + h \cdot (C_{ppf}(p) + O(n/p))$ and $M_{c_hull}(n, p) + h \cdot M_{ppf}(p)$, where h denotes the number of edges in the convex hull of the n two-dimensional points.*

2.10.8.5 Minimum-Area Enclosing Rectangle

Problem Given a set of n two-dimensional points $U = \{u_1, u_2, \dots, u_n\}$, determine the minimum-area rectangle that contains U .

Theorem 2.10.11 *There exists a p -processor BSP algorithm for solving the minimum-area-enclosing-rectangle problem on n two-dimensional points that for all integers n and p such that $p = O(n)$, requires time $C_{c_hull}(n, p) + O(C_{ppf}(p)) + O(n/p)$ and $M_{c_hull}(n, p) + O(M_{ppf}(p))$.*

Chapter 3

Fundamental Data Structures

3.1 Priority Queues

In this section we design, analyze and implement BSP algorithms for a fundamental dynamic data structure, namely the *priority queue*. Priority queues [133] have always been a fruitful research topic both in sequential [54, 69, 133, 202, 211] and parallel computation [27, 37, 121, 124, 125, 127, 171, 179, 180, 181].¹ Parallel priority queues are frequently recurring data structures in computer science and are fundamental for many parallel applications, such as implementing branch-and-bound algorithms [27, 54, 78, 79, 133, 189]. The material of this section is based on the following papers:

- (1) “Concurrent heaps on the BSP model.” Technical Report PRG-TR-14-96, Computing Laboratory, Oxford University, June 1996 [78].
- (2) “Selection on the bulk-synchronous parallel model with applications to priority queues.” In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, August 1996 [79].
- (3) “Selection on the BSP model with applications to parallel priority queues.” Submitted for journal publication, 1997 [94].

3.1.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a non-decreasing sequence whose items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments. We consider, in this section, the problem of *inserting*, *deleting-min* and *finding-min* for the items in sequence X into *parallel priority queues* (PPQs) – in particular, (n, d) -bandwidth concurrent heaps. A detailed account of the definitions and notation pertaining inserting, deleting-min and finding-min on (n, d) -bandwidth concurrent heaps is diverged to sub-section 3.1.4.1.

¹Since the advent of parallel computing considerable work has been done towards solving the problem of maintaining dynamic priority queues subject to the operations of insert, delete-min and find-min.

Previous approaches based on traditional *concurrent* priority queue algorithms [37, 121, 180, 181] provide only limited parallelism. In this paper, we concern ourselves with *parallel* priority queue algorithms [124, 125, 127, 171, 179] that offer the prospect of a high degree of parallelism.

3.1.2 Cost Model

The performance of the BSP parallel priority queue algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations. Furthermore, for some constants c_{ins} , c_{del} and c_{sel} , time $c_{ins} \cdot \lg m$, $c_{del} \cdot \lg m$ and $c_{sel} \cdot (n + C)$ is charged for inserting and deleting-min (a single item) in/from a heap of size m , and for selecting the C -th smallest item from a sequence of size n .

Remark In the remainder of this work we refer to the aforementioned inserting and deleting-min algorithms as SEQ_INSERTQ and SEQ_DELETEMINQ respectively. Additionally, we refer to the finding-min algorithm as SEQ_FINDMINQ.

Optimal $\Theta(n \cdot \lg m)$ time sequential inserting, deleting-min and finding-min (on ordinary heaps of size m) algorithms are well-known [54, 133]. In the following sections, we design and analyze optimal BSP inserting, deleting-min and finding-min algorithms – optimality is realized for an ample range of the BSP parameters.

3.1.3 Related Work and New Results

There has been a notable amount of previous work – mainly on the PRAM model – on concurrent and parallel priority queues (e.g., [37, 121, 171, 180, 181] on the PRAM model, [27] on the BSP* model, [179] on the butterfly model). The exhibits in tables 3.1, 3.2 and 3.3 summarize the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality. To simplify the exposition, inserting, deleting-min and searching-min is assumed to transpire on ordinary binary heaps of size $n \leq m^{1-\epsilon} \leq 2^{n/\lg^2 n}$, for some constant $0 < \epsilon < 1$.

Refs	Computation	Communication	Constants	Implem.	Procs
[180]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■	■ ■	$O(\lg m)$
[121]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■	■ ■	$O(\lg m)$
[171]	$O\left(\frac{n}{p} \cdot (\lg \frac{m}{n} + \lg n)\right)$	$O\left(\frac{n}{p} \cdot (\lg \frac{m}{n} + \lg p)\right)$	■ ■	■ ■ ■	$O(n)$
[179]	$O\left(\frac{n}{p} \cdot \lg \frac{m}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■	$O\left(\frac{n}{\lg n}\right)$
[27]	$O\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	$O\left(\frac{n}{p}\right) + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg^4 n}\right)$
[27]	$O\left(\frac{n}{p} \cdot \lg \frac{m}{p}\right)$	$O\left(\frac{n}{p}\right)$	■	■	$O\left(\frac{n}{\lg^2 n}\right)$
Sec 3.1.5	$6 \cdot c_{sel} \cdot \frac{n}{p} \cdot \lg \frac{m}{n} + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	$O\left(\frac{n}{p}\right) + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	■	■ ■	$O(n)$
Sec 3.3.2	$c_{ins} \cdot \frac{n}{p} \cdot \lg \frac{m}{p} + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	$O\left(\frac{n}{p}\right)$	■	■	$O\left(\frac{n}{\lg n}\right)$

Table 3.1: Performance comparison of parallel priority queue algorithms (insert operation) – the cyan color is associated with randomized approaches.

With reference to tables 3.1, 3.2 and 3.3 we note that the algorithms of [121, 179, 180] and of section 3.3.2 can be employed to support parallel priority queue operations on data sets of varying size (as opposed to the algorithms of [27, 171] that can only be employed for data sets of predetermined size).

The optimal BSP inserting algorithms of sections 3.1.5 and 3.3.2 improve upon previous approaches on the following particulars:

Refs.	Computation	Communication	Constants	Implem.	Procs
[180]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■	■ ■	$O(\lg m)$
[121]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■	■ ■	$O(\lg m)$
[171]	$O\left(\frac{n}{p} \cdot (\lg \frac{m}{n} + \lg n)\right)$	$O\left(\frac{n}{p} \cdot (\lg \frac{m}{n} + \lg p)\right)$	■ ■	■ ■ ■	$O(n)$
[179]	$O\left(\frac{n}{p} \cdot (\lg \frac{m}{p} + \lg n)\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[27]	$O\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	$O\left(\frac{n}{p}\right) + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg^4 n}\right)$
[27]	$O\left(\frac{n}{p} \cdot \lg \frac{m}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg^2 n}\right)$
Sec 3.1.5	$6 \cdot c_{sel} \cdot \frac{n}{p} \cdot \lg \frac{m}{n} + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	$O\left(\frac{n}{p}\right) + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	■	■ ■	$O(n)$
Sec 3.3.2	$c_{del} \cdot \frac{n}{p} \cdot \lg \frac{m}{p} + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	$o\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$

Table 3.2: Performance comparison of parallel priority queue algorithms (delete-min operation) – the cyan color is associated with randomized approaches.

Refs	Computation	Communication	Constants	Implem.	Procs
[180]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■	■ ■	$O(\lg m)$
[121]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■	■ ■	$O(\lg m)$
[171]	$O(1)$	–	■	■	$O(n)$
[179]	$O\left(\frac{n}{p} \cdot (\lg \frac{m}{p} + \lg n)\right)$	$O\left(\frac{n}{p} \cdot \lg p\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg n}\right)$
[27]	$O(1)$	–	■	■	$O\left(\frac{n}{\lg^4 n}\right)$
[27]	$O\left(\frac{n}{p} \cdot \lg \frac{m}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg^2 n}\right)$
Sec 3.1.5	$O(1)$	–	■	■	$O(n)$
Sec 3.3.2	$c_{del} \cdot \frac{n}{p} \cdot \lg \frac{m}{p} + o\left(\frac{n}{p} \cdot \lg \frac{m}{n}\right)$	$o\left(\frac{n}{p}\right)$	■	■ ■	$O\left(\frac{n}{\lg n}\right)$

Table 3.3: Performance comparison of parallel priority queue algorithms (find-min operation) – the cyan color is associated with randomized approaches.

- (1) the asymptotic computation time requirements of [121, 180] (if $n, p = m^{1-o(1)}$),
- (2) the asymptotic communication time requirements of [121, 171, 180],
- (3) the multiplicative constant factor requirements of [27, 121, 171, 179, 180],
- (4) the implementation effort requirements of [171] (algorithm of section 3.1.5) and [27, 121, 171, 180] (algorithm of section 3.3.2), and
- (5) the parallel slackness requirements (to achieve optimality) of [27, 121, 179, 180] (algorithm of section 3.1.5) and [27, 121, 180] (algorithm of section 3.3.2).

Similarly, the optimal BSP deleting-min algorithms of sections 3.1.5 and 3.3.2 improve upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [121, 180] (if $n, p = m^{1-o(1)}$),
- (2) the asymptotic communication time requirements of [121, 171, 180] (algorithm of section 3.1.5) and [27, 121, 171, 179, 180] (algorithm of section 3.3.2),
- (3) the multiplicative constant factor requirements of [27, 121, 171, 179, 180],

- (4) the implementation effort requirements of [171], and
- (5) the parallel slackness requirements (to achieve optimality) of [27, 121, 179, 180] (algorithm of section 3.1.5) and [27, 121, 180] (algorithm of section 3.3.2).

Finally, the BSP find-min algorithms of sections 3.1.5 and 3.3.2 improve upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [27, 121, 179, 180] (algorithm of section 3.1.5) and [121, 180] (algorithm of section 3.3.2 – if $n, p = m^{1-o(1)}$),
- (2) the asymptotic communication time requirements of [27, 121, 179, 180],
- (3) the multiplicative constant factor requirements of [27, 121, 179, 180],
- (4) the implementation effort requirements of [27, 121, 179, 180] (algorithm of section 3.1.5), and
- (5) the parallel slackness requirements (to achieve optimality) of [27, 121, 179, 180] (algorithm of section 3.1.5) and [27, 121, 180] (algorithm of section 3.3.2).

The principal contribution, however, of the inserting, deleting-min and finding-min algorithms of sections 3.1.5 and 3.3.2 is attributed to the *low communication cost* (refer to case 2), the *small constant multiplicative factors* (refer to case 3), and the *low amount of parallel slackness* required to achieve optimality (refer to case 5). Additionally, our techniques provide *one-optimality* for an ample range of the BSP parameters.

3.1.4 (n, d) -Bandwidth Concurrent Heaps

In this section we study the computation and communication requirements of inserting, deleting-min and finding-min on a randomized processor-mapping of a variant of n -bandwidth heaps [171], namely (n, d) -bandwidth concurrent heaps – a combination of n -bandwidth heaps [171] and ordinary d -ary heaps [54, 133, 202]. The utilization of (n, d) -bandwidth concurrent heaps allows elegant high-level design and analysis of several parallel techniques – using data structures similar to those of sequential models – such as *branch-and-bound* [124, 125, 127], which require the solution of several sub-problems, each associated with a possibly different cost.

3.1.4.1 Preliminaries and Notation

In the remainder of this section, we formally define bandwidth heaps and discuss some of their properties. A bandwidth heap is an abstract data structure for maintaining a collection of (possibly duplicate) items and selecting the n items associated with the smallest values. We assume that the items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments.

The randomized algorithms we present in this work for supporting the pre-mentioned bandwidth heap data structure are based on a new randomized processor-mapping of a variant of the n -bandwidth heaps introduced in [171]. Formally, the new data structure, which we call (n, d) -bandwidth concurrent heap (in short, (n, d) -BCH), is defined as a concurrent d -ary heap whose nodes, each consisting of n items, are arranged in *extended heap order*. By extended heap order we imply that the value of an item at any node is not greater than the value of the items at each of its children. By employing (n, d) -BCHs (in contrast

to $(n, 2)$ -BCHs) we achieve a speedup for insertions at the expense of deletions. We can therefore adapt the parameter d so as to select a data structure that fits the relative frequencies of the heap operations.

Let m be the number of items stored in a (n, d) -BCH H , with $k = m/n$ being the number of nodes in H . The height h of H is $\lceil \log_d((d-1) \cdot k + 1) \rceil$. Such a heap H can be conveniently implemented in terms of the following data structure. Each processor p_i , $0 \leq i < p$, maintains a one-dimensional array $A_i[1, 2, \dots, k]$ of size k , where each position j of the array is associated with a list L_j of items, $1 \leq j \leq k$. Each such list L_j contains the items of node v_j that reside on processor p_i . We note that the order of the items within these lists is irrelevant.

Along the lines of [180], we associate with H data fields *full_level* and *last_elem*. Data field *full_level* is the index of the leftmost node in the deepest non-empty level h of H , and *last_elem* is the index of the last non-empty node of H . For an empty heap, *full_level* and *last_elem* are both set to zero. The path from the root to the first empty node (target) of H , denoted by σ_{ins} , is called the *insertion-path* of H . Then, as shown in [181], the insertion path σ_{ins} of H can be represented by a sequence of $h-1$ digits in radix d , and hence, the next node in the insertion path can be determined in $O(1)$ time. In particular, if we denote by D the displacement of target at the deepest level, $D = \text{last_elem} - \text{full_level} + 1$, then we can view D as a $(h-1)$ -digit radix- d number that prescribes the insertion path of H . This path can be traversed as follows. Let $(i_{h-1}i_{h-2} \dots i_1)_d$ be the radix- d representation of D . Then the digits representing D (from the most significant to the least significant) indicate which node (child) to visit when we move from the root downward, that is, zero indicates the leftmost child and $d-1$ indicates the rightmost child with all intermediate values interpreted accordingly.

3.1.5 Optimal Randomized Inserting, Deleting-Min and Finding-Min Algorithms on (n, d) -Bandwidth Concurrent Heaps

The following operations are defined on a (n, d) -BCH H .

- (1) INSERT_BCH($\{y_1, y_2, \dots, y_n\}, H$) for the insertion of items $\{y_1, y_2, \dots, y_n\} \in U$ into H .
- (2) DELETMIN_BCH(n, H) for the determination and deletion of the n smallest items of H .
- (3) FINDMIN_BCH(n, H) for the determination of the n smallest items of H .

The implicit assumption is that there exists an operation CONSTRUCT_BCH(H) for the construction of an empty heap H .

In the remainder of this section we present the algorithmic idea underlying the usage of (n, d) -BCHs. More precisely, we demonstrate how to employ (n, d) -BCHs in order to embody a dynamically changing set of items induced by processing a sequence of n queries containing the instructions of (i) inserting an element, (ii) deleting the minimum element, and (iii) finding the minimum element. The algorithm is based on a communication efficient *load-balancing* scheme and achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal randomized insert, delete-min and find-min algorithms (on (n, d) -bandwidth concurrent heaps) are based on randomized load-balancing techniques and are iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Processors access the bandwidth concurrent heap (which is randomly distributed amongst the p -participating processors) in a top-down fashion. In each iteration of the insert algorithm, processors proceed from the top of the heap

and, through successive selection operations, actualize the "down-flow" of the new items to their target position. Finally, the delete-min algorithm (and similarly the find-min algorithm) is realized along the lines of the insertion algorithm (the delete-min algorithm is transformed to a complex variant of the insertion algorithm by removing the root node of the bandwidth concurrent heap and inserting in its position an appropriate leaf node).

Our main result is summarized in the following theorem.

Theorem 3.1.1 *Let H be a (n, d) -BCH of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the bandwidth heap operations insert, delete-min and find-min, that for all integers n and p such that $p = o(n/\lg n)$, requires time [with probability $1 - O(d \cdot h \cdot n^{-\rho})$]*

$$\begin{array}{ll} (c_{sel} + o(1)) \cdot (3 \cdot h) \cdot \frac{n}{p} & \text{for insert,} \\ (c_{sel} + o(1)) \cdot (((5/2) \cdot d + 1) \cdot (h - 1)) \cdot \frac{n}{p} + O(d \cdot \lg d) & \text{for delete-min,} \\ O(1) & \text{for find-min,} \end{array}$$

and

$$\begin{array}{ll} L \cdot O(h \cdot \lg p) + g \cdot O(h \cdot o(n/p) + \frac{n}{p}) & \text{for insert,} \\ L \cdot O(h \cdot d \cdot \lg p) + g \cdot O(h \cdot d \cdot o(n/p)) & \text{for delete-min,} \end{array}$$

for computation and communication respectively.

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 3.1.1 follows easily as a consequence of these results.

Consider a (n, d) -BCH H of height h . In the remainder of this section we consider a new randomized approach towards the implementation of a communication efficient algorithm for supporting inserting, deleting-min and finding-min on H .

The general formulation of our randomized technique is as follows. The algorithm computes the n insert (delete-min) processes for all n items in a sequence of phases. The number of phases corresponds to the number of levels (height) of the underlying (n, d) -BCH. At each phase l , $1 \leq l \leq h$, the items advance in a pipelined fashion through the bandwidth heap along their insert (delete-min) paths from level $l - 1$ to level l .² The procedures to manage the insert and delete-min operations are summarized in the following proposition.

Proposition 3.1.1 *Let H be a (n, d) -BCH of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the bandwidth heap operations insert, delete-min and find-min, that for all integers n and p such that $p = O(n/\lg n)$, requires time [with probability $1 - O(d \cdot h \cdot n^{-\rho})$]*

²Previous approaches employing bandwidth heaps [171] are quite complex and are based on communication intensive merging and sorting algorithms. In contrast, our approach is based on the communication efficient selection algorithm of section 2.8. Consequently, the communication requirements of our methods are within the lower-order terms of the computation requirements. Additionally, in contrast to previous approaches our approach achieves optimality to within a small multiplicative constant factor for a wide range of the BSP parameters.

$$C_{bch}(n, d, h, p) = \begin{cases} h \cdot (C_{sel}(2 \cdot p \cdot r, n, p) + C_{ppf}(p) + O(1)) & \text{for insert,} \\ (h-1) \cdot \left(\sum_{i=0}^{d-2} C_{sel}((d-i) \cdot p \cdot r, n, p) + C_{sel}(2 \cdot p \cdot r, n, p) + O(d \cdot \lg d) \right) & \text{for delete-min,} \\ O(1) & \text{for find-min,} \end{cases}$$

and

$$M_{bch}(n, d, h, p) = \begin{cases} h \cdot (L \cdot O(1) + M_{sel}(2 \cdot p \cdot r, n, p) + M_{ppf}(p)) + g \cdot O(r) & \text{for insert,} \\ (h-1) \cdot \left(\sum_{i=0}^{d-2} C_{sel}((d-i) \cdot p \cdot r, n, p) + M_{sel}(2 \cdot p \cdot r, n, p) + M_{bcast}^{O(d)}(p) \right) & \text{for delete-min,} \end{cases}$$

for computation and communication respectively, where $r = \lceil (1 + O(\sqrt{(p \cdot \lg n)/n})) \cdot (n/p) \rceil$. The overall time complexity of the algorithm is denoted by $T_{bch}(n, d, h, p) = C_{bch}(n, d, h, p) + M_{bch}(n, d, h, p)$.

Proof We initially present the procedures to realize the insert, delete-min and find-min operations on a (n, d) -BCH; later, we provide a correctness proof and an analysis of the resources required. Some notation is required before proceeding to the details of the algorithm. Let v be a node in a (n, d) -BCH H . Along the lines of [171], by E_v we denote the set of n items stored in node v of H , and by $\max\{E_v\}$ the item of E_v with the maximum value. Moreover, let nodes $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$ be the children of v . By $\min_j \{u^{(j)}\}$ we denote the child of v that has the minimum $\max\{E_{u^{(j)}}\}$. Finally, the path from the root to the first empty node of H , denoted by σ_{ins} , is called the insertion-path of H .

The insertion process is realized by procedure INSERT_BCH($\{y_1, y_2, \dots, y_n\}, H$) (refer to figure 3.1), where $Y = \{y_1, y_2, \dots, y_n\}$ denotes the set of n items to be inserted into H .³ We note that in order to achieve communication efficiency we maintain the following randomized mapping of H on the processors. The items $\{y_1, y_2, \dots, y_n\}$ are randomly distributed among the p processors at the beginning of the insertion process (line 8). In later phases, whenever items are to be transferred from one node to another they are simply re-labeled locally (refer to lines 13, 15-16), and therefore, no inter-processor communication is required during these subsequent phases. This random distribution (line 8) is the only operation where physical movement of items occurs (except from input and output). Thus, in the remainder of this analysis whenever a data movement is implied, a *local re-labeling operation* actually takes place. This re-labeling technique guarantees the low-contention of our algorithm and preserves the random distribution of items to processors.

A graphical example of the insertion process on a $(3, 3)$ -BCH of 33 items is given in figure 3.2. We observe that the insertion process preserves the extended heap order of H . In particular, for each node $v^{(i)}$, $1 \leq i \leq |\sigma_{ins}|$, in the insertion path σ_{ins} of H , the value of $\max\{E_{v^{(i)}}\}$ is not incremented by procedure INSERT. Consequently, the items stored in the children of $v^{(i)}$ are not less than $\max\{E_{v^{(i)}}\}$ and the correctness of the procedure follows.

The resources required for a single iteration of procedure INSERT are bounded as follows. The determination of the next node in the insertion path of H (line 3), by way of the analysis presented in section 3.1.4.1, requires $O(1)$ sequential time. The local operations (lines 5, 7) also require $O(1)$ time. The random distribution of items to processors (line 8) takes at most $g \cdot O(r)$ communication time with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, where $r = \lceil (1 + \varepsilon) \cdot (n/p) \rceil$, and $\varepsilon = O(\sqrt{(p \cdot \lg n)/n})$. This is true by way of the inequality in theorem A.1.3 on the right tail of the binomial distribution. For the remainder of this proof, we therefore assume that each processor has at most $r = \lceil (1 + \varepsilon) \cdot (n/p) \rceil$ items

³With reference to the pseudo-code of this section, the superscript notation $\langle k \rangle$ refers to processor p_k (or equivalently, processor $p^{(k)}$) and implies the “local segment” of that processor.

```

begin INSERT_BCH ( $\{y_1, y_2, \dots, y_n\}, H$ )
1.   for  $i = 1$  to  $h$ 
2.       do denote by  $Y$  set  $\{y_1, y_2, \dots, y_n\}$  ;
3.       let node  $v^{(i)}$  at level  $i$ ,  $i \in \{1, 2, \dots, h\}$ , be the next node in the
           insertion path  $\sigma_{ins}$  of  $H$  ;
4.       if ( $i = 1$ ) then
5.           let  $last\_elem = last\_elem + 1$  ;
6.           if (level of  $full\_level$  is full) then
7.               update  $full\_level$ ;
8.               distribute randomly the items of  $Y$  among the  $p$  processors ;
9.       let  $e = \text{SELECT}(E_{v^{(i)}} \cup Y, n)$  ;
10.      for each processor  $p^{(k)}$ ,  $k \in \{0, 1, \dots, p-1\}$ , in parallel
11.          do if ( $i = h$ ) then
12.              let  $E_{v^{(i)}}^{(k)} = Y^{(k)}$  ;
13.          else
14.              let  $E_{v^{(i)}}^{(k)} = \{x \mid x \in E_{v^{(i)}}^{(k)} \cup Y^{(k)} \text{ and } x \leq e\}$  ;
15.              let  $Y^{(k)} = \{x \mid x \in E_{v^{(i)}}^{(k)} \cup Y^{(k)} \text{ and } x > e\}$  ;
16.  PARALLEL_PREFIX_MAX( $E_{v^{(i)}}$ ) ;
end INSERT_BCH

```

Figure 3.1: Procedure INSERT_BCH, p – number of processors, h – height of H .

from any node in H . The median selection process (line 9) takes at most $T_{sel}(2 \cdot p \cdot r, n, p)$ time. We note that we use the expression $2 \cdot p \cdot r$ to compensate for the uneven distribution of items to processors. Finally, the re-labeling operation is realized as a by-product of the selection process, and thus, the local operations (lines 12, 14-15) require $O(1)$ time. For implementation purposes, we require that the maximum item of the newly added node is determined and maintained (line 16). This step requires time $T_{ppf}(p)$. The time bound on the insertion process follows by observing that the pre-mentioned steps (except from the random distribution of line 8) are iterated $h = |\sigma_{ins}|$ times.

The process to manage the deletion operation on a (n, d) -BCH H is rather more complicated. We first define an auxiliary routine ADJUST on a set of sibling nodes that does not violate the extended heap property and facilitates the deletion process [171]. In particular, this procedure rearranges the items stored in the children of $v^{(i)}$ so that the items of one of the children can be shifted up without violating the extended heap order. The process is outlined in procedure ADJUST($n, \{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$) (refer to figure 3.3), where n is the size of a node and $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$ is a set of at most d sibling nodes.

Procedure ADJUST re-labels the items stored in nodes $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$, whilst maintaining the extended heap order, so that after its execution, node $u^{(\pi(1))}$ holds the n smallest items of the set, node $u^{(\pi(2))}$ holds the next n smallest items of the set, and so on, where initially $\langle u^{(\pi(1))}, u^{(\pi(2))}, \dots, u^{(\pi(d))} \rangle$ is the sequence of nodes in $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$ such that $\max\{E_{u^{(\pi(k))}}\} \leq \max\{E_{u^{(\pi(m))}}\}$, for all $k < m$, $1 \leq m \leq d$. The preservation of the extended heap order follows directly from the fact that $\max\{E_{u^{(j)}}\}$, $1 \leq j \leq d$, is not incremented by the process.

The time complexity of procedure ADJUST is derived as follows. The determination of sequence $\langle u^{(\pi(1))}, u^{(\pi(2))}, \dots, u^{(\pi(d))} \rangle$ (line 1) is realized by broadcasting items $\max\{E_{u^{(j)}}\}$, for all $1 \leq j \leq d$, to all processors in time $T_{bcst}^{O(d)}(p)$, and subsequently, by sorting locally these items in time $O(d \cdot \lg d)$ [133]. The

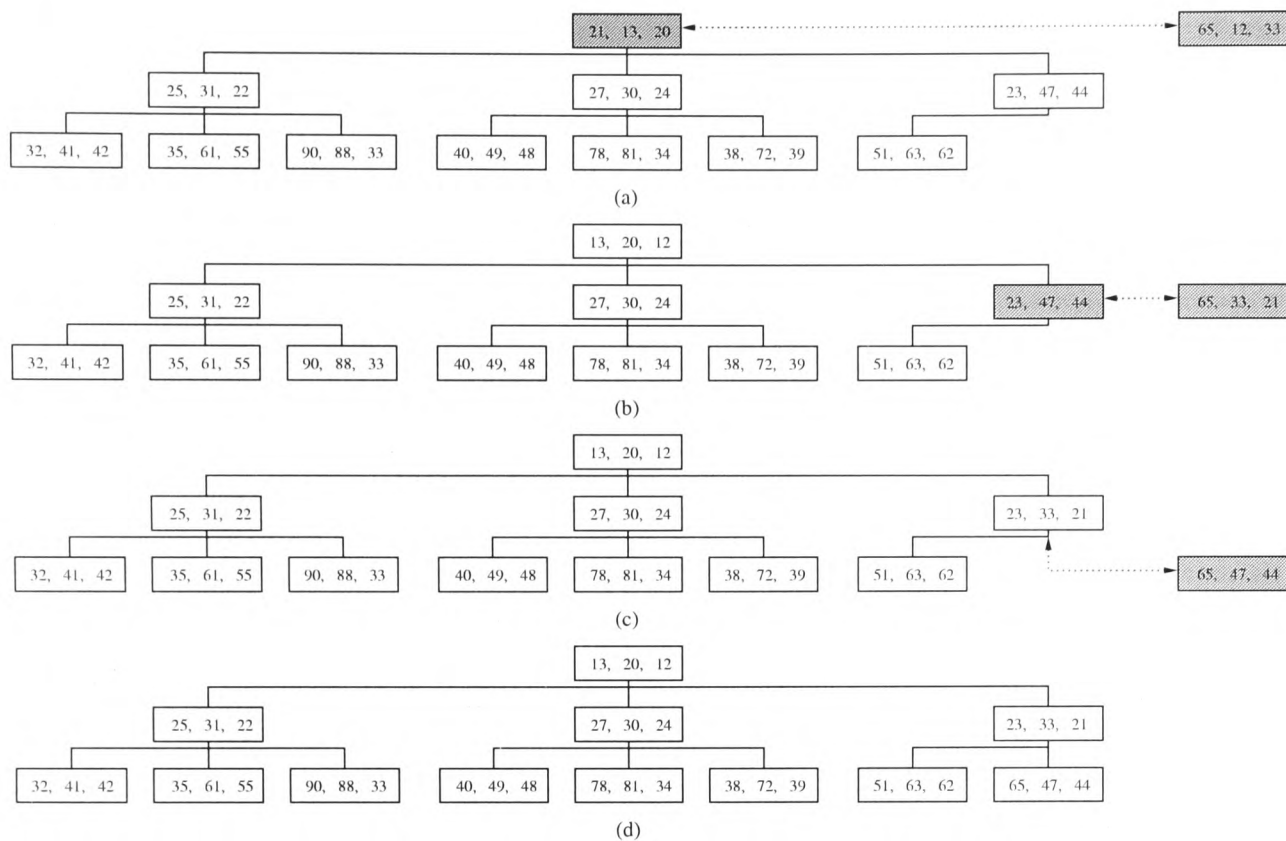


Figure 3.2: A (3,3)-BCH of 33 items. Insertion of item set {65, 12, 33}.

begin ADJUST ($n, \{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$)

1. **do let** $\langle u^{(\pi(1))}, u^{(\pi(2))}, \dots, u^{(\pi(d))} \rangle$ be the sequence of nodes in $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$,
such that $\max \{E_{u^{(\pi(k))}}\} \leq \max \{E_{u^{(\pi(m))}}\}, \forall k < m \in \{1, 2, \dots, d\}$;
2. **let** $e_j = \text{SELECT}(\bigcup_{l=1}^d E_{u^{(l)}}, j \cdot n), \forall j \in \{1, 2, \dots, d-1\}$;
3. **for** each processor $p^{(k)}, k \in \{0, 1, \dots, p-1\}$, in parallel
4. **do let** $E_{u^{(\pi(1))}}^{(k)} = \{x \mid x \in \bigcup_j E_{u^{(j)}}^{(k)} \text{ and } x \leq e_1\}$;
5. **let** $E_{u^{(\pi(j))}}^{(k)} = \{x \mid x \in \bigcup_j E_{u^{(j)}}^{(k)} \text{ and } e_{j-1} < x \leq e_j\}, \forall j \in \{2, \dots, d-2\}$;
6. **let** $E_{u^{(\pi(d))}}^{(k)} = \{x \mid x \in \bigcup_j E_{u^{(j)}}^{(k)} \text{ and } x > e_{d-1}\}$;

end ADJUST

Figure 3.3: Procedure ADJUST, p – number of processors.

selection process (line 2) takes time at most $\sum_{i=0}^{d-2} T_{sel}((d-i) \cdot p \cdot r, n, p)$. We note that for sufficiently large d it might be more efficient to sort the $d \cdot n$ items instead of performing $d-1$ selection processes. The re-labeling operation is realized as a by-product of the selection process, and therefore the local operations (lines 4-6) take $O(d)$ time.

The deletion process is realized by procedure DELETMIN_BCH(n, H) (figure 3.4), where n denotes the number of items to be deleted from H .

A graphical example of the deletion process on a (3,3)-BCH of 36 items is given in figure 3.5. We note that the deletion process maintains the extended heap order of H . In fact, for each node $v^{(i)}, 1 \leq i \leq h$ in the deletion path of H , the value of $\max \{E_{v^{(i)}}\}$ is not incremented by procedure DELETMIN_BCH by way of procedure ADJUST (line 10). Consequently, the items stored in the children of $v^{(i)}$ are not less than $\max \{E_{v^{(i)}}\}$ and the correctness of the procedure follows. We note that the preservation of the extended heap order is guaranteed by way of the auxiliary procedure ADJUST (line 10), that enables to shift up the items of one of the children of $v^{(i)}$ whilst maintaining the extended heap order.

```

begin DELETEMIN_BCH ( $n, H$ )
1.  for  $i = 1$  to  $h - 1$ 
2.      do if ( $i = 1$ ) then
3.          return  $E_{root(H)}$  ;
4.          let  $root(H) = v_{last\_elem}$  ;
5.          let node  $v^{(1)} = root(H)$  be the first node in the deletion path of  $H$  ;
6.          let  $last\_elem = last\_elem - 1$  ;
7.          if ( $(last\_elem < full\_level)$ ) then
8.              update  $full\_level$ ;
9.          denote by  $u^{(1)}, u^{(2)}, \dots, u^{(d)}$  the children of node  $v^{(i)}$  ;
10.         ADJUST ( $n, \{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$ ) ;
11.         let  $e = SELECT(E_{v^{(i)}} \cup E_{\min_j \{u^{(j)}\}}, n)$  ;
12.         for each processor  $p^{(k)}, k \in \{0, 1, \dots, p-1\}$ , in parallel
13.             do let  $E_{v^{(i)}}^{(k)} = \{x \mid x \in E_{v^{(i)}}^{(k)} \cup E_{\min_j \{u^{(j)}\}}^{(k)} \text{ and } x \leq e\}$  ;
14.             let  $E_{\min_j \{u^{(j)}\}}^{(k)} = \{x \mid x \in E_{v^{(i)}}^{(k)} \cup E_{\min_j \{u^{(j)}\}}^{(k)} \text{ and } x > e\}$  ;
15.         let node  $v^{(i)} = \min_j \{u^{(j)}\}$  be the next node in the deletion path of  $H$  ;
end DELETEMIN_BCH

```

Figure 3.4: Procedure DELETEMIN_BCH, p – number of processors, h – height of H , d – degree of H .

The time required for a single iteration of procedure DELETEMIN_BCH is determined as follows. The deletion of the n smallest items (line 3) and the local operations (lines 4-5, 6, 8, 15) are realized in $O(1)$ computation time. The time required for the ADJUST procedure (line 10) has been analyzed in the previous paragraphs. The selection process (line 11) takes at most $T_{sel}(2 \cdot p \cdot r, n, p)$ time. The re-labeling operations are realized as a by-product of the selection process, and therefore, the local operations (lines 13-14) take $O(1)$ time. Finally, the time bound on the deletion process follows by observing that the pre-mentioned steps are iterated $h - 1$ times, where h is the height of the underlying (n, d) -BCH H .

Finally, the find-min process is realized in $O(1)$ time by procedure FINDMIN_BCH(n, H) (refer to figure 3.6), where n denotes the number of minimal items in H to be determined.

Analysis The correctness of the algorithm follows from its description. By summing up the computation and communication requirements of procedures INSERT_BCH, ADJUST, DELETEMIN_BCH and FINDMIN_BCH we obtain the desired time bound. The probability bound follows by way of the analysis of the random distribution of procedure INSERT_BCH. We also observe that the aforementioned procedures perform a total of at most $d \cdot h$ selection operations – by way of proposition 2.8.3, the failure probability of each such selection operation is $O(n^{-\rho})$.

The proposition follows. $\square$

3.1.5.1 Improving Deleting-Min and Finding-Min

The performance of auxiliary procedure ADJUST, presented in section 3.1.5 to facilitate the deletion process, can be improved by utilizing a rather more complicated algorithmic scheme. Recall that procedure ADJUST (refer to figure 3.3) rearranges the items stored in the children of $v^{(i)}$ so that the items of one of the children can be shifted up without violating the extended heap order. As outlined in figure 3.3, the

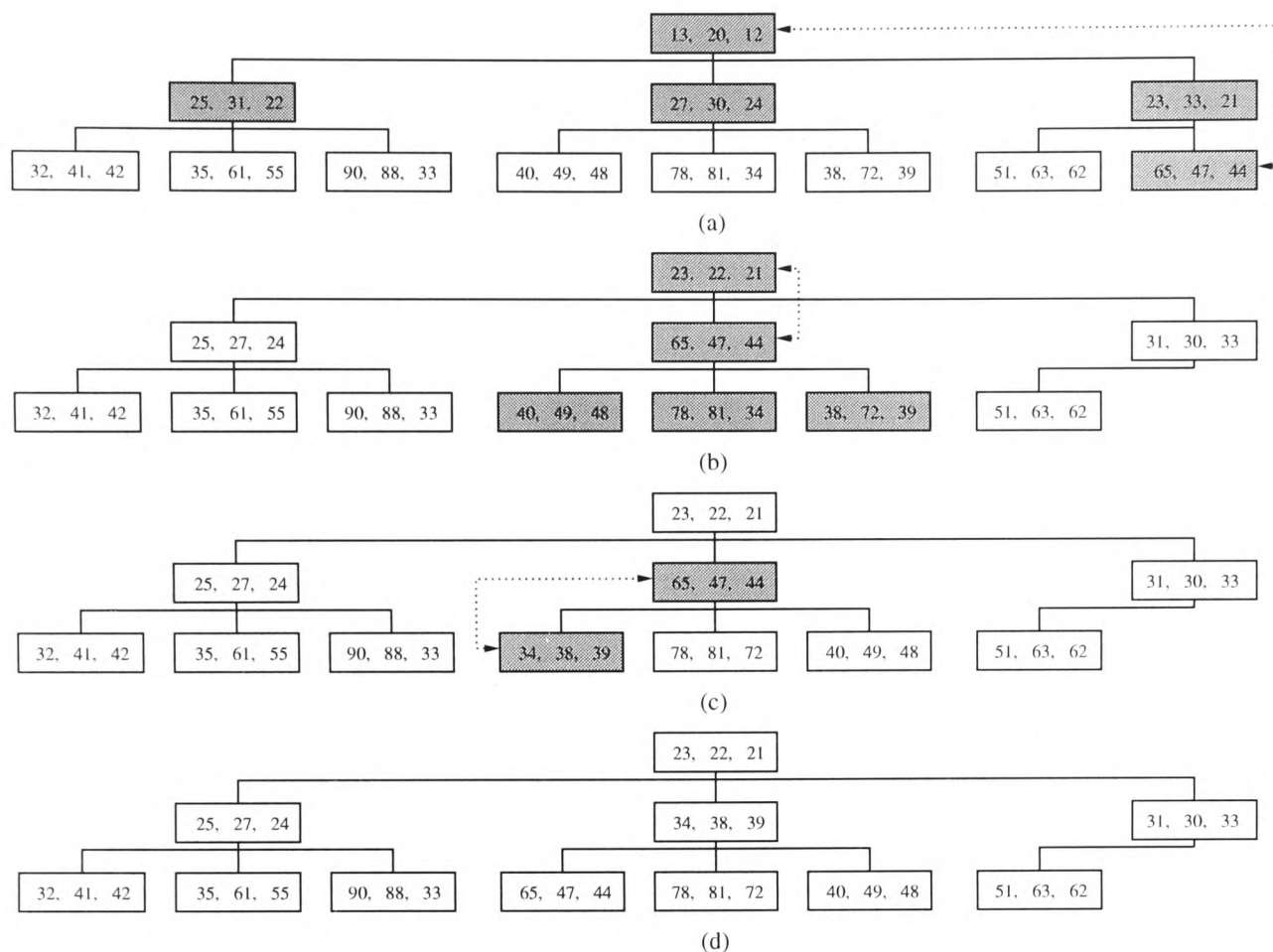


Figure 3.5: A (3,3)-BCH of 36 items. Deletion of the 3 smallest items.

begin FINDMIN_BCH (n, H)

1. **for** each processor $p^{(k)}$, $k \in \{0, 1, \dots, p-1\}$, in parallel
2. **do return** $E_{root(H)}^{(k)}$;

end FINDMIN_BCH

Figure 3.6: Procedure FINDMIN_BCH, p - number of processors.

procedure realizes the desired rearrangement by performing $d-1$ selections on a set of size $d \cdot n$. We next describe how procedure ADJUST can be realized by performing 1 and $(d-2)$ selections on sets of sizes $d \cdot n$ and n respectively. The process is outlined in procedure ADJUST_H($n, \{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$) (refer to figure 3.7), where n is the size of a node and $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$ is a set of at most d sibling nodes.

Procedure ADJUST_H re-labels the items stored in nodes $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$, whilst maintaining the extended heap order, so that after its execution, node $u^{(\pi(1))}$ holds the n smallest items of the set, where initially $\langle u^{(\pi(1))}, u^{(\pi(2))}, \dots, u^{(\pi(d))} \rangle$ is the sequence of nodes in $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$ such that $\max\{E_{u^{(\pi(k))}}\} \leq \max\{E_{u^{(\pi(m))}}\}$, for all $k < m$, $1 \leq m \leq d$. The preservation of the extended heap order follows directly from the fact that $\max\{E_{u^{(j)}}\}$, $1 \leq j \leq d$, is not incremented by the adjustment process.

The time complexity of procedure ADJUST_H is derived as follows. The determination of sequence $\langle u^{(\pi(1))}, u^{(\pi(2))}, \dots, u^{(\pi(d))} \rangle$ (line 1) is realized by broadcasting items $\max\{E_{u^{(j)}}\}$, for all $j \in \{1, 2, \dots, d\}$, to all processors in time $T_{broadcast}^{O(d)}(p)$, and subsequently, by sorting locally these items in time $O(d \cdot \lg d)$ [133]. The selection process (line 2) takes time at most $T_{sel}(d \cdot p \cdot r, n, p)$. The local operations (lines 4-5, 9) require time $O(d)$. The re-labeling operation is realized as a by-product of the selection process, and therefore the local operations (lines 6-8) take $O(1)$ time. The parallel-prefix and broadcast operations (lines 10-11) are realized in time at most $T_{ppf}^d(p)$ and $T_{broadcast}^{O(d)}(p)$ respectively. The selection process (line 12)

begin ADJUST_H ($n, \{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$)

1. **do let** $\langle u^{(\pi(1))}, u^{(\pi(2))}, \dots, u^{(\pi(d))} \rangle$ be the sequence of nodes in $\{u^{(1)}, u^{(2)}, \dots, u^{(d)}\}$,
such that $\max \{E_{u^{(\pi(k))}}\} \leq \max \{E_{u^{(\pi(m))}}\}, \forall k < m \in \{1, 2, \dots, d\}$;
2. **let** $e_1 = \text{SELECT}(\bigcup_{l=1}^d E_{u^{(l)}}, n)$;
3. **for each processor** $p^{(k)}, k \in \{0, 1, \dots, p-1\}$, in parallel
4. **do let** $r_j^{(k)} = |E_{u^{(\pi(j))}}^{(k)}|, \forall j \in \{2, \dots, d-1\}$;
5. **let** $E_{u^{(tmp)}}^{(k)} = E_{u^{(\pi(1))}}^{(k)}$;
6. **let** $E_{u^{(\pi(1))}}^{(k)} = \{x \mid x \in \bigcup_j E_{u^{(\pi(j))}}^{(k)} \text{ and } x \leq e_1\}$;
7. **let** $E_{u^{(tmp)}}^{(k)} = E_{u^{(tmp)}}^{(k)} - (E_{u^{(\pi(1))}}^{(k)} \cap E_{u^{(tmp)}}^{(k)})$;
8. **let** $E_{u^{(\pi(j))}}^{(k)} = E_{u^{(\pi(j))}}^{(k)} - (E_{u^{(\pi(1))}}^{(k)} \cap E_{u^{(\pi(j))}}^{(k)}), \forall j \in \{2, \dots, d\}$;
9. **let** $r_j^{(k)} = r_j^{(k)} - |E_{u^{(\pi(j))}}^{(k)}|, \forall j \in \{2, \dots, d-1\}$;
10. **PARALLEL_PREFIX_SUM** ($\{r_j^{(k)}\}, \forall j \in \{2, \dots, d-1\}$;
11. **BROADCAST** ($\{\sum_k r_j^{(k)}\}, \forall j \in \{2, \dots, d-1\}$;
12. **do let** $e_j = \text{SELECT}(E_{u^{(tmp)}}, \sum_{i=2}^j \sum_k r_i^{(k)}), j \in \{2, \dots, d-1\}$;
13. **for each processor** $p^{(k)}, k \in \{0, 1, \dots, p-1\}$, in parallel
14. **do let** $E_{u^{(\pi(2))}}^{(k)} = E_{u^{(\pi(2))}}^{(k)} \cup \{x \mid x \in E_{u^{(tmp)}}^{(k)} \text{ and } x \leq e_2\}$;
15. **let** $E_{u^{(\pi(j))}}^{(k)} = E_{u^{(\pi(j))}}^{(k)} \cup \{x \mid x \in E_{u^{(tmp)}}^{(k)} \text{ and } e_{j-1} < x \leq e_j\}, \forall j \in \{3, \dots, d-1\}$;
16. **let** $E_{u^{(\pi(d))}}^{(k)} = E_{u^{(\pi(d))}}^{(k)} \cup \{x \mid x \in E_{u^{(tmp)}}^{(k)} \text{ and } x > e_{d-1}\}$;

end ADJUST_H

Figure 3.7: Procedure ADJUST_H, p – number of processors.

takes time at most $(d-2) \cdot T_{sel}(p \cdot r, \lceil n/2 \rceil, p)$. We note that for sufficiently large d it might be more efficient to sort the at most n items instead of performing $d-1$ selection processes. Finally, the re-labeling operation is realized as a by-product of the selection process, and therefore the local operations (line 14-16) take $O(d)$ time. We therefore obtain the following analogue of proposition 3.1.1.

Proposition 3.1.2 *Let H be a (n, d) -BCH of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the bandwidth heap operations insert, delete-min and find-min, that for all integers n and p such that $p = O(n/\lg n)$, requires time [with probability $1 - O(d \cdot h \cdot n^{-\rho})$]*

$$C_{bch}(n, d, h, p) = \begin{cases} h \cdot (C_{sel}(2 \cdot p \cdot r, n, p) + C_{ppf}(p) + O(1)) & \text{for insert,} \\ (h-1) \cdot (C_{sel}(d \cdot p \cdot r, n, p) + (d-2) \cdot C_{sel}(p \cdot r, \lceil \frac{n}{2} \rceil, p) + \\ C_{sel}(2 \cdot p \cdot r, n, p) + C_{ppf}^d(p) + O(d \cdot \lg d)) & \text{for delete-min,} \\ O(1) & \text{for find-min,} \end{cases}$$

and

$$M_{bch}(n, d, h, p) = \begin{cases} h \cdot (L \cdot O(1) + M_{sel}(2 \cdot p \cdot r, n, p) + M_{ppf}(p)) + g \cdot O(r) & \text{for insert,} \\ (h-1) \cdot (C_{sel}(d \cdot p \cdot r, n, p) + (d-2) \cdot C_{sel}(p \cdot r, \lceil \frac{n}{2} \rceil, p) + \\ C_{sel}(2 \cdot p \cdot r, n, p) + C_{ppf}^d(p) + 2 \cdot M_{bcast}^{O(d)}(p)) & \text{for delete-min,} \end{cases}$$

for computation and communication respectively, where $r = \lceil (1 + O(\sqrt{(p \cdot \lg n)/n})) \cdot (n/p) \rceil$. The overall time complexity of the algorithm is denoted by $T_{bch}(n, d, h, p) = C_{bch}(n, d, h, p) + M_{bch}(n, d, h, p)$.

Proof (of theorem 3.1.1) The theorem follows easily from proposition 3.1.2 by substituting for T_{sel} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.8.3, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

3.1.5.2 Introducing Concurrency

Procedure INSERT_BCH and DELETMIN_BCH of section 3.1.5 (refer to proposition 3.1.2) can be conveniently pipelined, since both manipulate the underlying (n, d) -BCH heap from top to bottom. The pipelining of these operations can be exploited on a twofold basis: (i) procedures INSERT_BCH and DELETMIN_BCH heavily utilize parallel-prefix and broadcast operations, and therefore, the underlying algorithms are expected to benefit from pipelining these operations, and (ii) pipelining can be exploited in order to decrease the amount of parallel slack required to obtain optimality. In the remainder of this section we focus on the latter case; the former case can be treated by a similar line of arguments.

Informal Description of Algorithm The optimal randomized insert, delete-min and find-min algorithms (on (n, d) -bandwidth concurrent heaps) are immediate pipelined adaptations of the insert, delete-min and find-min algorithms of section 3.1.5.

Our main result is summarized in the following theorem.

Theorem 3.1.2 *Let H be a $(n, 2)$ -BCH of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the bandwidth heap operations insert, delete-min and find-min, that for all integers n and p such that $p = o(\min\{n, h \cdot (n/\lg n)\})$, requires amortized time [with probability $1 - O(h \cdot n^{-\rho})$]*

$$\begin{array}{ll} (c_{sel} + o(1)) \cdot (6 \cdot (h - 1)) \cdot \frac{n}{p} & \text{for insert/delete-min,} \\ O(1) & \text{for find-min,} \end{array}$$

and

$$L \cdot O(\lg p) + g \cdot O(h \cdot o(n/p) + \frac{n}{p}) \quad \text{for insert/delete-min}$$

for computation and communication respectively.

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a number of technical results. Then, theorem 3.1.2 follows easily as a consequence of these results.⁴

The pipelining process is straightforward when procedures INSERT_BCH and DELETMIN_BCH do not operate simultaneously. Then, we can trivially verify the following analogue of proposition 3.1.2.

Proposition 3.1.3 *Let H be a (n, d) -BCH of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the bandwidth heap operations insert, delete-min*

⁴For the special case that the insert and delete-min operations are not realized simultaneously the following (improved) analogue of theorem 3.1.2 applies.

Theorem 3.1.3 *Let H be a (n, d) -BCH of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the bandwidth heap operations insert, delete-min and find-min (not simultaneously), that for all integers n and p such that $p = o(\min\{n, h \cdot (n/\lg n)\})$, requires time [with probability $1 - O(d \cdot h \cdot n^{-\rho})$]*

and *find-min* (not simultaneously), that for all integers n, p and h such that $p = O(\min \{n, h \cdot n / \lg n\})$, requires amortized time [with probability $1 - O(d \cdot h \cdot n^{-\rho})$]

$$C_{bch}(n, d, h, p) = \begin{cases} C_{sel}^h(2 \cdot p \cdot r, n, p) + C_{ppf}^h(p) + O(h) & \text{for insert,} \\ 2 \cdot \left(C_{sel}^{(h-1)/2}(d \cdot p \cdot r, n, p) + (d-2) \cdot C_{sel}^{(h-1)/2}(p \cdot r, \lceil \frac{n}{2} \rceil, p) + \right. \\ \left. C_{sel}^{(h-1)/2}(2 \cdot p \cdot r, n, p) + C_{ppf}^{d \cdot (h-1)/2}(p) \right) + O(h \cdot d \cdot \lg d) & \text{for delete-min,} \\ O(1) & \text{for find-min,} \end{cases}$$

and

$$M_{bch}(n, d, h, p) = \begin{cases} L \cdot O(1) + M_{sel}^h(2 \cdot p \cdot r, n, p) + M_{ppf}^h(p) + g \cdot O(H_{dstr}) & \text{for insert,} \\ 2 \cdot \left(C_{sel}^{(h-1)/2}(d \cdot p \cdot r, n, p) + (d-2) \cdot C_{sel}^{(h-1)/2}(p \cdot r, \lceil \frac{n}{2} \rceil, p) + \right. \\ \left. C_{sel}^{(h-1)/2}(2 \cdot p \cdot r, n, p) + C_{ppf}^{d \cdot (h-1)/2}(p) + 2 \cdot M_{bcast}^{O(d \cdot (h-1)/2)}(p) \right) & \text{for delete-min,} \end{cases}$$

for computation and communication respectively, where $r = \lceil (1 + O(\sqrt{(p \cdot \lg n)/n})) \cdot (n/p) \rceil$ and

$$H_{dstr} = \begin{cases} O\left(\min \left\{ \lg n, \frac{n}{p} \cdot \frac{\lg n}{\lg \lg n} \right\}\right) & \text{if } \frac{n}{p} = O(\lg n), \\ (1 + o(1)) \cdot \frac{n}{p} & \text{otherwise.} \end{cases}$$

The overall time complexity of the algorithm is denoted by $T_{bch}(n, d, h, p) = C_{bch}(n, d, h, p) + M_{bch}(n, d, h, p)$.

Proof The proof follows a line of arguments similar to that of proposition 3.1.2 and is briefly sketched in the following discussion. Pipelining is possible since procedures INSERT_BCH and DELETMIN_BCH both manipulate the underlying (n, d) -BCH from top to bottom and they both require a small portion of the heap at a time, consisting of one level for the INSERT_BCH procedure and two levels for the DELETMIN_BCH procedure. By substituting therefore in the analysis of proposition 3.1.2 for the broadcast and parallel-prefix operations, their pipelined versions (refer to propositions 2.1.2 and 2.1.4), and by employing proposition 2.8.4 for selection we obtain the desired result. We note that $\min \{h, \lg n\}$ expresses the portion of the degree of concurrency that is exploited to decrease the amount of parallel slackness required to achieve optimality. Unfortunately, the initial distribution of items to processors (line 8) of procedure INSERT_BCH cannot be efficiently pipelined (this is due to the synchronous nature of the BSP model). In the remainder of this proof we focus on bounding the time required for this initial distribution. We distinguish two cases.

Case 1 If $\frac{n}{p} = O(\lg n)$, then the random distribution of items to processors results, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, in $O(\min \{\lg n, (n/p) \cdot (\lg n / \lg \lg n)\})$ items per processor. This

$$\begin{aligned} (c_{sel} + o(1)) \cdot (3 \cdot h) \cdot \frac{n}{p} & \quad \text{for insert,} \\ (c_{sel} + o(1)) \cdot ((5/2) \cdot d + 1) \cdot (h-1) \cdot \frac{n}{p} + O(d \cdot \lg d) & \quad \text{for delete-min,} \\ O(1) & \quad \text{for find-min,} \end{aligned}$$

and

$$\begin{aligned} L \cdot O(\lg p) + g \cdot O(h \cdot o(n/p) + \frac{n}{p}) & \quad \text{for insert,} \\ L \cdot O(d \cdot \lg p) + g \cdot O(h \cdot d \cdot o(n/p)) & \quad \text{for delete-min,} \end{aligned}$$

for computation and communication respectively.

claim can be derived by way of the inequality in theorem A.1.2 on the right tail of the binomial distribution.

Case 2 Otherwise, the random distribution of items to processors results, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, in $(1 + o(1)) \cdot (n/p)$ items per processor. This claim can be derived by way of the inequality in theorem A.1.3 on the right tail of the binomial distribution.

The proposition follows. $\square$

Proof (of theorem 3.1.3) The theorem follows easily from proposition 3.1.3 by substituting for T_{sel} , $T_{broadcast}$ and T_{ppf} the corresponding expressions in propositions 2.8.4, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

Although procedures INSERT_BCH and DELETEMIN_BCH both manipulate the underlying heap from top to bottom, the problem of realizing them concurrently (simultaneously) turns out to be rather complicated. Informally, the main problem associated with realizing insertions and deletions concurrently is summarized as follows. If procedure INSERT_BCH is still in progress when procedure DELETEMIN_BCH begins its execution, then the target node for procedure DELETEMIN_BCH is empty. We are then faced with two options to remedy the situation, neither of which is satisfactory: (i) procedure DELETEMIN_BCH selects some other leaf node, which may result in an unbalanced heap, and (ii) procedure DELETEMIN_BCH delays its operation until the most recent INSERT_BCH procedure finishes, which results in limited concurrency. To alleviate these limitations we adopt an implementation that is based on the locking scheme of [180], that achieves concurrency, while preserving the consistent and deadlock-free operation of the underlying algorithm. Instead of locking the entire heap, the algorithm locks only a small portion of the heap at a time, consisting of one node (corresponding to one level of the underlying heap) for the INSERT_BCH procedure and $d + 1$ nodes (corresponding to two levels of the underlying heap) for the DELETEMIN_BCH procedure. Distinct insert and delete-min operations access the contents of a node only after locking it to ensure mutual exclusion. The implementation details follow. We associate a field called *status* with every node in the heap. This field may obtain two distinct values, each associated with the following semantics [180]: *Pending* – an insertion is in progress which ultimately inserts into the node, and *Wanted* – a delete-min operation is waiting for the node. When procedure INSERT_BCH starts, the status of its target is set to *Pending*. Accordingly, when procedure DELETEMIN_BCH starts (and an insert operation is still in progress), the status of the target of the last insert operation is set to *Wanted*, and the DELETEMIN_BCH procedure is delayed for one phase. Moreover, at each iteration of procedure INSERT_BCH, the status of the target node is checked, and if it has been set to *Wanted*, it is placed at the root and the insertion process for that node is abandoned. Once the node is placed at the root, the DELETEMIN_BCH procedure resumes its operation. Finally, data fields *full_level* and *last_elem*, are modified only in the initialization phase of procedures INSERT and DELETEMIN_BCH.

This locking scheme can be implemented through a broadcast operation and a constant number of local operations. This additional cost does not affect (asymptotically) the time bounds of section 3.1.5. We thus obtain a result similar to that of proposition 3.1.3 for the case that procedures INSERT_BCH and DELETEMIN_BCH operate concurrently.

Remark It should be noted, however, that the cost of procedures INSERT and DELETEMIN_BCH can no longer be de-coupled (due to the concurrent nature of the implementation), and therefore, we should consider, in each iteration, the combined cost of the two procedures.

Therefore, we can easily verify the following analogue of proposition 3.1.3.

Proposition 3.1.4 *Let H be a (n, d) -BCH of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the bandwidth heap operations insert, delete-min and find-min, that for all integers n, p and h such that $p = O(\min\{n, h \cdot n/\lg n\})$, requires amortized time [with probability $1 - O(d \cdot h \cdot n^{-\rho})$]*

$$C_{bch}(n, d, h, p) = \begin{cases} \max \left\{ \begin{aligned} &C_{sel}^h(2 \cdot p \cdot r, n, p) + C_{ppf}^h(p) + O(h), \\ &2 \cdot \left(C_{sel}^{(h-1)/2}(d \cdot p \cdot r, n, p) + (d-2) \cdot C_{sel}^{(h-1)/2}(p \cdot r, \lceil \frac{n}{2} \rceil, p) + \right. \\ &\left. C_{sel}^{(h-1)/2}(2 \cdot p \cdot r, n, p) + C_{ppf}^{d \cdot (h-1)/2}(p) \right) + O(h \cdot d \cdot \lg d) \end{aligned} \right\} & \text{for insert/delete-min,} \\ O(1) & \text{for find-min,} \end{cases}$$

and

$$M_{bch}(n, d, h, p) = \begin{cases} \max \left\{ \begin{aligned} &L \cdot O(1) + M_{sel}^h(2 \cdot p \cdot r, n, p) + M_{ppf}^h(p) + g \cdot O(H_{dstr}), \\ &2 \cdot \left(C_{sel}^{(h-1)/2}(d \cdot p \cdot r, n, p) + (d-2) \cdot C_{sel}^{(h-1)/2}(p \cdot r, \lceil \frac{n}{2} \rceil, p) + \right. \\ &\left. C_{sel}^{(h-1)/2}(2 \cdot p \cdot r, n, p) + C_{ppf}^{d \cdot (h-1)/2}(p) + 2 \cdot M_{bcast}^{O(d \cdot (h-1)/2)}(p) \right) \end{aligned} \right\} & \text{for insert/delete-min,} \end{cases}$$

for computation and communication respectively, where $r = \lceil (1 + O(\sqrt{(p \cdot \lg n)/n})) \cdot (n/p) \rceil$ and

$$H_{dstr} = \begin{cases} O\left(\min\left\{\lg n, \frac{n}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{n}{p} = O(\lg n), \\ (1 + o(1)) \cdot \frac{n}{p} & \text{otherwise.} \end{cases}$$

The overall time complexity of the algorithm is denoted by $T_{bch}(n, d, h, p) = C_{bch}(n, d, h, p) + M_{bch}(n, d, h, p)$.

The main inefficiency of the method implied in the aforementioned proposition derives from that fact that procedures INSERT_BCH and DELETMIN_BCH can no longer be de-coupled (due to the concurrent nature of the implementation), and therefore, the advantages of (n, d) -BCHs (over $(n, 2)$ -BCHs) are no longer valid – this fact is reflected in theorem 3.1.2 where the notion of d -ary bandwidth heaps is limited to that of binary ones.

Proof (of theorem 3.1.2) The theorem follows easily from proposition 3.1.4 by substituting for T_{sel} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.8.4, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

3.2 Dictionaries

In this section we design, analyze and implement BSP algorithms for a fundamental dynamic data structure, namely the *dictionary*. Dictionaries [133] have always been a fruitful research topic both in sequential [2, 28, 29, 54, 62, 65, 74, 133, 202] and parallel computation [25, 30, 36, 57, 63, 64, 67, 120, 138, 139, 143, 170, 178, 185, 188, 193].⁵ Parallel dictionaries are frequently recurring data structures in computer science and are fundamental for many parallel applications, such as implementing database primitives

⁵Since the advent of parallel computing considerable work has been done towards solving the problem of maintaining static and dynamic dictionaries subject to the operations of insert, delete and find.

Previous approaches based on traditional *concurrent* dynamic dictionary algorithms [30, 36, 67, 120, 143, 138, 139, 143, 188, 193] provide only limited parallelism. In this paper, we concern ourselves with *parallel* dictionary algorithms [25, 57, 63, 64, 170, 178, 185] that offer the prospect of a high degree of parallelism.

and improving the performance of combinatorial algorithms [25, 26, 54, 76, 80, 133]. The material of this section is based on the following papers:

- (1) “Communication efficient data structures on the BSP model with applications.” Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, May 1996 [76].
- (2) “Communication efficient data structures on the BSP model with applications in computational geometry.” In *Proceedings of Euro-Par’96 Parallel Processing, Lecture Notes in Computer Science* 1124, Springer-Verlag, August 1996 [80].
- (3) “Ordered h -level graphs on the BSP model.” Submitted for journal publication, 1997 [93].
- (4) “Static and dynamic hashing strategies on the BSP model.” Technical Report PRG-TR-25-97, Computing Laboratory, Oxford University, July 1997 [199].

3.2.1 Preliminaries and Notation

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ be a non-decreasing sequence whose items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments. Additionally, for hash-based operations we assume that the items belong to a finite universe $U = \{0, 1, \dots, q-1\}$, and that all the items appearing in sequence X are distinct. We consider, in this section, the problem of *inserting*, *deleting* and *finding*⁶ the items in sequence X into/from *parallel dictionaries* (PDs) – in particular, *ordered h -level graphs* and *hash tables*. A detailed account of the definitions and notation pertaining inserting, deleting and finding on ordered h -level graphs and hash tables is diverged to sub-sections 3.2.4.1 and 3.2.6.1 respectively.

3.2.2 Cost Model

The performance of the BSP dictionary algorithms of this section is specified as follows. Constant $O(1)$ time is charged for each (i) basic computational operation, and each (ii) operation over an associative operator in parallel-prefix computations. Furthermore, for some constants c_{ins} , c_{del} and c_{srch} , time $c_{ins} \cdot \lg m$, $c_{del} \cdot \lg m$, $c_{srch} \cdot \lg m$, and $O(n)$ is charged respectively for inserting, deleting and finding (a single item) in/from a balanced search tree of size m , and for integer-sorting a sequence (in the range $[0, 1, \dots, n^{O(1)}]$) of size n – within the context of this section we denote c_{srch} by c_* .

Remark In the remainder of this work we refer to the aforementioned liner-time integer-sorting algorithm as SEQ_RADIX_SORT. Additionally, we refer to the parallel version of algorithm SEQ_RADIX_SORT as RADIX_SORT (refer to section 2.4.4 for a detailed discussion of the parallel integer-sorting algorithm). We also refer to the aforementioned inserting, deleting and finding algorithms as SEQ_INSERTD, SEQ_DELETED and SEQ_FINDD respectively.

Optimal $\Theta(n \cdot \lg m)$ and $\Theta(n)$ time sequential inserting, deleting and finding (on balanced d -ary trees and hash tables of size m respectively) algorithms are well-known [54, 133]. In the following sections, we design and analyze optimal BSP inserting, deleting and finding algorithms – optimality is realized for an ample range of the BSP parameters.

⁶In the remainder of this section we employ the terms “search” and “find” interchangeably.

3.2.3 Related Work and New Results

There has been a notable amount of previous work – mainly on the PRAM model – on concurrent and parallel dictionaries (e.g., [67, 139, 143, 170] on the PRAM model, [24, 25] on the BSP* model, [63] on the distributed memory model, [57] on the hyper-cube model, [178, 185] on the butterfly model). The exhibits in tables 3.4, 3.5 and 3.6 summarize the aforementioned results with respect to the following statistics: computation time, communication time, multiplicative constant factors, implementation effort and amount of parallel slackness (in terms of maximum number of processors) required to achieve optimality. To simplify the exposition, inserting, deleting and finding is assumed to transpire on balanced d -ary trees of appropriate degree and of size $m \geq n$; with the exception of hash-based approaches that operate on hash tables of size $m \geq n$. Hash-based approaches are identified in tables 3.4, 3.5 and 3.6 with the $\dagger$ symbol.⁷

Refs	Computation	Communication	Constants	Implem.	Procs
[67]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[143]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[139]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[170]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■ ■	■ ■	$O(n)$
[63] $\dagger$	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■ ■	$O(n^\epsilon)$
[178]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg^2 n \cdot \lg \lg n}\right)$
[26]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■ ■ ■	$O\left(\frac{n}{\lg^3 n}\right)$
Sec 3.2.5	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■ ■ ■	$O\left(\frac{n}{\lg^{1+\alpha} n}\right)$
Sec 3.2.7 $\dagger$	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg n \cdot \lg \lg n}\right)$

Table 3.4: Performance comparison of parallel dictionary algorithms (insert operation) (for any constant $0 < \epsilon < 1$ and any $\alpha = \Omega(1)$) – the cyan color is associated with randomized approaches.

With reference to table 3.6 we note that the algorithms of [57] and of section 2.9.6 can be employed to realize searching (finding) for a much wider class of data structures than balanced d -ary trees (as opposed to the algorithms of [25, 26, 67, 139, 143, 170, 178, 185] that can only be employed for balanced d -ary trees). Additionally, we note that the parallel dictionary algorithms of section 3.2.7 operate on data sets consisting of unique items (no duplicates), and therefore, compare unfairly with the remaining algorithms; however, the algorithms can be trivially modified to operate on data sets consisting of duplicate items – at the expense of increasing the parallel slackness requirements (to achieve optimality) from $\lg n \cdot \lg \lg n$ to n^ϵ .

The optimal BSP inserting, deleting and finding algorithms of sections 2.9.6, 3.2.5 and 3.2.7 improve upon previous approaches on the following particulars:

- (1) the asymptotic computation time requirements of [25, 26, 57, 67, 139, 143, 170, 178, 185] (algorithms of section 3.2.7),
- (2) the asymptotic communication time requirements of [57, 67, 139, 143, 170, 178, 185] (algorithms of section 3.2.5 and 2.9.6) and of [25, 26, 57, 67, 139, 143, 170, 178, 185] (algorithms of

⁷In spite of the immense research interest in parallel hashing on concurrent read/write models, the research on exclusive read/write models has been rather sparse, and consequently, the references in the aforementioned tables (on parallel hashing) are likewise rather limited.

Refs	Computation	Communication	Constants	Implem.	Procs
[67]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[143]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[139]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[170]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■ ■	■ ■	$O(n)$
[63] †	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■ ■	$O(n^\epsilon)$
[178]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg^2 n \cdot \lg \lg n}\right)$
[26]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■ ■ ■	$O\left(\frac{n}{\lg^3 n}\right)$
Sec 3.2.5	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■ ■ ■	$O\left(\frac{n}{\lg^{1+\alpha} n}\right)$
Sec 3.2.7 †	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg n \cdot \lg \lg n}\right)$

Table 3.5: Performance comparison of parallel dictionary algorithms (delete operation) (for any constant $0 < \epsilon < 1$ and any $\alpha = \Omega(1)$) – the cyan color is associated with randomized approaches.

section 3.2.7),

- (3) the multiplicative constant factor requirements of [57, 63, 67, 139, 143, 170, 178, 185] (algorithms of section 3.2.5 and 2.9.6),
- (4) the implementation effort requirements of [25, 26, 63, 185], and
- (5) the parallel slackness requirements (to achieve optimality) of [25, 26, 63, 67, 139, 143, 178].

The principal contribution, however, of the inserting, deleting and finding algorithms of sections 3.2.5 and 2.9.6 is attributed to the *low communication cost* (refer to case 2), the *small constant multiplicative factors* (refer to case 3), and the *low amount of parallel slackness* required to achieve optimality (refer to case 5). Additionally, our searching techniques provide support for a much *wider class of data structures* than previous approaches [25, 26, 67, 139, 143, 170, 178, 185] and *one-optimality* for an ample range of the BSP parameters. Accordingly, the main contribution of the inserting, deleting and finding algorithms of section 3.2.7 is attributed to the *low computation cost* (refer to case 1), the *low communication cost* (refer to case 2), and the *low amount of parallel slackness* required to achieve optimality (refer to case 5).

3.2.4 (d, α) - $B_{2-3-4}+$ -Trees

In this section we study the computation and communication requirements of inserting, deleting and finding on a randomized processor-mapping of a special class of ordered h -level graphs [57], namely (d, α) - $B_{2-3-4}+$ -trees – a combination of $B+$ -trees [51, 54, 133] and $2-3-4+$ -trees [54].⁸ The utilization of (d, α) - $B_{2-3-4}+$ -trees allows elegant high-level design and analysis of several parallel techniques – using data structures similar to those of sequential models.

⁸We restrict ourselves to (d, α) - $B_{2-3-4}+$ -trees (as opposed to the broader class of ordered h -level graphs), because the full generality of ordered h -level graphs complicates – beyond the scope of this work – the design, analysis and implementation of dynamic algorithms.

Refs	Computation	Communication	Constants	Implem.	Procs
[67]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[143]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[139]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O(\lg m)$
[170]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■ ■	■ ■	$O(n)$
[57]	$O\left(\frac{n}{p} \cdot (\lg m \cdot \lg n)\right)$	$O\left(\frac{n}{p} \cdot (\lg m \cdot \lg n)\right)$	■ ■	■ ■	$O(n)$
[63] †	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■ ■	$O(n^\epsilon)$
[178]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■	$O\left(\frac{n}{\lg^2 n \cdot \lg \lg n}\right)$
[185]	$O\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \lg m\right)$	■ ■	■ ■ ■	$O(n)$
[25]	$c_* \cdot \frac{n}{p} \cdot \lg m + o\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■ ■ ■	$O\left(\frac{n}{\lg^3 n}\right)$
Sec 2.9.6	$c_* \cdot \frac{n}{p} \cdot \lg m + o\left(\frac{n}{p} \cdot \lg m\right)$	$O\left(\frac{n}{p} \cdot \frac{\lg m}{\lg(n/p)}\right)$	■	■ ■	$O\left(\frac{n}{\lg^{1+\alpha} n}\right)$
Sec 3.2.7 †	$O\left(\frac{n}{p}\right)$	$O\left(\frac{n}{p}\right)$	■ ■ ■	■ ■	$O\left(\frac{n}{\lg n \cdot \lg \lg n}\right)$

Table 3.6: Performance comparison of parallel dictionary algorithms (find operation) (for any constant $0 < \epsilon < 1$ and any $\alpha = \Omega(1)$) – the cyan color is associated with randomized approaches.

3.2.4.1 Preliminaries and Notation

In the remainder of this section, we formally define search trees and discuss some of their properties. The definitions and notation of this section derive from [30, 51, 54]. A search tree is an abstract data structure for maintaining a collection of (possibly duplicate) items and supporting efficient insert, delete, and find operations on data sets of size n . We assume that the items belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments.

The randomized algorithms we present in this work for supporting the pre-mentioned search tree data structure are based on a new randomized processor-mapping of a variant of $B+$ -trees [51]. Formally, the new data structure, which we call (d, α) - $B_{2-3-4}+$ -tree, is defined as follows. Let $d \geq 1$ and $\alpha \geq 2$ be non-negative integers. A directed tree T is in the class (d, α) - $B_{2-3-4}+$ -trees if T is either empty or has the following properties.

- (1) Each path from the root to any leaf of T has the same length (*height*) h .
- (2) All the keys reside in the leaves of T and internal nodes contain copies of the keys stored in the leaves.
- (3) Every node v of T has the following associated information: (i) $s[v]$ – the size of node v , and (ii) the $s[v]$ keys $k_1[v], k_2[v], \dots, k_{s[v]}[v]$ stored as a $2-3-4+$ -tree.⁹
- (4) If v is not a leaf node, then v accommodates $s[v] + 1$ additional pointers $e_0[v], e_1[v], \dots, e_{s[v]}[v]$ to the children of v .
- (5) If $K[e_i[v]]$ is the set of keys stored in the sub-tree $T[e_i]$ rooted at e_i , then the following

⁹A $2-3-4+$ -tree is a $B+$ -tree of minimum out-degree 2 and maximum out-degree 4 [54].

conditions hold:

$$\begin{aligned} \text{if } k \in K[e_0[v]] & \quad \text{then } k \leq k_0, \\ \text{if } k \in K[e_i[v]] & \quad \text{then } k_{i-1} < k \leq k_i, \quad 1 \leq i < s[v], \\ \text{if } k \in K[e_{s[v]}[v]] & \quad \text{then } k_{s[v]} < k. \end{aligned}$$

(6) Each node except the root and the leaves has at least $d + 1$ children. The root is a leaf or has at least α children.

(7) Each node has at most $\alpha \cdot d + 1$ children.

The following lemma captures many important facts about (d, α) - $B_{2-3-4}+$ -trees.

Lemma 3.2.1 *Let T be a (d, α) - $B_{2-3-4}+$ -tree consisting of m items. Then, the following statements hold.*

(1)

$$h = \begin{cases} 0 & \text{if } m = 0 \\ 1 & \text{if } 0 < m < \alpha \cdot d \\ \lfloor \log_{d+1} \frac{m}{\alpha \cdot d} \rfloor + 2 & \text{if } m \geq \alpha \cdot d. \end{cases}$$

$$(2) \quad 1 + (\alpha/d) \cdot ((d+1)^{h-1} - 1) \leq |T| \leq (1/(\alpha \cdot d)) \cdot ((\alpha \cdot d + 1)^h - 1) \text{ if } h \geq 1,$$

where $|T|$ denotes the number of nodes in T .

Proof The case $h = 0$ is trivial, since T is empty, and hence $|T| = 0$. We therefore assume that $h \geq 1$. Then,

$$|T| \geq 1 + \alpha \left((d+1)^0 + (d+1)^1 + \dots + (d+1)^{h-2} \right) = 1 + \frac{\alpha}{d} \left((d+1)^{h-1} - 1 \right).$$

Similarly, we obtain that

$$|T| \leq \sum_{i=0}^{h-1} (\alpha \cdot d + 1)^i = \frac{1}{\alpha \cdot d} \left((\alpha \cdot d + 1)^h - 1 \right).$$

We complete the proof of the lemma by observing that at height $h \geq 2$ there are at most $\alpha \cdot (t+1)^{h-2}$ nodes, or equivalently, at most $\alpha \cdot (t+1)^{h-2} \cdot t$ items. The lemma follows easily. $\square$

3.2.5 Optimal Randomized Inserting, Deleting and Finding Algorithms on (d, α) - $B_{2-3-4}+$ -Trees

The following operations are defined on a (d, α) - $B_{2-3-4}+$ -tree T .

- (1) INSERT- $T(\{y_1, y_2, \dots, y_n\}, T)$ for the insertion of $\{y_1, y_2, \dots, y_n\}$ into T .
- (2) DELETE- $T(\{y_1, y_2, \dots, y_n\}, T)$ for the deletion of $\{y_1, y_2, \dots, y_n\}$ from T .
- (3) FIND- $T(\{y_1, y_2, \dots, y_n\}, T)$ for the determination of $\{e_1, e_2, \dots, e_n\}$, where $e_i = y_i$ if $y_i \in T$, and $e_i = \{\}$ otherwise, for all $1 \leq i < n$.

The implicit assumption is that there exists a $\text{CONSTRUCT_T}(T)$ operation for the construction of an empty tree T .

In the remainder of this section we briefly sketch the algorithmic idea underlying the usage of (d, α) - B_{2-3-4} -trees. More precisely, we demonstrate how to employ (d, α) - B_{2-3-4} -trees in order to embody a dynamically changing set of items induced by processing a sequence of n queries containing the instructions of (i) inserting an element, (ii) deleting an element, and (iii) finding an element. The algorithm is based on a communication efficient *load-balancing* scheme and achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal randomized insert, delete and search algorithms (on (d, α) - B_{2-3-4} -trees) are immediate adaptations of the search algorithm of section 2.9.6.2 – with the exception that inserting and deleting require, in addition to some extra synthetic processing, a second sweep (in the inverse direction) of the underlying graph.

Our main result is summarized in the following theorem.

Theorem 3.2.1 *Let T be a (d, α) - B_{2-3-4} -tree of height h . Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting operations insert, delete and find on n queries, that for all integers n and p such that $p = O(n/((\alpha \cdot d)^3 \cdot \lg n))$ and $\alpha \cdot d = o(n/p)$, requires time [with probability $1 - O(h \cdot n^{-\rho \cdot n/(p \cdot (\alpha \cdot d)^3 \cdot \lg n)})$] $h \cdot (n/p) \cdot O(\lg(\alpha \cdot d))$ and $L \cdot h \cdot O(\lg p) + g \cdot h \cdot O(n/p)$, for computation and communication respectively.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to an informal discussion of the underlying algorithmic scheme. Then, theorem 3.2.1 follows as a consequence of this algorithmic scheme.

The general formulation of the algorithm is along the lines of section 2.9.6. The n processes for all n queries are realized in a sequence of phases. The number of phases corresponds to the number of levels (height) of the underlying (d, α) - B_{2-3-4} -tree. At each phase l , $1 \leq l \leq h$, the queries advance through the tree along their paths from level $l - 1$ to level l . We observe that the first phase is different from subsequent phases, since it has to manage an arbitrary permutation of the queries, whereas each subsequent phase may utilize the processing performed in the previous stage to improve the performance of the algorithm. The search algorithm (downward sweep of the tree) terminates within h phases and is realized by way of the randomized algorithm of sub-section 2.9.6.2.¹⁰

For insert and delete operations a maximum of at most h additional phases are required (upward sweep of the tree) [30, 51, 54]. For insert operations the upward sweep of the (d, α) - B_{2-3-4} -tree implies the splitting of over-full nodes – i.e., nodes of size greater than $\alpha \cdot d + 1$. For delete operations the upward sweep of the (d, α) - B_{2-3-4} -tree implies the joining of under-full nodes – i.e., nodes of size less than $d + 1$. The algorithms for the insert and delete operations are straightforward generalizations of the searching algorithm of section 2.9.6.2; however, the technical details involved in the underlying implementations are rather complicated and are omitted in this work.

Proof (of theorem 3.2.1) The theorem follows easily from theorem 2.9.3, by substituting for $d_{max} = \alpha \cdot d + 1$, $r = (\alpha - 1) \cdot d$ and $\tau_l = O(\lg(\alpha \cdot d))$. $\square$

¹⁰The motivation behind employing the randomized algorithm of section 2.9.6.2 is that it delivers superior performance for the case that $m > n$, where m denotes the number of nodes in the underlying (d, α) - B_{2-3-4} -tree.

3.2.6 Hash Tables

In this section we study the computation and communication requirements of inserting, deleting and finding on a randomized processor-mapping of hash tables. The utilization of hash tables allows elegant high-level design and analysis of several parallel techniques – using data structures similar to those of sequential models.

3.2.6.1 Preliminaries and Notation

In the remainder of this section, we formally define hash tables and discuss some of their properties. A hash table is an abstract data structure for maintaining a collection of (unique) items¹¹ and supporting efficient insert, delete, and find operations on data sets of size n . We assume that the item values belong to a finite universe $U = \{0, 1, \dots, q-1\}$, q prime.

3.2.6.2 Hashing

Given a set S of n elements from a finite universe $U = \{0, 1, \dots, q-1\}$, the *hashing* problem is to determine a function $h : S \rightarrow \{0, 1, \dots, s-1\}$, for some $s \leq q$ such that the following properties hold.

- h is “easy” to generate;
- h needs a “small” amount of additional space to store;
- h is “easy” to evaluate;
- h behaves “almost” as a random function.

In the remainder of this section we quantify the aforementioned properties and present a class of hash functions that possesses them.

A Universal Class of Hash Functions For the scope of this work we use a universal class [42] of hash functions initially introduced in [62] and further addressed in [63, 65].

Definition Let q be prime and $U = \{0, 1, \dots, q-1\}$ be a finite universe. Define

$$H_s^d = \{h : U \rightarrow \{0, 1, \dots, s-1\}\},$$

where $d \geq 1$ and $s \geq 1$ are integers, $b_0, b_1, \dots, b_d \in U$, and h is given by

$$h(x) = \left(\sum_{i=0}^d (b_i \cdot x^i) \bmod q \right) \bmod s, \quad x \in U.$$

Definition Let q be prime and $U = \{0, 1, \dots, q-1\}$ be a finite universe. Define

$$H_{r,s}^{d_1,d_2} = \{h : U \rightarrow \{0, 1, \dots, s-1\}\},$$

¹¹The explicit assumption in the forthcoming analysis of the hash table operations is that items are unique. In case of duplicate items, the upper bounds presented in the following sections do not apply. To remedy the situation, in section 3.2.8 we outline an efficient duplicate item elimination scheme that removes the requirement for unique items.

where $d_1, d_2 \geq 1$ and $r, s \geq 1$ are integers, $a_0, a_1, \dots, a_{r-1} \in \{0, 1, \dots, s-1\}$, and h is given by

$$h(x) = (\hat{f}(x) + a_{f(x)}) \bmod s, \quad x \in U, f \in H_{r,s}^{d_1} \text{ and } \hat{f} \in H_s^{d_2}.$$

Several of the intrinsic properties of the aforementioned classes of hash functions are established and discussed in some detail in appendix A.2.3.

Construction and Evaluation of $h \in H_{r,s}^{d_1,d_2}$ In this section we concentrate on BSP algorithms for the construction and evaluation of random members of $H_{r,s}^{d_1,d_2}$. Our main results are summarized in the following two propositions.

Proposition 3.2.1 *Let h be a randomly chosen function from $H_{r,s}^{d_1,d_2}$. Then, there exists a p -processor BSP algorithm for constructing h [with $O(d_1 + d_2 + \lceil r/p \rceil)$ space per processor] that for all integers n and p such that $p = O(n)$, requires time*

$$C_{c_hsh}(n, d_1, d_2, r, p) = O\left(d_1 + d_2 + \frac{r}{p}\right)$$

and

$$M_{c_hsh}(n, d_1, d_2, r, p) = M_{bcast}^{O(d_1+d_2)}(p) + M_{bcast}^{O(\lceil r/p \rceil)}\left(\left\lfloor \frac{p}{r} \right\rfloor\right),$$

for computation and communication respectively – we refer to this construction as construction CP. The overall time complexity of the algorithm (construction CP) is denoted by $T_{c_hsh}(n, d_1, d_2, r, p) = C_{c_hsh}(n, d_1, d_2, r, p) + M_{c_hsh}(n, d_1, d_2, r, p)$. Alternatively, the algorithm for constructing h [with $O(d_1 + d_2 + r)$ space per processor] requires time

$$C_{c_hsh^*}(n, d_1, d_2, r, p) = O(d_1 + d_2 + r)$$

and

$$M_{c_hsh^*}(n, d_1, d_2, r, p) = M_{bcast}^{O(d_1+d_2)}(p) + M_{bcast}^{O(r)}(p),$$

for computation and communication respectively – we refer to this construction as construction CS. The overall time complexity of the algorithm (construction CS) is denoted by $T_{c_hsh^*}(n, d_1, d_2, r, p) = C_{c_hsh^*}(n, d_1, d_2, r, p) + M_{c_hsh^*}(n, d_1, d_2, r, p)$.

Proof Processor p_0 selects at random functions $f \in H_r^{d_1}$ and $\hat{f} \in H_s^{d_2}$ in time $O(d_1 + d_2)$, and subsequently, broadcasts f and $\hat{f}$ to all other processors in $T_{bcast}^{O(d_1+d_2)}(p)$ time. The construction of h requires the additional selection of random $a_0, a_1, \dots, a_{r-1} \in \{0, 1, \dots, s-1\}$. To this end, processors $p_0, p_1, \dots, p_{\min\{p-1, r\}}$ select $a_0, a_1, \dots, a_{r-1}$ in $O(\lceil r/p \rceil)$ time. Then, the processors replicate each a_i , $0 \leq i < s$, $\lfloor p/r \rfloor$ times in $T_{bcast}^{O(\lceil r/p \rceil)}(\lfloor p/r \rfloor)$ time. The overall representation (construction CP) requires $O(d_1 + d_2 + \lceil r/p \rceil)$ space per processor.

Alternatively, processor p_0 selects $a_0, a_1, \dots, a_{r-1} \in \{0, 1, \dots, s-1\}$ in $O(r)$ time, and subsequently, broadcasts them to all other processors in $T_{broadcast}^{O(r)}(p)$ time. The overall alternative representation (construction CS) requires $O(d_1 + d_2 + r)$ space per processor and the desired result follows. $\square$

Proposition 3.2.2 *Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and h be randomly chosen from $H_{r,s}^{d_1,d_2}$. Assume any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq (d_1 + 1) \cdot r/2$. Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for evaluating (for construction CP in proposition 3.2.1) $h(x)$, for all $x \in S$, that for all integers n and p , and any u such that $p \leq n$, and $u > 0$, requires time [with probability $1 - O(\alpha \cdot (1/u)^{d_1+1} \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + n^{-\rho})$]*

$$C_{e_hsh}(n, r, u, p) = \tau(h) \cdot \left\lceil \frac{n}{p} \right\rceil$$

and

$$M_{e_hsh}(n, r, u, p) = L \cdot O(1) + g \cdot O(H_{dstr}),$$

for computation and communication respectively, where $\alpha = ((d+1)/2)^{d+2}/((d+1)/2)!$, $\tau(h)$ denotes the time required to evaluate $h(x)$ and

$$H_{dstr} = \begin{cases} (1+u) \cdot \left\lceil \frac{n}{r} \right\rceil \cdot \left\lceil \frac{r}{p} \right\rceil & \text{if } p \leq r, \\ O\left(\min\left\{\lg n, \frac{(1+u) \cdot n}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } p > r \text{ and } (1+u) \cdot \frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{(1+u) \cdot n}}\right)\right) \cdot \frac{(1+u) \cdot \lceil n/r \rceil}{\lfloor p/r \rfloor} & \text{otherwise.} \end{cases}$$

The overall time complexity of the evaluation algorithm (for construction CP) is denoted by $T_{e_hsh}(n, p) = C_{e_hsh}(n, p) + M_{e_hsh}(n, p)$. Alternatively, the algorithm for evaluating (for construction CS in proposition 3.2.1) $h(x)$, for all $x \in S$, requires time

$$C_{e_hsh^*}(n, p) = \tau(h) \cdot \left\lceil \frac{n}{p} \right\rceil,$$

for computation. The overall time complexity of the evaluation algorithm (for construction CS) is denoted by $T_{e_hsh^*}(n, p) = C_{e_hsh^*}(n, p)$.

Proof We assume that S is evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the evaluation process. The computation of $h(x) = (\hat{f}(x) + a_{f(x)}) \bmod s$, for all $x \in S$, involves the parameters associated with f , $\hat{f}$ and $a_{f(x)}$. Clearly, for construction CS in proposition 3.2.1, these parameters are local, and therefore, the evaluation of $h(x)$, for all $x \in S$, requires $\tau(h) \cdot \lceil n/p \rceil$ time. For construction CP in proposition 3.2.1, only the parameters associated with f and $\hat{f}$ are local. In the remainder of this proof we propose a randomized scheme for the determination of $a_{f(x)}$ that results in low contention.

Recall that by construction CP in proposition 3.2.1 there are $\lceil p/r \rceil$ copies of $a_{f(x)}$. Additionally, for any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq (d_1 + 1) \cdot r/2$, by way of lemma A.2.9, it follows that a randomly chosen $f \in H_r^{d_1}$ satisfies $b_i^f \leq (1+u) \cdot n^\delta$, for all $0 \leq i < r$ and any $u > 0$, with probability

$$1 - O\left(n^{1-\delta} \cdot \alpha \cdot \left(\frac{1}{u}\right)^{d_1+1} \cdot n^{-\delta \cdot (d_1+1)/2}\right) = 1 - O\left(\alpha \cdot \left(\frac{1}{u}\right)^{d_1+1} \cdot n^{1-\delta \cdot ((d_1+1)/2+1)}\right),$$

where $\alpha = ((d+1)/2)^{d+2}/((d+1)/2)!$. We distinguish two cases. In case $p \leq r$, then it is straightforward to verify that the maximum number of accesses to any single processor is at most $(1+u) \cdot \lceil n/r \rceil \cdot \lceil r/p \rceil$. In case $p > r$, then we assume that processors select uniformly at random one of the $\lfloor p/r \rfloor$ copies of $a_{f(x)}$. Then, for any constant $\rho > 0$, by invoking theorem A.1.2, it follows that the maximum number of accesses to any single processor is upper bounded as follows:

$$\begin{aligned} &O\left(\min\left\{\lg n, \frac{(1+u) \cdot n}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) && \text{if } \frac{(1+u) \cdot n}{p} = O(\lg n), \\ &\left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{(1+u) \cdot n}}\right)\right) \cdot \frac{(1+u) \cdot \lceil n/r \rceil}{\lfloor p/r \rfloor} && \text{otherwise,} \end{aligned}$$

with probability $1 - O(n^{-\rho})$. The desired result follows. $\square$

Remark We note that the time $\tau(h)$ required to evaluate h is upper bounded by $O(d_1 + d_2)$.

3.2.6.3 (δ, s) -Quasi-Random Functions

In the forthcoming sections we employ the aforementioned class of hash functions to obtain simple and efficient BSP algorithms for a multiplicity of problems. In order to simplify the exposition of these subsequent algorithms we assume the following definition and corollaries that are immediate applications of theorems A.2.1, A.2.2, A.2.3 and A.2.4, and propositions 3.2.1 and 3.2.2 respectively.

Definition Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and consider function h to be chosen at random from $H_{n^{1-\delta}, s}^{d_1, d_2}$, for constant integers d_1 and d_2 , and any constant $0 < \delta < 1$ such that $n^\delta \geq (d_1 + 1)/2$. Then, we say that h is (δ, s) -quasi-random for S . For the special case that $\delta = 1/2$ and $s = n$, we say that h is quasi-random for S .

Corollary 3.2.1 *Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and consider a (δ, s) -quasi-random function h . Then, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$ such that $d_1 \geq 2 \cdot (\rho + 1)/\delta - 3$ and $d_2 \geq (\rho + 1)/(\lg s / \lg n - \delta)$, and any $u > 0$ and $0 < \varepsilon < 1$,*

$$\Pr\left[b_j^h \geq (1+u) \cdot \frac{n}{s}\right] \leq \left(\frac{e^u}{(1+u)^{(1+u)}}\right)^{n/(d_2 \cdot s)}, \quad 0 \leq j < s \quad (3.1)$$

and

$$\Pr\left[b_j^h \geq (1+\varepsilon) \cdot \frac{n}{s}\right] \leq e^{-(1/3) \cdot \varepsilon^2 \cdot n/(d_2 \cdot s)}, \quad 0 \leq j < s. \quad (3.2)$$

Corollary 3.2.2 *Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n \leq s$, and consider a (δ, s) -quasi-random function h . Then, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$ such that $d_1 \geq 2 \cdot (\rho + 1)/\delta - 3$ and $d_2 \geq (\rho + 1)/(\lg s / \lg n - \delta)$, and any $u > 0$,*

$$\Pr\left[\sum_{j=0}^{s-1} (b_j^h)^k \geq (c_k + u) \cdot n\right] \leq 2 \cdot e^{-u^2 \cdot n^{(1/2) - o(1)}/8}, \quad \text{for any constant } k \geq 1,$$

where $c_k = (k-1)^{k-1} + \gamma \cdot e^{k-1}$ and $\gamma \leq (1 + s/q)^{d_2+1}$.

Corollary 3.2.3 Let q be prime, $\bar{S} \subseteq S \subseteq U = \{0, 1, \dots, q-1\}$, $|\bar{S}| = \bar{n}$, $|S| \leq n$, and consider a (δ, s) -quasi-random function h . Then, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$ such that $d_1 \geq 2 \cdot (\rho + 1)/\delta - 3$ and $d_2 \geq (\rho + 1)/(\lg s / \lg n - \delta)$, and any $u > 0$,

$$\Pr \left[\left| \{x \in S \mid \exists \bar{x} \in \bar{S} \ (h(x) = h(\bar{x}))\} \right| \geq \left((1+u) \cdot \frac{n}{s} + d_2 \right) \cdot \bar{n} \right] \leq \left(2 \cdot n^\delta \cdot \bar{n} \right)^{d_2+1} \cdot \left(\frac{1}{s} \right)^{d_2} + \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n \cdot \bar{n} / (2 \cdot d_2 \cdot n^\delta \cdot s)}.$$

Corollary 3.2.4 Let q be prime, $\bar{S} \subseteq S \subseteq U = \{0, 1, \dots, q-1\}$, $|\bar{S}| = \bar{n}$, $|S| \leq n$, $\bar{J} = \{j \mid 0 \leq j < s \text{ and } B_j^h \cap \bar{S} \neq \{\}\}$, and consider a (δ, s) -quasi-random function h . Then, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$ such that $d_1 \geq 2 \cdot (\rho + 1)/\delta - 3$ and $d_2 \geq (\rho + 1)/(\lg s / \lg n - \delta)$, and any $u > 0$,

$$\Pr \left[b_j^h \geq (1+u) \cdot \frac{n}{s} + d_2 \right] \leq \left(2 \cdot n^\delta \cdot \bar{n} \right)^{d_2+1} \cdot \left(\frac{1}{s} \right)^{d_2} + \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n / (d_2 \cdot s)}, \quad j \in \bar{J}.$$

Corollary 3.2.5 Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and h be a (δ, s) -quasi-random function. Then, there exists a p -processor BSP algorithm for constructing h [with $O(\lceil n^{1-\delta}/p \rceil)$ space per processor] that for all integers n and p such that $p = O(n)$, requires time

$$C_{c\text{-}qrnd}(n, \delta, p) = O \left(\left\lceil \frac{n^{1-\delta}}{p} \right\rceil \right)$$

and

$$M_{c\text{-}qrnd}(n, \delta, p) = M_{bcast}^{O(1)}(p) + M_{bcast}^{O(\lceil n^{1-\delta}/p \rceil)} \left(\left\lceil \frac{p}{n^{1-\delta}} \right\rceil \right),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{c\text{-}qrnd}(n, p) = C_{c\text{-}qrnd}(n, p) + M_{c\text{-}qrnd}(n, p)$.

Corollary 3.2.6 Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and h be a (δ, s) -quasi-random function. Then, for any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for evaluating $h(x)$, for all $x \in S$, that for all integers n, p and d_1 , and any u such that $p \leq n$, $d_1 \geq 2 \cdot (\rho + 1 - \delta)/(2 \cdot \lg u / \lg n + \delta) - 1$ and $u > n^{-(\delta/2)+\zeta}$ (for arbitrarily small positive constant $\zeta \leq 1/4$), requires time [with probability $1 - O(n^{-\rho})$]

$$C_{e\text{-}qrnd}(n, \delta, u, p) = \tau(h) \cdot \left\lceil \frac{n}{p} \right\rceil$$

and

$$M_{e\text{-}qrnd}(n, \delta, u, p) = L \cdot O(1) + g \cdot O(H_{dstr}),$$

for computation and communication respectively, where $\tau(h)$ denotes the time required to evaluate h and

$$H_{dstr} = \begin{cases} (1+u) \cdot \lceil n^\delta \rceil \cdot \left\lceil \frac{n^{1-\delta}}{p} \right\rceil & \text{if } p \leq n^{1-\delta}, \\ O \left(\min \left\{ \lg n, \frac{(1+u) \cdot n}{p} \cdot \frac{\lg n}{\lg \lg n} \right\} \right) & \text{if } p > n^{1-\delta} \text{ and } (1+u) \cdot \frac{n}{p} = O(\lg n), \\ \left(1 + O \left(\sqrt{\frac{p \cdot \lg n}{(1+u) \cdot n}} \right) \right) \cdot \frac{(1+u) \cdot \lceil n^\delta \rceil}{\lfloor p/n^{1-\delta} \rfloor} & \text{otherwise.} \end{cases}$$

The overall time complexity of the evaluation algorithm is denoted by $T_{e_grnd}(n, \delta, u, p) = C_{e_grnd}(n, \delta, u, p) + M_{e_grnd}(n, \delta, u, p)$.

Remark The time $\tau(h)$ required to evaluate a (δ, s) -quasi-random function h is upper bounded by $O(d_1 + d_2) = O(1)$.

3.2.7 Optimal Randomized Inserting, Deleting and Finding Algorithms on Parallel Hash Tables

The following operations are defined on a hash table HT .

- (1) INSERT-HT($\{y_1, y_2, \dots, y_n\}, HT$) for the insertion of $\{y_1, y_2, \dots, y_n\}$ into HT .
- (2) DELETE-HT($\{y_1, y_2, \dots, y_n\}, HT$) for the deletion of $\{y_1, y_2, \dots, y_n\}$ from HT .
- (3) FIND-HT($\{y_1, y_2, \dots, y_n\}, HT$) for the determination of $\{e_1, e_2, \dots, e_n\}$, where $e_i = y_i$ if $y_i \in HT$, and $e_i = \{\}$ otherwise, for all $1 \leq i < n$.

The implicit assumption is that there exists a CONSTRUCT-HT(HT) operation for the construction of an empty table HT . In contrast to previous approaches (e.g., [26, 178]), our implementation has the advantage of supporting hash table operations on data sets of varying size – in [26, 178], insert, delete and find operations have to be performed on data sets of predetermined size, by contrast, in this work we support data sets of varying size.

3.2.7.1 Static Hash-Table Algorithms

In this section we present a randomized BSP algorithm for the static hash table operations (construct and find). More precisely, we demonstrate how to employ hash tables in order to embody a static set of items induced by processing a sequence of n queries containing the unique instruction of finding an element. The algorithm derives from the ideas of [65, 74] and utilizes the notion of *secondary hashing*. The algorithm is fairly simple and achieves optimality for an ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal randomized hash-table construction and find algorithms are synthetic generalizations of the sequential hash-table construction and find algorithms of [74] – with the noted difference that the underlying hash functions of the algorithms of [74] are non-trivially substituted by the more powerful (δ, m) -quasi-random hash functions.

Our main result is summarized in the following two theorems.

Theorem 3.2.2 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for constructing a hash table on m items, that for all integers m and p such that $p = O(m/(\lg m \cdot \lg \lg m))$, requires time [with probability $1 - O(m^{-\rho})$] $O(m/p)$ and $L \cdot O(\lg p) + g \cdot O(m/p)$, for computation and communication respectively.*

Theorem 3.2.3 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting operation find on a hash table of size m , that for all integers n , m and p such that $m \geq n$ and*

$p = O(n/\lg n)$, requires time [with probability $1 - O(n^{-\rho})$] $O(n/p)$ and $L \cdot O(1) + g \cdot O(n/p)$, for computation and communication respectively.

The proof of the aforementioned results is diverged to the end of this section. The remainder of this section is devoted to the proof of the following two propositions. Then, theorems 3.2.2 and 3.2.3 follow easily as a consequence of these results.

Proposition 3.2.3 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for constructing a hash table on m items, that for all integers m and p , and constant δ such that $p = O(m \cdot \lg \lg m / \lg^2 m)$ and $0 < \delta < 1$, requires time [with probability $1 - O(m^{-\rho})$]*

$$C_{c_hsh}(m, \delta, p) = O\left(\frac{m}{p}\right) + C_{c_qrnd}(m, \delta, p) + C_{e_qrnd}(m, \delta, 1, p) + C_{ppf}(p)$$

and

$$M_{c_hsh}(m, \delta, p) = L \cdot O(1) + g \cdot O\left(\frac{m}{p}\right) + M_{c_qrnd}(m, \delta, p) + M_{e_qrnd}(m, \delta, 1, p) + M_{ppf}(p),$$

for computation and communication respectively. Furthermore, for $p = \omega(m \cdot \lg \lg m / \lg^2 m)$ and $p = O(m/(\lg m \cdot \lg \lg m))$, the algorithm requires additional time

$$C_{rdx_srt}\left(k \cdot O(\lg m), k^2 \cdot O(\lg m), \Omega\left(\left\lceil \frac{k}{\lceil m/(p \cdot \lg m \cdot \lg \lg m) \rceil} \right\rceil\right)\right) + \\ C_{lbal}\left(O\left(\frac{m}{\lg m \cdot \lg \lg m}\right), O\left(\frac{m}{p}\right), p\right) + C_{ppf}(k) + O\left(\lg m + \frac{m}{p \cdot \lg \lg m}\right)$$

and

$$M_{rdx_srt}\left(k \cdot O(\lg m), k^2 \cdot O(\lg m), \Omega\left(\left\lceil \frac{k}{\lceil m/(p \cdot \lg m \cdot \lg \lg m) \rceil} \right\rceil\right)\right) + \\ M_{lbal}\left(O\left(\frac{m}{\lg m \cdot \lg \lg m}\right), O\left(\frac{m}{p}\right), p\right) + M_{bcast}^{O(\lg m)}(k) + M_{ppf}(k),$$

for computation and communication respectively, where $k = O(\lg m / \lg \lg m)$. The overall time complexity of the algorithm is denoted by $T_{c_hsh}(m, \delta, p) = C_{c_hsh}(m, \delta, p) + M_{c_hsh}(m, \delta, p)$. The algorithm requires $O(m/p)$ space per processor.

Proof We assume that the input is evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm. Moreover, we assume that the items are distinct; we later explain how we can remove such an assumption and still maintain the desired time and space bounds.

Procedure CONSTRUCT_HT

begin

[Stage 1] A (δ, m) -quasi-random function h is constructed – we refer to this function as the *primary hash function*. We assume sufficiently large constants d_1 and d_2 in the construction of h so as to satisfy the conditions of corollaries 3.2.1, 3.2.2 and 3.2.6.

[**Analysis of Stage 1**] By way of corollary 3.2.5, this stage requires time $T_{c_qrd}(m, \delta, p)$.

[**Stage 2**] Each processor determines the processor destination of each item in its possession by evaluating the function $h' = \lfloor h/(m/p) \rfloor$ – item y is destined to processor $p_{h'(y)}$.

[**Analysis of Stage 2**] For any constant $\rho > 0$, by way of corollary 3.2.6, this stage requires time $T_{e_qrd}(m, \delta, 1, p)$, with probability $1 - O(m^{-\rho})$.

[**Stage 3**] Each processor knows where to route its items, and performs the routing – item x along with its hashed value $h(y)$ is communicated to processor $p_{h'(y)}$.

[**Analysis of Stage 3**] In view of lemma A.2.5 and the fact that $p = O(m/(\lg m \cdot \lg \lg m))$, it is not difficult to verify that h' shares the properties in corollary 3.2.1 of $(\delta, m/(\lg m \cdot \lg \lg m))$ -quasi-random functions. Thus, for any constant $\rho > 0$, by way of corollary 3.2.1, processors receive $O(m/p)$ items, with probability $1 - O(m^{-\rho})$. Hence, the communication cost of this stage is bounded above by $L + g \cdot O(m/p)$.

[**Stage 4**] A memory block of size $2 \cdot (b_j^h)^2 + 1$ is allocated for each bucket B_j^h , $0 \leq j < m$ – we refer to this memory block as the *secondary hash table* ST_j . Additionally, a memory block of size $O(m)$ is allocated for the secondary hash functions to be introduced in the next stage – we refer to this memory block as the *primary hash table* HT . We assume that the primary and secondary hash tables are evenly and contiguously distributed amongst the processors.

[**Analysis of Stage 4**] In order to achieve an even and contiguous distribution of the secondary hash tables a prefix operation on at most m bucket sizes ($O(m/p)$ per processor) is realized. Thus, this stage requires time $O(m/p) + C_{ppf}(p)$ and $M_{ppf}(p)$ for computation and communication respectively. For any constant $\rho > 0$, by way of corollary 3.2.2, the total memory allocated in this stage is upper bounded by $O(m)$, with probability $1 - O(m^{-\rho})$.

[**Stage 5**] Let $J = \{j \mid 1 \leq b_j^h \leq 2 \cdot \lg \lg m\}$. Each processor constructs a random function $h_j \in H_{2(b_j^h)^2+1}^1$ for each bucket B_j^h , $j \in J$, in its possession – we refer to this function as the *secondary hash function* h_j . If the secondary function h_j is one-perfect for B_j^h , then h_j is stored in local memory and B_j^h is set to $\{\}$. This stage is repeatedly executed until $J = \{\}$.

[**Analysis of Stage 5**] Consider an arbitrary processor p_i , $0 \leq i < p$, and let $J_i = \{j \mid 1 \leq b_j^h \leq 2 \cdot \lg \lg m \text{ and } B_j^h \text{ resides in the local memory of processor } p_i\}$. Define (for processor p_i) the random variable Y_j as follows:

$$Y_j = \text{time required for the construction of one-perfect } h_j, \quad j \in J_i.$$

The random function h_j is constructed in $O(1)$ time. Furthermore, the injectiveness of h_j is tested in $O(b_j^h \cdot (\lg(2 \cdot (b_j^h)^2 + 1)/\lg b_j^h) + b_j^h) = O(b_j^h)$ time by employing algorithm SEQ_RADIX_SORT (refer to section 3.2.2). The test implies radix sorting of b_j^h integers in the range $[0, 1, \dots, 2 \cdot (b_j^h)^2]$, and subsequently, scanning this sorted sequence. By way of lemma A.2.7, the probability that a randomly chosen $h_j \in H_{2(b_j^h)^2+1}^1$ is one-perfect for B_j^h is at least $1/2$. Therefore, for any $k \geq 1$, we have that

$$\Pr \left[Y_j = k \cdot O \left(b_j^h \right) \right] \leq 2^{-k}, \quad j \in J_i.$$

By way of the analysis of stage 3 we have that $\sum_{j \in J_i} b_j^h = O(m/p)$ and by definition $b_j^h \leq 2 \cdot \lg \lg m$. Therefore, by invoking lemma A.1.3, we conclude that, for any constant $\rho > 0$,

$$\Pr \left[\exists i \left(\sum_{j \in J_i} Y_j = O \left(\frac{m}{p} \right) \right) \right] = 1 - O(m^{-\rho}), \quad 0 \leq i < p.$$

Thus, this stage requires time $O(m/p)$.

[Stage 6] Let $\bar{J} = \{j \mid b_j^h > 2 \cdot \lg \lg m\}$. The elements in $\cup_{j \in \bar{J}} B_j^h$ are load-balanced amongst the processors, maintaining the items of each distributed bucket in the memory of contiguous processors.

[Analysis of Stage 6] For any constant $\rho > 0$, by way of corollary A.2.1,

$$\Pr \left[\exists j \left(b_j^h = O \left(\frac{\lg m}{\lg \lg m} \right) \right) \right] = 1 - O(m^{-\rho}), \quad 0 \leq j < m.$$

Furthermore, in [98] it is shown that,

$$\Pr \left[\exists k \left(\left| \{j \mid b_j^h \geq k\} \right| = O \left(\frac{m}{2^k} \right) \right) \right] = 1 - O(m^{-\rho}), \quad 0 \leq j < m.$$

We conclude that, for any constant $\rho > 0$,

$$\Pr \left[\sum_{j \in \bar{J}} b_j^h = O \left(\frac{m}{\lg m \cdot \lg \lg m} \right) \right] = 1 - O(m^{-\rho}).$$

Thus, the time required for this stage is upper bounded by $T_{\text{bal}}(O(m/(\lg m \cdot \lg \lg m)), O(m/p), p)$.

[Stage 7] The first processor associated with each bucket B_j^h , $j \in \bar{J}$, constructs l random functions $h_{j,1}, h_{j,2}, \dots, h_{j,l} \in H_{2 \cdot (b_j^h)^2 + 1}^1$ for B_j^h and subsequently broadcasts these functions to all processors associated with bucket B_j^h , where $l = O(\lg m)$. The first $h_{j,i}$, $1 \leq i \leq l$, that is one-perfect for B_j^h is stored in the local memory of the first processor associated with B_j^h – we refer to this function as the *secondary hash function* h_j .

[Analysis of Stage 7] By way of the load-balancing operation of stage 6, each processor maintains $O(\lceil m/(p \cdot \lg m \cdot \lg \lg m) \rceil)$ items of $\cup_{j \in \bar{J}} B_j^h$. Therefore, the construction of the l random functions (per bucket) takes $O(\lg m + m/(p \cdot \lg \lg m))$ time. Moreover, by way of lemma A.2.7, the probability that a randomly chosen $h_{j,i} \in H_{2 \cdot (b_j^h)^2 + 1}^1$, $1 \leq i \leq l$, is one-perfect for B_j^h is at least $1/2$. Therefore, for any constant $\rho > 0$ and $l = O(\lg m)$, with probability $1 - O(m^{-\rho})$, at least one of the l hash functions is one-perfect for B_j^h .

In the analysis of stage 6 it is shown that the maximum bucket size is $O(\lg m / \lg \lg m)$. Thus, the broadcasting operation requires $M_{bcst}^{O(\lg m)}(k)$ time for communication, where $k = O(\lg m / \lg \lg m)$. Furthermore, since each processor shares at most two buckets with other processors, injectiveness is tested (along the lines of stage 5) in time

$$O\left(\left\lceil \frac{m}{p \cdot \lg m \cdot \lg \lg m} \right\rceil \cdot \lg m\right) + C_{rdx_srt}\left(k \cdot O(\lg m), k^2 \cdot O(\lg m), \Omega\left(\left\lceil \frac{k}{\lceil m/(p \cdot \lg m \cdot \lg \lg m) \rceil} \right\rceil\right)\right)$$

and

$$M_{rdx_srt}\left(k \cdot O(\lg m), k^2 \cdot O(\lg m), \Omega\left(\left\lceil \frac{k}{\lceil m/(p \cdot \lg m \cdot \lg \lg m) \rceil} \right\rceil\right)\right),$$

for computation and communication respectively, where $k = O(\lg m / \lg \lg m)$. Finally, the selection of one out of the (at most) l one-perfect hash functions is realized by way of a parallel prefix operation in time $O(\lceil m/(p \cdot \lg m \cdot \lg \lg m) \rceil \cdot \lg m) + C_{ppf}(k)$ and $M_{ppf}(k)$ for computation and communication respectively.

[Stage 8] Each processor knows where to route its items, and performs the routing – item x is communicated to memory cell $ST_{h(x)}[h_{h(x)}(x)]$. Additionally, secondary hash function h_j is stored in memory cell $HT[j]$.

[Analysis of Stage 8] By way of the analysis of stage 4, each processor maintains a segment of size $O(m/p)$ from tables HT and $\cup_j ST_j$. Therefore, this stage requires $L + g \cdot O(m/p)$ communication time.

end

Remark Stages 6 and 7 can be safely removed, without affecting the asymptotic analysis of the algorithm, by defining J in stage 5 as follows:

$$J = \{j \mid b_j^h = O(\lg m / \lg \lg m)\}, \quad 0 \leq j < m.$$

Then, the process involved in stages 6 and 7 is redundant. Unfortunately, to achieve the desired probability bound the condition $p = O(m/(\lg m \cdot \lg \lg m))$ has to be relaxed to $p = O(m \cdot \lg \lg m / \lg^2 m)$.

Analysis The correctness of the algorithm follows from its description. By summing up the computation and communication requirements of procedure `CONSTRUCT_HT` we obtain the desired time bound. The probability bound follows by assuming sufficiently large d_1 and d_2 in the construction of corollary 3.2.5 – $d_1 \geq 2 \cdot (\rho + 1)/\delta - 3$ and $d_2 \geq (\rho + 1)/(1 - \delta - o(1))$ are sufficient (refer to corollaries 3.2.1, 3.2.2 and 3.2.6). Finally, the space requirement follows from the analysis of stage 4.

The proposition follows. $\square$

Proposition 3.2.4 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operation `find` on a hash table of size m , that for all integers n , m and p , and constant δ such that $m \geq n$, $p = O(n/\lg n)$ and $0 < \delta < 1$, requires time [with probability $1 - O(n^{-\rho})$]*

$$C_{hsh}(n, \delta, p) = O\left(\frac{n}{p}\right) + C_{e_qrnd}(n, \delta, 1, p)$$

and

$$M_{hsh}(n, \delta, p) = L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + C_{e_qrnd}(n, \delta, 1, p),$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{hsh}(n, \delta, p) = C_{hsh}(n, \delta, p) + M_{hsh}(n, \delta, p)$.

Proof We initially present the procedure to realize the `find` operation on a hash table that is constructed by the algorithm implied in proposition 3.2.3; later, we provide a correctness proof and an analysis of the resources required. We assume that the input is evenly but otherwise arbitrarily distributed among the p processors before the beginning of the execution of the algorithm. Moreover, we assume that the items are distinct; we later explain how we can remove such an assumption and still maintain the desired time bound. Some notation is required before proceeding to the details of the algorithm. By h we denote the (δ, m) -quasi-random function that serves as the primary hash function of the underlying primary hash table HT , and by h' we denote the composite function $\lfloor h/(m/p) \rfloor$. Accordingly, by h_j , $0 \leq j < m$, we denote the function that serves as the secondary hash function of the underlying secondary hash table ST_j .

The procedure to manage the `find` operation is `FIND_HT` ($\{y_1, y_2, \dots, y_n\}, HT$) (refer to figure 3.8), where $Y = \{y_1, y_2, \dots, y_n\}$ denotes the set of n items to be found in hash table HT .¹²

```

begin FIND_HT ( $\{y_1, y_2, \dots, y_n\}, HT$ )
1.   do denote by  $Y$  set  $\{y_1, y_2, \dots, y_n\}$  ;
2.   for each processor  $p^{(k)}$ ,  $k \in \{0, 1, \dots, p-1\}$ , in parallel
3.       do let  $E^{(k)} = \{y \mid y \in Y \text{ and } h'(y) = k\}$  ;
4.       report (as found) set  $\{x \mid x \in E^{(k)} \text{ and } x = ST_{h(x)}[h_{h(x)}(x)]\}$  ;
end FIND_HT

```

Figure 3.8: Procedure `FIND_HT`, p – number of processors.

The correctness of the procedure follows from its description and the construction implied in proposition 3.2.3. The time required for procedure `FIND_HT` is determined as follows. For any constant $\rho > 0$, by way of corollary 3.2.6, the hash function evaluation (line 3) requires time $T_{e_qrnd}(n, \delta, 1, p)$, with probability $1 - O(n^{-\rho})$. Accordingly, by way of corollary 3.2.1, processors receive $O(n/p)$ items, with probability $1 - O(n^{-\rho})$, and therefore, the routing operation (line 3) takes $L + g \cdot O(n/p)$ communication time. Finally, by way of the analysis of stages 4 and 8 of procedure `CONSTRUCT_HT`, the “reporting” operation (line 4) requires time $O(n/p)$ and $L + g \cdot O(n/p)$ for computation and communication respectively.

¹²With reference to the pseudo-code of this section, the superscript notation $\langle k \rangle$ refers to processor p_k (or equivalently, processor $p^{(k)}$) and implies the “local segment” of that processor.

Analysis The correctness of the algorithm follows from its description. By summing up the computation and communication requirements of procedure FIND-HT. The probability bound follows by assuming sufficiently large d_1 and d_2 in the construction of proposition 3.2.3 – $d_1 \geq 2 \cdot (\rho + 1)/\delta - 3$ and $d_2 \geq (\rho + 1)/(1 - \delta - o(1))$ are sufficient (refer to corollaries 3.2.1, 3.2.2 and 3.2.6).

The proposition follows. $\square$

Proof (of theorem 3.2.2) The theorem follows easily from proposition 3.2.3, for any constant $0 < \delta < 1$, by substituting for T_{c_qnd} , T_{e_qnd} , T_{rdx_srt} , T_{lbal} , T_{bcast} and T_{ppf} the corresponding expressions in corollaries 3.2.5 and 3.2.6, and propositions 2.4.1, 2.1.5, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

Proof (of theorem 3.2.3) The theorem follows easily from proposition 3.2.4, for any constant $0 < \delta < 1$, by substituting for T_{e_qnd} the corresponding expression in corollary 3.2.6. $\square$

3.2.7.2 Dynamic Hash-Table Algorithms

In this section we present randomized BSP algorithms for the dynamic hash table operations (construct, insert, delete and find). More precisely, we demonstrate how to employ hash tables in order to embody a dynamically changing set of items induced by processing a sequence of n queries containing the instructions of (i) inserting an element, (ii) deleting an element, and (iii) finding an element. The algorithm derives from the ideas of [62, 65, 74] and utilizes the notion of *secondary hashing*. The algorithm is fairly simple and achieves optimality for a ample range of the parameters n , p , L and g .

Informal Description of Algorithm The optimal randomized hash-table insert, delete and find algorithms are synthetic generalizations of the sequential hash-table insert, delete and find algorithms of [65, 74] – with the noted difference that the underlying hash functions of the algorithms of [65, 74] are non-trivially substituted by the more powerful (δ, m) -quasi-random hash functions.

Our main result is summarized in the following theorem.

Theorem 3.2.4 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete and find on a hash table of size m , that for all integers n , m and p such that $m \geq n$ and $p = O(n/(\lg n \cdot \lg \lg n))$, requires time [with probability $1 - O(n^{-\rho})$]*

$$O\left(\frac{n}{p}\right) \quad \text{for insert, delete and find,}$$

and

$$\begin{aligned} &L \cdot O\left(\left\lceil \min\left\{1, \frac{n}{\sqrt{m}}\right\} \cdot \lg p \right\rceil\right) + g \cdot O\left(\frac{n}{p}\right) \quad \text{for insert,} \\ &L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) \quad \text{for delete/find,} \end{aligned}$$

for computation and communication respectively.

The proof of the aforementioned results is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 3.2.4 follows easily as a consequence of this result.

Proposition 3.2.5 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete and find on a hash table of size m , that for all integers n , m and p , and constant δ such that $m \geq n$, $p = O(n/(\lg n \cdot \lg \lg n))$ and $0 < \delta < 1/2$, requires time [with probability $1 - O(n^{-\rho})$]*

$$C_{hsh}(n, \delta, p) = \begin{cases} \left(O\left(\frac{n}{p}\right) + C_{e_qrnd}(n, \delta, 1, p) \right) \cdot [n < \sqrt{m}] + \\ \frac{1}{\lceil \sqrt{m}/n \rceil} \cdot C_{c_hsh}(O(\max\{\sqrt{m}, n\}), \delta, p) & \text{for insert,} \\ O\left(\frac{n}{p}\right) + C_{e_qrnd}(n, \delta, 1, p) & \text{for delete/find,} \end{cases}$$

and

$$M_{hsh}(n, \delta, p) = \begin{cases} \left(L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + M_{e_qrnd}(n, \delta, 1, p) \right) \cdot [n < \sqrt{m}] + \\ \frac{1}{\lceil \sqrt{m}/n \rceil} \cdot M_{c_hsh}(O(\max\{\sqrt{m}, n\}), \delta, p) & \text{for insert,} \\ L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) + C_{e_qrnd}(n, \delta, 1, p) & \text{for delete/find,} \end{cases}$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{hsh}(n, \delta, p) = C_{hsh}(n, \delta, p) + M_{hsh}(n, \delta, p)$.

Proof The parallel dynamic hash table is maintained as a collection of disjointed tables. In particular, along the lines of [62], the dynamic hash table consists of a *primary hash table* and of a *cache hash table* both of size $O(m)$. Items are initially inserted into the cache and later transferred (as a background process) to the permanent primary hash table. To this end, two hash tables of size $O(m)$ are alternately employed as the cache – new items are inserted into one of the two cache tables (*foreground*) whilst the other table (*background*) is flashed into the permanent primary hash table. In our implementation, the primary table is essentially a static hash table (refer to sub-section 3.2.7.1). The truly dynamic hash table is the cache.

Initially, the two caches are empty. The primary hash table is constructed by way of the algorithm implied in proposition 3.2.3. Let (δ, m) -quasi-random function h be the primary hash function associated with the so constructed primary table; then, h also serves as the primary hash function for the cache tables. We assume sufficiently large d_1 and d_2 in the construction of h so as to satisfy the conditions of corollaries 3.2.1, 3.2.2 and 3.2.6.

The dynamic nature of the overall structure is based on the fact that the number of items maintained at any time in the cache tables is considerably smaller than the corresponding number in the primary table. In particular, the underlying idea is to maintain m items in the primary table and at most $2 \cdot \sqrt{m}$ items in the cache table. An informal description of the BSP algorithm follows.

Let $n \leq m$ denote the number of items involved in insert, delete and find operations. In the remainder of this proof we assume that $p = O(n/(\lg n \cdot \lg \lg n))$ and $n < \sqrt{m}$; the case $n \geq \sqrt{m}$ is treated at the end of this discussion. For the insertion of n new items (at most $\lceil n/p \rceil$ per processor), each processor sends its (at most) $\lceil n/p \rceil$ items to uniformly at (δ, p) -quasi-random distributed processors by employing function $h' = \lfloor h/(m/p) \rfloor$. In view of lemma A.2.5 and the fact that $p = O(n/(\lg n \cdot \lg \lg n))$, it is not difficult to verify that h' shares the properties described in corollary 3.2.1 of $(\delta, n/(\lg n \cdot \lg \lg n))$ -quasi-random functions. Hence, for any constant $\rho > 0$, by way of corollary 3.2.1, processors receive $O(n/p)$ items, with probability $1 - O(n^{-\rho})$. We conclude that this step requires time $C_{e_qrnd}(n, \delta, 1, p)$ and $L + g \cdot O(n/p) + M_{e_qrnd}(n, \delta, 1, p)$ for computation and communication respectively. Furthermore, by way of lemma A.2.7, primary function h is d_2 -perfect for the input set, with probability

$$\Pr[h \text{ is } d_2\text{-perfect for the input set}] = 1 - O\left(n \cdot \left(\frac{n}{m}\right)^{d_2}\right) = 1 - O(n^{1-d_2}) = 1 - O(n^{-\rho}),$$

for any $d_2 \geq \rho + 1$. We conclude that primary function h maps at most d_2 items per cache table position. Therefore, a foreground cache table of size $O(d_2 \cdot m) = O(m)$ is sufficient for supporting the insertion of $n = O(\sqrt{m})$ new items (at most $\lceil n/p \rceil$ per processor) in time $O(n/p) + C_{e_grnd}(n, \delta, 1, p)$ and $L + g \cdot O(n/p) + M_{e_grnd}(n, \delta, 1, p)$ for computation and communication respectively. Additionally, if a sequential list – consisting of the elements that are currently in the cache table – is maintained, then the cache table can also be reinitialized in $O(n/p)$ time. In case the number of items in the cache table exceeds $\sqrt{m}$, then the roles of the foreground and background cache tables are interchanged.

It remains to establish the time requirements associated with transferring the background cache table into the permanent primary table. Recall that this transfer is realized as a background process, and therefore, has to be accounted in the time complexity of the overall insertion process. By way of corollary 3.2.3, under primary function h , the number of items in the cache table that collide with themselves or with items from the primary table is $O(d_2 \cdot \sqrt{m}) = O(\sqrt{m})$, with probability

$$1 - O\left(n^{((1/2)+\delta) \cdot (d_2+1) - d_2}\right) = 1 - O(n^{-\rho}),$$

for any $0 < \delta < 1/2$ and $d_2 \geq (2 \cdot \rho + 1 + 2\delta)/(1 - 2 \cdot \delta)$. Furthermore, by comparing corollaries 3.2.1 and 3.2.4, we conclude that the bucket size distribution of these items is not significantly different to the “ideal” distribution of the combined primary and cache tables. In particular, the bucket size associated with items in the cache table that collide with themselves or with items from the primary table is merely increased by an additive constant factor of d_2 . Therefore, algorithm CONSTRUCT_HT can be employed to efficiently rehash these colliding items in time $C_{c_hsh}(O(\sqrt{m}), \delta, p)$ and $M_{c_hsh}(O(\sqrt{m}), \delta, p)$, for computation and communication respectively. The analysis pertaining the aforementioned claim is similar to that of algorithm CONSTRUCT_HT and is therefore omitted.

In case $n \geq \sqrt{m}$, then the n items to be inserted along with the $O(\sqrt{m})$ items of the cache (foreground and background) are directly inserted in the primary hash table as outlined in the preceding discussion in time $C_{c_hsh}(O(n), \delta, p)$ and $M_{c_hsh}(O(n), \delta, p)$ for computation and communication respectively. Evidently, the resulting primary table is identical to the static table of sub-section 3.2.7.1, and hence, requires the same resources. Thus, we conclude the desired time bound on the insert operation.

For the deletion of n new items (at most $\lceil n/p \rceil$ per processor) an adaptation of the algorithm implied in proposition 3.2.4 can be shown to work efficiently for the dynamic hash tables of this section. In particular, deletions are performed by attaching a tag “deleted” to the item to be removed – only when a new primary hash function h or a new secondary function $h_j ST_j$ is chosen, does the algorithm remove the elements with a tag “deleted” from the hash table. Note however that the dynamic hash table presented in this section consists of three disjointed tables – the primary and the two caches. Hence, the interaction of insert and delete operations has to be analyzed within this disjointed-table framework. In particular, delete operations should treat the hash tables in the following order: foreground cache table, background cache table and primary cache table. Thus, by way of proposition 3.2.4, we conclude the desired time bound on the delete and find operations.

Analysis The correctness of the algorithm follows from its description. The probability bound follows by assuming sufficiently large d_1 and d_2 in the construction of corollary 3.2.5 – $d_1 \geq 2(\rho + 1)/\delta - 3$ and

$d_2 \geq (2\rho + 1 + 2\delta)/(1 - 2\delta)$ are sufficient (refer to corollaries 3.2.1, 3.2.2 and 3.2.6 and the conditions imposed in the preceding analysis).

Remark In the aforementioned discussion we have assumed that the size of the primary hash table is $O(m)$. For the case that the size of the permanent primary table fluctuates, standard doubling techniques for managing growing tables show that the analysis is still valid. To maintain however the real-time property of our algorithm we need to introduce two further temporary tables – new items are inserted into one of the two temporary tables whilst the other temporary table is flashed into the permanent primary hash table (for the details of this technique refer to [62]).

The proposition follows. $\square$

Proof (of theorem 3.2.4) The theorem follows easily from proposition 3.2.3 and 3.2.5, for any constant $0 < \delta < 1/2$, by substituting for T_{c_qnd} , T_{e_qnd} , T_{rdx_srt} , T_{lbal} , T_{bcast} and T_{ppf} the corresponding expressions in corollaries 3.2.5 and 3.2.6, and propositions 2.4.1, 2.1.5, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

3.2.8 Duplicate Item Handling

The explicit assumption in the analysis of the hash table operations of section 3.2.7 is that items are unique; in case of duplicate items, the upper bounds presented in the previous sections do not apply. To remedy the situation, in the remainder of this section we present a simple randomized scheme for the elimination of duplicates.

Informal Description of Algorithm The optimal randomized duplicate item elimination algorithm is an immediate adaptation of the integer-sorting algorithm of section 2.4.4 – with the utilization of the hashing ideas of appendix A.2.3 for the actualization of the elimination process.

Our main result is summarized in the following theorem.

Theorem 3.2.5 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for eliminating duplicate items from a multi-set of size n in the range $[0, 1, \dots, q-1]$, q prime, that for all integers q , n and p such that $n \leq q$ and $p = O(n)$, requires time [with probability $1 - O(n^{-\rho})$] $O((n/p + \lg p) \cdot (\lg n / \lg(n/p + \lg p)))$ and $L \cdot O(\lg p \cdot (\lg n / \lg(n/p + \lg p))) + g \cdot O((n/p + \lg p) \cdot (\lg n / \lg(n/p + \lg p)))$, for computation and communication respectively. The algorithm requires $O(n/p + \lg p)$ space.*

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of a technical result. Then, theorem 2.4.1 follows easily as a consequence of this result.

One technique for eliminating duplicates is to sort items according to their value so that items with the same value are adjacent in the sorted output. A multiplicity of BSP comparison-based sorting algorithms can be employed to realize the aforementioned scheme (refer to [75, 77, 81, 82, 87, 101] and sections 2.6.4, 2.6.6, 2.6.7, 2.6.8). For the scope of this work, however, we employ the BSP integer sorting algorithm of section 2.4.4. The choice of integer sorting (rather than comparison sorting) is motivated by the fact that the item values are known to belong to some finite universe $U = \{0, 1, \dots, q-1\}$. Unfortunately, no BSP integer sorting algorithm is known to achieve optimality for all values of q . Along the lines of [205], we define the weaker operation of *semi-sorting* as that of reordering the items so that any subset of identically valued items is adjacent in the semi-sorted output (but these subsets need not be ordered).

Proposition 3.2.6 *For any constant $\rho > 0$, there exists a p -processor randomized BSP algorithm for semi-sorting an integer multi-set of size n in the range $[0, 1, \dots, R-1]$, R prime, that for all integers n and p , requires time*

$$C_{\text{semi_srt}}(n, R, p) = C_{\text{rdx_srt}}(n, \min\{n^{2+\rho}, R\}, p)$$

and

$$M_{\text{semi_srt}}(n, R, p) = C_{\text{rdx_srt}}(n, \min\{n^{2+\rho}, R\}, p)$$

for computation and communication respectively. The overall time complexity of the algorithm is denoted by $T_{\text{semi_srt}}(n, R, p) = C_{\text{semi_srt}}(n, R, p) + M_{\text{semi_srt}}(n, R, p)$.

Proof Let R be prime, $X \subseteq U = \{0, 1, \dots, R-1\}$, $|X| = n$, and consider function h to be chosen at random from $H_{n^{2+\rho}}^1$. Then, by way of lemma A.2.7,

$$\Pr[h \text{ is one-perfect for } X] = 1 - O(n^{-\rho}).$$

It is therefore straightforward to verify that h maps distinct items of X to distinct integer values in $[0, 1, \dots, n^{2+\rho} - 1]$, with probability $1 - O(n^{-\rho})$. These values can be subsequently sorted in time $T_{\text{rdx_srt}}(n, n^{2+\rho}, p)$ by way of proposition 2.4.1. We note that the case $R \leq n^{2+\rho}$ requires no randomization since integer sorting can be directly applied to the input. $\square$

Proof (of theorem 3.2.5) The theorem follows easily from proposition 3.2.6 by substituting for $T_{\text{rdx_srt}}$ the corresponding expression in proposition 2.4.1, and letting $t = 2$ in this expression. $\square$

3.3 Priority Queues and Dictionaries Revisited

In this section we design, analyze and implement simple BSP algorithms for two fundamental dynamic data structures, namely the *priority queue* and the *dictionary*. In contrast to the elaborate parallel priority queue (PPQ) and parallel dictionary (PD) algorithms of preceding sections (refer to the discussion of sections 2.9.6, 3.1.5, 3.2.5 and 3.2.7), the algorithms of this section are remarkably simple, yet, highly efficient both theoretically and practically. In particular, the algorithms are based on sequential data structures and a few primitive parallel operations, such as broadcasting, parallel-prefix and selection. The material of this section is based on the following papers:

- (1) "Concurrent heaps on the BSP model." Technical Report PRG-TR-14-96, Computing Laboratory, Oxford University, June 1996 [78].
- (2) "Selection on the bulk-synchronous parallel model with applications to priority queues." In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, August 1996 [79].
- (3) "Dynamic data structures: The BSP approach." Technical Report PRG-TR-11-97, Computing Laboratory, Oxford University, March 1997 [85].

- (4) “Parallel priority queues and list contraction: The BSP approach.” In *Proceedings of Euro-Par’97 Parallel Processing, Lecture Notes in Computer Science*, Springer-Verlag, August 1997 [88].

Remark For a detailed account of the notation, cost model and related work pertaining to this section refer to sections 3.1 and 3.2 respectively.

3.3.1 Parallel Priority Queues

In this section we study the computation and communication requirements of a recently introduced data structure, namely the *parallel priority queue* (PPQ) [171], and formulate an implementation on the BSP model. As noted in [171], PPQs allow the elegant high-level design and analysis of several parallel techniques, such as *branch-and-bound* [124, 125, 127], which require the solution of several sub-problems, each associated with a possibly different cost.

3.3.1.1 Preliminaries and Notation

A PPQ is an abstract data structure for maintaining a collection of (possibly duplicate) items and selecting the n items associated with the smallest values. We assume that the item values belong to a universe U equipped with a total order $<$ that can be evaluated in constant time by one processor for any given pair of arguments.

3.3.2 Optimal Randomized Inserting, Deleting-Min and Finding-Min Algorithms on Parallel Priority Queues

The following operations are defined on a PPQ Q .

- (1) $\text{INSERTQ}(\{y_1, y_2, \dots, y_n\}, Q)$ for the insertion of $\{y_1, y_2, \dots, y_n\}$ into Q .
- (2) $\text{DELETEMINQ}(n, Q)$ for the deletion of the n smallest items from Q .
- (3) $\text{FINDMINQ}(n, Q)$ for the determination of the n smallest items of Q .

The implicit assumption is that there exists a $\text{CONSTRUCTQ}(Q)$ operation for the construction of an empty queue Q . In contrast to previous approaches [27, 171], our implementation has the advantage of supporting PPQ operations on data sets of varying size – in [27, 171], insert, delete-min and find-min operations have to be performed on data sets of predetermined size.

The randomized algorithms we present in this section for supporting the pre-mentioned PPQ operations are based on an efficient processor-mapping of sequential ordinary (and leftist) heaps, even though, other sequential data structures might be supported as well (e.g., relaxed heaps, fibonacci heaps, balanced search trees [54, 133, 202]). To this end, in the forthcoming analysis of the PPQ algorithms we do not make any assumptions about the underlying local (sequential) priority queues. In order to simplify the exposition, the following notational convention is employed.

- $\tau_{IQ}(n, m)$ – denotes the time required to sequentially insert n items into a local priority queue of size m .

- $\tau_{FQ}(n, m)$ – denotes the time required to sequentially determine the n smallest items of a local priority queue of size m .
- $\tau_{DQ}(n, m)$ – denotes the time required to sequentially delete the n smallest items from a local priority queue of size m .
- $\tau_{CQ}(m)$ – denotes the time required to sequentially construct a local priority queue of size m (we note that in general $\tau_{CQ}(m) \leq \sum_{i=0}^{m-1} \tau_{IQ}(1, i)$).
- $\tau_{MQ}(m_1, m_2)$ – denotes the time required to sequentially meld two local priority queues of size m_1 and m_2 into one of size $m_1 + m_2$.

In [125, 127], a parallel randomized technique to approximate a priority queue is presented. This technique is extended in [179] to handle exact priority queues. In particular, in [179] a randomized variant of the approximate PPQs of [127] is presented that delivers optimal performance for the insert operation but sub-optimal performance for the find-min and delete-min operations. The main inefficiency of the delete-min and find-min operations of [179] derives from a global sorting that is inherent in the particular implementation of these operations. Parallel *sorting* – in the setting of PPQs – has two major drawbacks: (i) it is communication intensive, and (ii) it has a super-linear computation and communication time complexity. In this work we propose parallel *selection*, in order to alleviate the limitations imposed by sorting.

The randomized methods we present in this section derive from the ideas of [125, 127] as extended in [179]. The underlying algorithms are fairly simple and achieve optimality for a wide range of values of n , p , L and g .

Informal Description of Algorithm The optimal randomized insert, delete-min and find-min algorithms (on parallel priority queues) are based on randomized load-balancing techniques and are non-iterative in nature. The input is assumed to be evenly but otherwise arbitrarily divided among the p participating processors before the beginning of the execution of the algorithm. Processors access independent partitions of the parallel priority queue (which is randomly distributed amongst the p -participating processors) simultaneously. In the insert algorithm, processors randomly distribute the new items amongst the p participating processors and each processor inserts, by means of some optimal sequential insertion algorithm, the items it received in the distribution phase to its local priority queue (ordinary heap). In the delete-min algorithm (and similarly in the find-min algorithm), each processor selects, by means of some optimal sequential find-min algorithm, an appropriate number of minimum items from its local priority queue (ordinary heap) and from the union of these sets the processors extract, by means of the selection algorithm of section 2.8.5, the required number of minimum items.

Our main result is summarized in the following theorem – additional results are presented later in this section.

Theorem 3.3.1 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete-min and find-min on a PPQ of size m , that for all integers n , m and p such that $m \geq n$ and $p = o(n/\lg n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$\begin{aligned} (c_{ins} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} & \text{ for insert,} \\ (c_{del} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} & \text{ for delete-min/find-min,} \end{aligned}$$

and

$$\begin{aligned} L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) & \text{ for insert,} \\ L \cdot O(\lg p) + g \cdot o\left(\frac{n}{p}\right) & \text{ for delete-min/find-min,} \end{aligned}$$

for computation and communication respectively.

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 3.3.1 follows easily as a consequence of this result.

Proposition 3.3.1 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete-min and find-min on a PPQ of size m , that for all integers n and p such that $p = O(n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$C_{ppq}(n, m, p) = \begin{cases} \tau_{IQ}(S(n, n), S(n, m)) & \text{for insert,} \\ \tau_{FQ}\left(2 \cdot \left\lceil \frac{n}{p} \right\rceil, S(n, m)\right) + \tau_{DQ}(1, S(n, m)) \cdot (S(n, 1.2 \cdot n) + S(n, n)) + \\ 2 \cdot \tau_{FQ}(1, S(n, m)) + C_{sel}\left(2 \cdot p \cdot \left\lceil \frac{n}{p} \right\rceil, n, p\right) + \\ C_{sel}(p \cdot S(n, 1.2 \cdot n), n, p) + O(1) & \text{for delete-min/find-min,} \end{cases}$$

and

$$M_{ppq}(n, m, p) = \begin{cases} L \cdot O(1) + g \cdot O(S(n, n)) & \text{for insert,} \\ M_{sel}\left(2 \cdot p \cdot \left\lceil \frac{n}{p} \right\rceil, n, p\right) + M_{sel}(p \cdot S(n, 1.2 \cdot n), n, p) & \text{for delete-min/find-min,} \end{cases}$$

for computation and communication respectively, where

$$S(n, k) = \begin{cases} O\left(\min\left\{\lg n, \frac{k}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{k}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{k}}\right)\right) \cdot \frac{k}{p} & \text{otherwise.} \end{cases}$$

The overall time complexity of the algorithm is denoted by $T_{ppq}(n, m, p) = C_{ppq}(n, m, p) + M_{ppq}(n, m, p)$.

Proof The PPQ is maintained as a collection of disjoint local (sequential) priority queues. For the insertion of n new items (at most $\lceil n/p \rceil$ per processor), each processor sends its (at most) $\lceil n/p \rceil$ items to uniformly at random chosen processors, where they are inserted into their local priority queues. The procedure to manage the insert operation on a PPQ Q is INSERTQ($\{y_1, y_2, \dots, y_n\}, Q$) and is outlined in figure 3.9.¹³

The correctness of procedure INSERTQ is immediate; it therefore remains to bound the resources required by the aforementioned scheme. The following lemma bounds the number of items received by any processor as a result of the random distribution of items to processors (line 2).

¹³With reference to the pseudo-code of this section, the superscript notation $\langle k \rangle$ refers to processor p_k (or equivalently, processor $p^{(k)}$) and implies the “local segment” of that processor.

begin INSERTQ ($\{y_1, y_2, \dots, y_n\}, Q$)
 1. denote by Y set $\{y_1, y_2, \dots, y_n\}$;
 2. *distribute* randomly the items of Y among the p processors ;
 3. **for** each processor $p^{(k)}$, $0 \leq k < p$, in parallel
 4. **do** SEQ_INSERTQ($Y^{(k)}, Q^{(k)}$) ;
end INSERTQ

Figure 3.9: Procedure INSERTQ, p - number of processors.

Claim 3.3.1 *Let in procedure INSERTQ $Y = \bigcup_k Y^{(k)}$ and $Q = \bigcup_k Q^{(k)}$, for all $0 \leq k < p$. Then, for any constant $\rho > 0$ and all $0 \leq k < p$,*

$$|Y^{(k)}| = \begin{cases} O\left(\min\left\{\lg n, \frac{n}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{n}}\right)\right) \cdot \frac{n}{p} & \text{otherwise,} \end{cases}$$

and

$$|Q^{(k)}| = \begin{cases} O\left(\min\left\{\lg n, \frac{m}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{m}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{m}}\right)\right) \cdot \frac{m}{p} & \text{otherwise,} \end{cases}$$

with probability $1 - O(n^{-\rho})$, where m denotes the total number of items in Q prior to the insertion.

Proof In procedure INSERTQ, the random distribution of items to processors implies that the probability of a processor receiving an item is $1/p$. Therefore, the expected number of items received per processor is n/p . By invoking theorem A.1.2, for $n/p = 1$, it follows that, for all $0 \leq k < p$,

$$\Pr\left[|Y^{(k)}| = O\left(\frac{\lg n}{\lg \lg n}\right)\right] = 1 - O(n^{-\rho}).$$

Similarly, by invoking theorem A.1.3, for $n/p = \omega(\lg n)$, it follows that, for all $0 \leq k < p$,

$$\Pr\left[|Y^{(k)}| = \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{n}}\right)\right) \cdot \frac{n}{p}\right] = 1 - O(n^{-\rho}).$$

The analysis of the remaining cases follows a line of arguments similar to that of the aforementioned cases and is therefore omitted. The bound on $|Q^{(k)}|$ follows similarly. $\square$

The resources required for procedure INSERTQ are bounded as follows. The random distribution of items to processors (line 2) takes communication time $L + g \cdot O(|Y^{(k)}|)$, with probability $1 - O(n^{-\rho})$. The insertion of the $|Y^{(k)}|$ items in the local priority queues (line 4) requires time $\tau_{IQ}(|Y^{(k)}|, |Q^{(k)}|)$. Thus, insertion can be efficiently implemented by employing this simple randomized scheme in time

$$\tau_{IQ}\left(|Y^{(k)}|, |Q^{(k)}|\right)$$

and

$$L + g \cdot O(|Y^{(k)}|),$$

for computation and communication respectively.

For the deletion of the n smallest items, the following (rather more complicated) procedure is employed. The $2 \cdot \lceil n/p \rceil$ smallest items in each processor are duplicated. Let X denote the set of these $2 \cdot p \cdot \lceil n/p \rceil$ items. By employing selection, the n -th smallest item e of X is determined and broadcast. Let $R^{(k)}$ denote the set of items in processor $p^{(k)}$, $0 \leq k < p$, that are not larger than e . By employing selection, the n -th smallest item e' from $\bigcup_k R^{(k)}$ is determined and broadcast to all processors. Finally, the set S of the n smallest items (the items smaller than e') is removed from the local priority queues and returned as the result of the delete-min operation. The procedure to manage the deletion operation on a PPQ Q is DELETETEMINQ(n, Q) and is outlined in figure 3.10. The find-min operation is implemented similarly.

```

begin DELETETEMINQ ( $n, Q$ )
1.  if  $|Q| = n$  then
2.    return  $Q^{(k)}$  ;
3.  else
4.    for each processor  $p^{(k)}$ ,  $0 \leq k < p$ , in parallel
5.      do  $X^{(k)} = \{\text{SEQ\_FINDMINQ}(2 \cdot \lceil n/p \rceil, Q^{(k)})\}$  ;
6.      let  $R^{(k)} = \{\}$  ;
7.      let  $e = \text{SELECT}(\bigcup_k X^{(k)}, n)$  ;
8.      while  $\text{SEQ\_FINDMINQ}(1, Q^{(k)}) \leq e$ 
9.        let  $x = \text{SEQ\_DELETETEMINQ}(1, Q^{(k)})$  ;
10.       let  $R^{(k)} = R^{(k)} \cup \{x\}$  ;
11.      let  $S^{(k)} = \{\}$  ;
12.      let  $e' = \text{SELECT}(\bigcup_k R^{(k)}, n)$  ;
13.      while  $\text{SEQ\_FINDMINQ}(1, Q^{(k)}) \leq e'$ 
14.        let  $x = \text{SEQ\_DELETETEMINQ}(1, Q^{(k)})$  ;
15.        let  $S^{(k)} = S^{(k)} \cup \{x\}$  ;
16.      return  $S^{(k)}$  ;
end DELETETEMINQ

```

Figure 3.10: Procedure DELETETEMINQ, p - number of processors.

The correctness of procedure DELETETEMINQ is easy to verify; it therefore remains to bound the resources required by the aforementioned scheme. The following claim bounds the total number of items (as well as the number of items per processor) that are smaller than e (lines 6-10).

Claim 3.3.2 *Let in procedure DELETETEMINQ $R = \bigcup_k R^{(k)}$, for all $0 \leq k < p$. Then, for some constant $\rho' > 0$, we have that $|R| < \lceil 1.2 \cdot n \rceil$, with probability at least $1 - O(2^{-\rho' \cdot n})$. Additionally, for any constant $\rho > 0$ and all $0 \leq k < p$,*

$$|R^{(k)}| = \begin{cases} O\left(\min\left\{\lg n, \frac{1.2 \cdot n}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{1.2 \cdot n}}\right)\right) \cdot \frac{1.2 \cdot n}{p} & \text{otherwise,} \end{cases}$$

with probability $1 - O(n^{-\rho})$.

Proof We prove the bound on R for the case $n = p$; the other cases follow trivially. Let $R_{1.2n}$ denote the smallest $\lceil 1.2 \cdot n \rceil$ items (overall) at the beginning of procedure DELETEMINQ. Along the lines of [179], we establish that with probability $1 - O(2^{-\rho' \cdot n})$, set $X = \bigcup_k X^{(k)}$, for all $0 \leq k < p$, of procedure DELETEMINQ contains at least n items from $R_{1.2n}$. Therefore, since $R \subseteq R_{1.2n}$ we have that $|R| \leq 1.2 \cdot n$, with the stated probability.

Let Z be the random variable denoting the number of processors containing no items when $\lceil 1.2 \cdot n \rceil$ items are placed randomly into n processors. Similarly, let W be the random variable denoting the number of processors containing exactly one item when $\lceil 1.2 \cdot n \rceil$ items are placed randomly into n processors. Substituting in lemmas A.1.1 and A.1.2, $m = \lceil 1.2 \cdot n \rceil$, $k = n$, and $\lambda = c \cdot n$ (for any positive constant c), we obtain that

$$\Pr[|Z - \mu_0| < c \cdot n \text{ and } |W - \mu_1| < c \cdot n] = 1 - O(2^{-\rho' \cdot n}),$$

where $\mu_0 = (1 - (1/n))^{\lceil 1.2 \cdot n \rceil} \cdot n$ and $\mu_1 = (1 - (1/n))^{\lceil 1.2 \cdot n \rceil - 1} \cdot \lceil 1.2 \cdot n \rceil$. By definition, X contains the two smallest items from each processor at the beginning of procedure DELETEMINQ. Thus, X contains at least

$$W + 2 \cdot (n - Z - W) \geq 2 \cdot n - 2 \cdot (\mu_0 + c \cdot n) - (\mu_1 + c \cdot n) \geq n,$$

items from $R_{1.2n}$, for some appropriately small constant c . Therefore, we obtain that $R \subseteq R_{1.2n}$, or equivalently, that $|R| < 1.2 \cdot n$, with probability $1 - O(2^{-\rho' \cdot n})$. The analysis pertaining to the size of $R^{(k)}$ follows a line of arguments similar to that of claim 3.3.1 and is omitted. $\square$

The resources required for procedure DELETEMINQ are bounded as follows. The determination of the $2 \cdot \lceil n/p \rceil$ smallest items per local priority queue (line 5) requires time $\tau_{FQ}(2 \cdot \lceil n/p \rceil, |Q^{(k)}|)$. The median selection process (line 7) takes time $T_{sel}(2 \cdot p \cdot \lceil n/p \rceil, n, p)$. The sequential find-min and delete-min operations per local priority queue (lines 8-10) involve $|R^{(k)}|$ items, and therefore, require time $\tau_{FQ}(1, |Q^{(k)}|) + \tau_{DQ}(1, |Q^{(k)}|) \cdot |R^{(k)}|$. Accordingly, the selection process (line 12) requires time at most $T_{sel}(p \cdot |R^{(k)}|, n, p)$. By employing probabilistic arguments similar to those of claim 3.3.1 we can further establish that the sequential find-min and delete-min operations per local priority queue (lines 13-15) involve $|Y^{(k)}|$ items, with probability $1 - O(n^{-\rho})$, for any constant $\rho > 0$, and thus, require time $\tau_{FQ}(1, |Q^{(k)}|) + \tau_{DQ}(1, |Q^{(k)}|) \cdot |Y^{(k)}|$. Finally, the local operations (lines 6, 11, 16) take time $O(1)$. Thus, deletion can be implemented efficiently by employing this simple randomized scheme in time

$$\begin{aligned} &\tau_{FQ}\left(2 \cdot \left\lceil \frac{n}{p} \right\rceil, |Q^{(k)}|\right) + 2 \cdot \tau_{FQ}(1, |Q^{(k)}|) + \tau_{DQ}(1, |Q^{(k)}|) \cdot (|R^{(k)}| + |Y^{(k)}|) + \\ &C_{sel}\left(2 \cdot p \cdot \left\lceil \frac{n}{p} \right\rceil, n, p\right) + C_{sel}(p \cdot |R^{(k)}|, n, p) + O(1) \end{aligned}$$

and

$$M_{sel}\left(2 \cdot p \cdot \left\lceil \frac{n}{p} \right\rceil, n, p\right) + M_{sel}(p \cdot |R^{(k)}|, n, p),$$

for computation and communication respectively.

Analysis The correctness of the algorithm follows from its description. By summing up the computation and communication requirements of procedures INSERTQ and DELETEMINQ we obtain the desired time bound. The probability bound follows by way of claims 3.3.1 and 3.3.2. We also observe that the aforementioned procedures perform $O(1)$ selection operations, and by way of proposition 2.8.3, the failure probability of each such selection operation is $O(n^{-\rho})$.

The proposition follows. $\square$

Proof (of theorem 3.3.1) The theorem follows easily from proposition 3.3.1 by employing ordinary heaps [54, 202] – as the underlying sequential PQ – and substituting for T_{sel} , T_{bcast} and T_{ppf} the corresponding expressions in propositions 2.8.3, 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

We note that we can improve the performance of the INSERTQ and DELETEMINQ operations in several ways. For example, in case $n/p = \Omega(\lg n)$, a simplified version of procedure DELETEMINQ can be employed to realize the delete-min operation. The procedure to manage the simplified deletion operation on a PPQ Q is DELETEMINQ $^*(n, Q)$ and is outlined in figure 3.11. The find-min operation is implemented similarly. To simplify the exposition, in the remainder of this section we assume that $n/p = \Omega(\lg n)$.

```

begin DELETEMINQ $^*$  ( $n, Q$ )
1.  if  $|Q| = n$  then
2.      return  $Q^{(k)}$  ;
3.  else
4.      for each processor  $p^{(k)}$ ,  $0 \leq k < p$ , in parallel
5.          do  $X^{(k)} = \{\text{SEQ\_FINDMINQ}(\lceil (1 + \varepsilon)n/p \rceil, Q^{(k)})\}$  ;
6.          let  $R^{(k)} = \{\}$  ;
7.          let  $e = \text{SELECT}(\bigcup_k X^{(k)}, n)$  ;
8.          while  $\text{SEQ\_FINDMINQ}(1, Q^{(k)}) \leq e$ 
9.              let  $x = \text{SEQ\_DELETEMINQ}(1, Q^{(k)})$  ;
10.             let  $R^{(k)} = R^{(k)} \cup \{x\}$  ;
11.         return  $R^{(k)}$  ;
end DELETEMINQ $^*$ 

```

Figure 3.11: Procedure DELETEMINQ * , p - number of processors, ε - $0 < \varepsilon < 1$.

In theorem 3.3.1, the underlying local (sequential) PQ maintained by each processor is assumed to be an ordinary sequential heap, and therefore, the insertion of $O(n/p)$ items per processor requires time $(c_{ins} + o(1)) \cdot (n/p) \cdot \lg(m/p)$. If instead of using ordinary heaps, we employ leftist heaps [202], then the INSERTQ operation can be performed in $O(n/p + \lg(m/p))$ time as follows. Each processor constructs an auxiliary leftist heap on the $(1 + o(1)) \cdot (n/p)$ items it receives, as a result of the random distribution of items to processors (line 2), in time $\tau_{CQ}((1 + o(1)) \cdot (n/p)) = O(n/p)$ [202], and subsequently melds this auxiliary leftist heap with its local heap that is part of Q . The latter operation requires time $\tau_{MQ}((1 + o(1)) \cdot (n/p), (1 + o(1)) \cdot (m/p)) = O(\lg(n/p) + \lg(m/p))$ [202]. Overall, an improved $O(n/p + \lg(m/p))$ computation time bound can be claimed for insertion.

The usage of leftist heaps can also be exploited to obtain $O(n/p)$ computation time for the find-min operation. The general idea is as follows (recall that the find-min operation is realized along the lines of the delete-min operation). Each processor maintains a copy of the $2 \cdot \lceil n/p \rceil$ smallest items of its local leftist heap in a separate list. We observe that the $2 \cdot \lceil n/p \rceil$ smallest items can be maintained in a separate

local list in time $O(n/p)$ under the operations insert and delete-min by employing a linear time sequential selection algorithm [54, 70, 133]. As it is noted in [202], the heap structure of leftist heaps allows the determination of all items not exceeding a pre-specified value $\bar{e}$ in time proportional to the number of such items, by performing a pre-order traversal of the underlying leftist heap starting from the root, listing every encountered item with value not exceeding $\bar{e}$ and immediately retreating from each item with value exceeding $\bar{e}$. Therefore, the determination of items with values not exceeding e and e' (lines 8-10, 13-15 of procedure DELETEMINQ) requires time $O(n/p)$. The $O(n/p)$ time bound on the find-min operation follows. The following analogue of theorem 3.3.1 can then be derived.

Theorem 3.3.2 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete-min and find-min on a PPQ of size m , that for all integers n , m and p such that $m \geq n$ and $p = o(n/\lg n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$\begin{array}{ll} O\left(\frac{n}{p} + \lg \frac{m}{p}\right) & \text{for insert,} \\ (c_{del} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} & \text{for delete-min,} \\ O\left(\frac{n}{p}\right) & \text{for find-min,} \end{array}$$

and

$$\begin{array}{ll} L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) & \text{for insert,} \\ L \cdot O(\lg p) + g \cdot o\left(\frac{n}{p}\right) & \text{for delete-min/find-min,} \end{array}$$

for computation and communication respectively.

Proof The proof follows a line of arguments similar to that of theorem 3.3.1, where ordinary heaps are replaced by leftist heaps [202]. $\square$

3.3.2.1 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the PPQ algorithms (INSERTQ, DELETEMINQ and FINDMINQ) of the previous sections.

Implementation The prototypes adopted for the *BSPlib* priority queue procedures are outlined in figure 3.12.

```
void BSP_insert_ppq (byte *src, ppq **trg, int n, int *m, int sb, int (*cmp)()) ;
void BSP_delete_min_ppq (byte **src, ppq **trg, int n, int *m, int sb, int (*cmp)()) ;
void BSP_find_min_ppq (byte **src, ppq *trg, int n, int m, int sb, int (*cmp)()) ;
```

src – input/output array.
trg – priority queue data structure.
n – input size.
m – priority queue size.
sb – item size.
cmp – generic compare function.

Figure 3.12: Procedure for BSP PPQ inserting, deleting-min and finding-min.

The aim of functions (procedures) `BSP_insert_ppq`, `BSP_delete_min_ppq` and `BSP_find_min_ppq` is to provide a data-type independent high-performance *BSPlib* priority queue implementation.¹⁴ In the following section we study the experimental performance of the parallel priority queue insertion and deletion algorithms, using as underlying sequential data structures ordinary binary heaps.

Experimental Results The theoretical performance of the algorithms is summarized in theorems 3.3.1. To determine the practical significance of the aforementioned results we benchmarked our implementations (`BSP_insert_ppq` and `BSP_delete_min_ppq`) on two different machines with a varying number of processors and input sets (consisting of uniformly distributed integer valued items – unless otherwise indicated).¹⁵

Figures 3.13, 3.15, 3.17, 3.19, 3.21, 3.23, 3.25 and 3.27 present the execution time of procedures `BSP_insert_ppq` and `BSP_delete_min_ppq` on the two machines. Similarly, figures 3.14, 3.16, 3.18, 3.20, 3.22, 3.24, 3.26 and 3.28 expose the efficiency of the corresponding timing results.

As shown in figures 3.13, 3.15, 3.17, 3.19, 3.21, 3.23, 3.25 and 3.27, the execution time of procedures `BSP_insert_ppq` and `BSP_delete_min_ppq` using a fixed number of processors scale with the input size (approximately as $n \cdot \lg n$). Accordingly, the execution time of procedures `BSP_insert_ppq` and `BSP_delete_min_ppq` for a fixed input size scale inversely with the number of processors (approximately as $1/p$). Both experimental results are in accordance with the theoretical analysis of section 3.3.2. In terms of efficiency, procedure `BSP_insert_ppq` utilizes approximately (within a 5%):

- 50%–85% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 85%–100% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),
- 15%–85% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 30%–85% of the available processor resources on the Cray T3D machine (for large problem sizes).

Similarly, in terms of efficiency, procedure `BSP_delete_min_ppq` utilizes approximately (within a 5%):

- 40%–50% of the available processor resources on the SGI Power Challenge machine (for small problem sizes),
- 35%–45% of the available processor resources on the SGI Power Challenge machine (for large problem sizes),
- 15%–50% of the available processor resources on the Cray T3D machine (for small problem sizes), and
- 20%–85% of the available processor resources on the Cray T3D machine (for large problem sizes).

¹⁴The terms generic and data-type independent imply that procedures `BSP_insert_ppq`, `BSP_delete_min_ppq` and `BSP_find_min_ppq` can be employed to handle *any* data type. To this end, a comparison function `cmp` is used to facilitate the comparison of two elements e_1 and e_2 . In particular, the `cmp` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2 .

¹⁵The find-min implementation (`BSP_find_min_ppq`) is similar to the delete-min implementation (`BSP_delete_min_ppq`), and therefore, has been omitted from our experimental study.

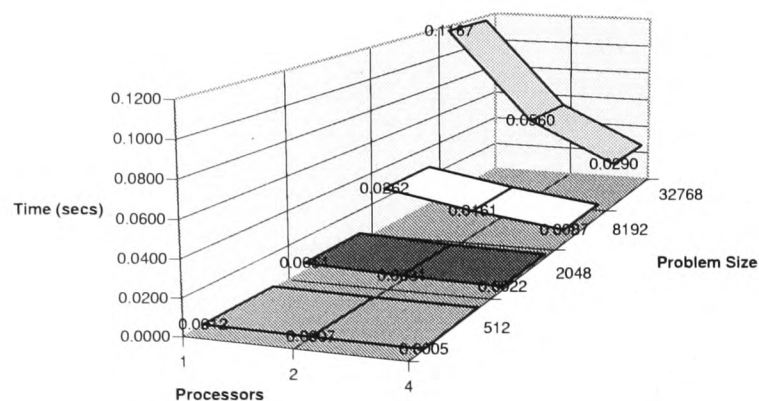


Figure 3.13: Execution time on a SGI Power Challenge for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{17} .

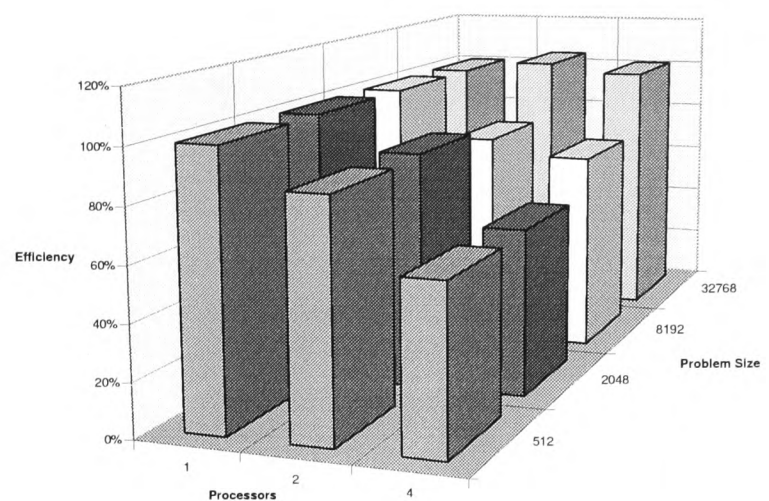


Figure 3.14: Efficiency on a SGI Power Challenge for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{17} .

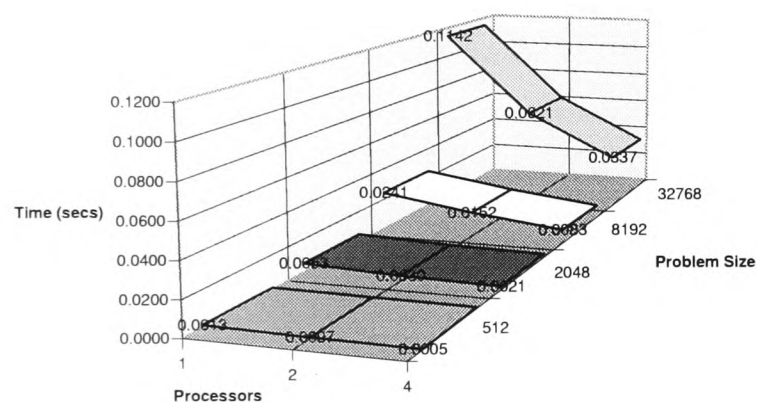


Figure 3.15: Execution time on a SGI Power Challenge for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{19} .

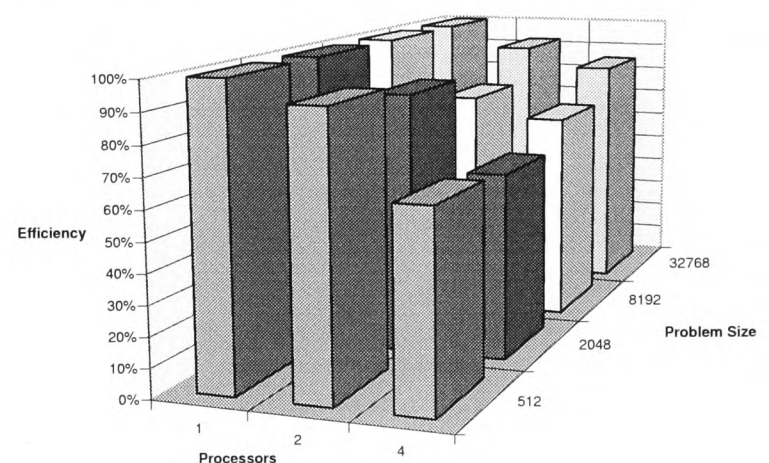


Figure 3.16: Efficiency on a SGI Power Challenge for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{19} .

These results should be interpreted with caution on a twofold basis: (i) the random choices of the parallel and sequential algorithm might be different, and therefore, small inconsistencies in timing results are possible, and (ii) the timing results are influenced by caching effects, particularly for the SGI machine through the presence of first- and second-level cache on operations involving a relatively small number of items.

3.3.3 Parallel Dictionaries

In this section we study the computation and communication requirements of a recently introduced data structure, namely the *parallel dictionary* (PD) [170], and formulate an implementation on the BSP model. As noted in [57], PDs allow the elegant high-level design and analysis of several parallel techniques, using data structures similar to those of sequential models.

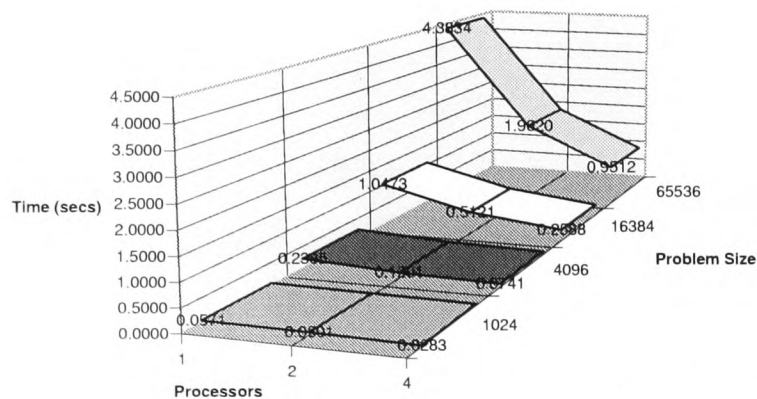


Figure 3.17: Execution time on a SGI Power Challenge for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{17} .

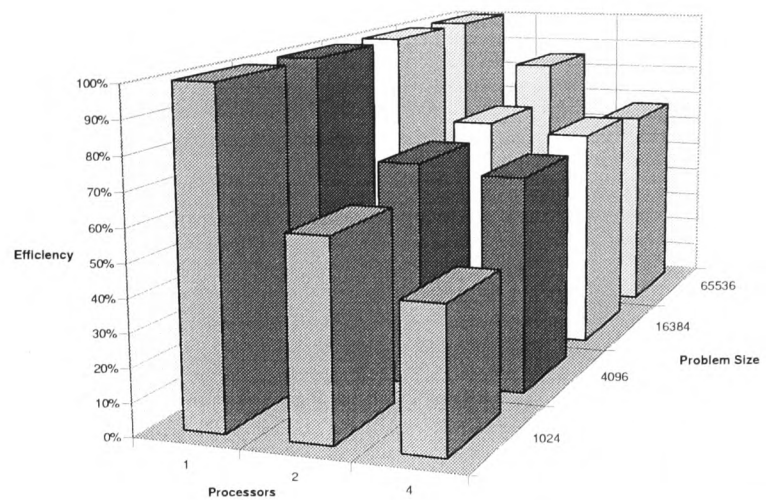


Figure 3.18: Efficiency on a SGI Power Challenge for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{17} .

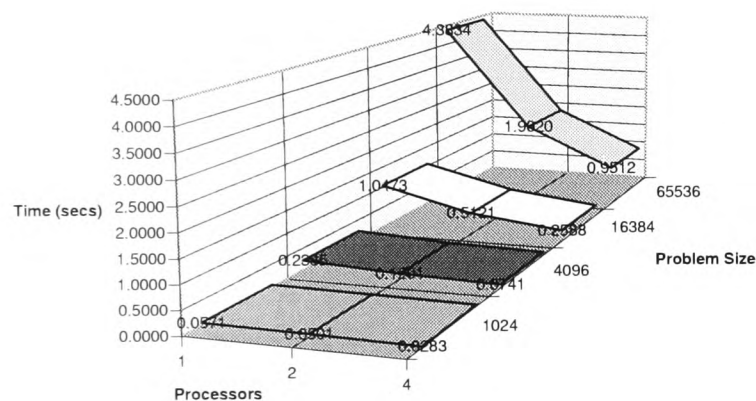


Figure 3.19: Execution time on a SGI Power Challenge for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{19} .

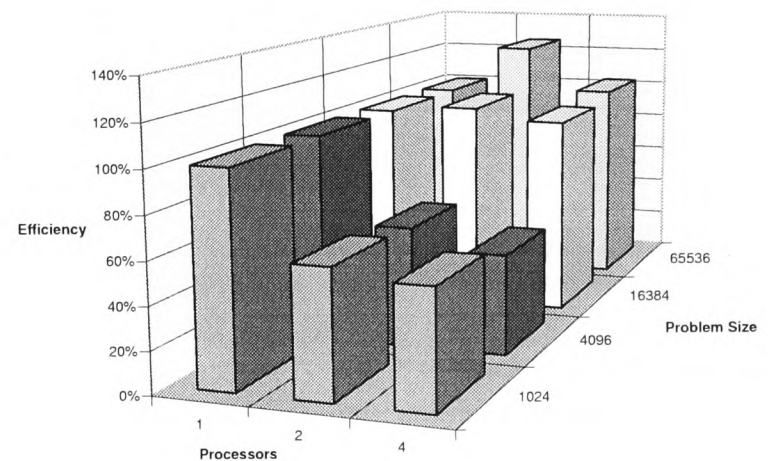


Figure 3.20: Efficiency on a SGI Power Challenge for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{19} .

3.3.3.1 Preliminaries and Notation

A PD is an abstract data structure for maintaining a collection of (unique) items¹⁶ and supporting efficient insert, delete, and find operations on data sets of size n . We assume that the item values belong to a finite universe $U = \{0, 1, \dots, q-1\}$, q prime.

3.3.4 Optimal Randomized Inserting, Deleting and Finding Algorithms on Parallel Dictionaries

The following operations are defined on a PD D .

- (1) INSERTD($\{y_1, y_2, \dots, y_n\}, D$) for the insertion of $\{y_1, y_2, \dots, y_n\}$ into D .
- (2) DELETED($\{y_1, y_2, \dots, y_n\}, D$) for the deletion of $\{y_1, y_2, \dots, y_n\}$ from D .

¹⁶The explicit assumption in the forthcoming analysis of the PD operations is that items are unique. In case of duplicate items, the upper bounds presented in the following sections do not apply. To remedy the situation, in section 3.2.8 we have outlined an efficient duplicate elimination scheme that removes the requirement for unique items.

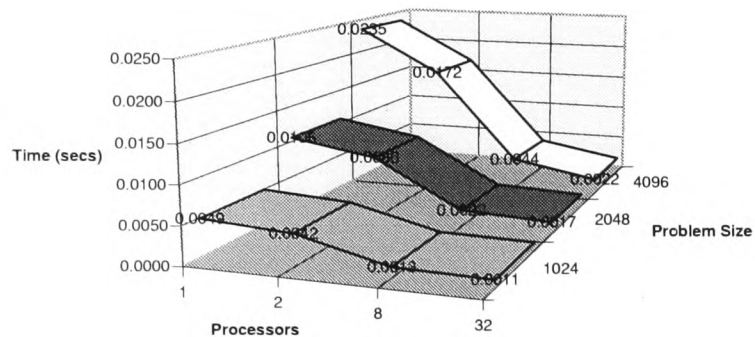


Figure 3.21: Execution time on a Cray T3D for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{16} .

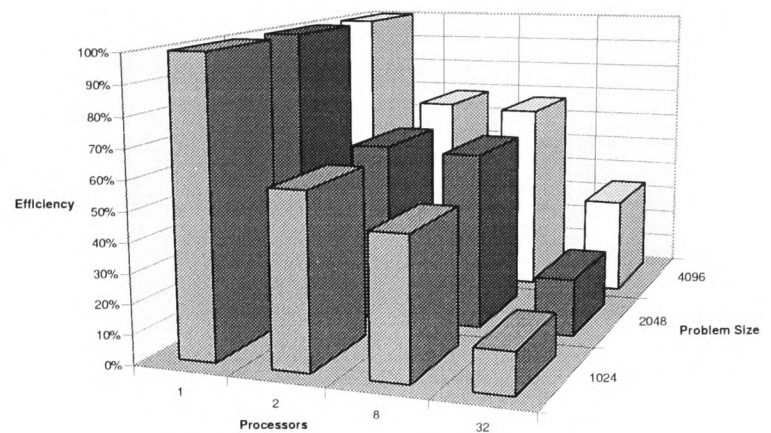


Figure 3.22: Efficiency on a Cray T3D for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{16} .

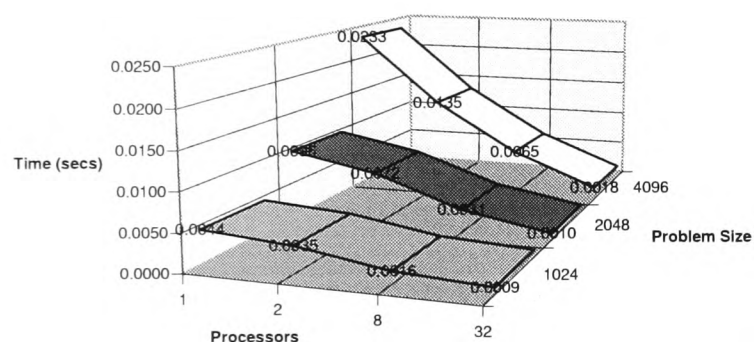


Figure 3.23: Execution time on a Cray T3D for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{18} .

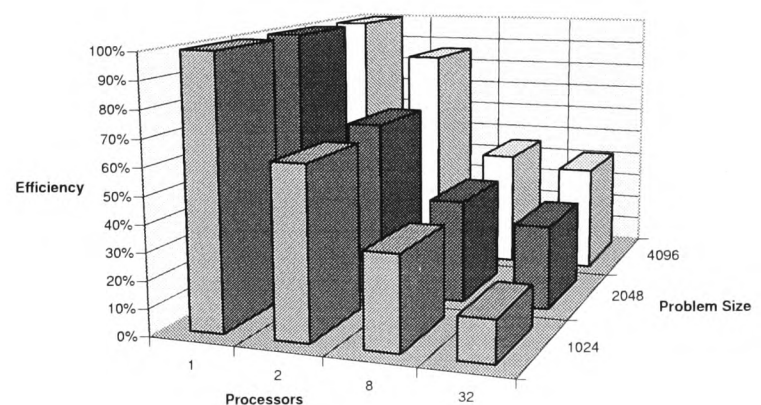


Figure 3.24: Efficiency on a Cray T3D for randomized heap insertion into an ordinary heap – uniform distribution (U), heap size 2^{18} .

- (3) $\text{FINDD}(\{y_1, y_2, \dots, y_n\}, D)$ for the determination of $\{e_1, e_2, \dots, e_n\}$, where $e_i = y_i$ if $y_i \in D$, and $e_i = \{\}$ otherwise, for all $1 \leq i < n$.

The implicit assumption is that there exists a $\text{CONSTRUCTD}(D)$ operation for the construction of an empty dictionary D . In contrast to previous approaches [26, 178], our implementation has the advantage of supporting PD operations on data sets of varying size – in [26, 178], insert, delete and find operations have to be performed on data sets of predetermined size.

The randomized algorithm we present in this work for supporting the pre-mentioned PD operations is based on an efficient processor-mapping of sequential balanced search trees, even though, other sequential data structures might be supported as well (e.g., hash tables, treaps, skip lists [54, 165, 202]). To this end, in the forthcoming analysis of the PD algorithm we do not make any assumptions about the underlying local (sequential) dictionary. In order to simplify the exposition, the following notational conventions are employed.

- $\tau_{ID}(n, m)$ – denotes the time required to sequentially insert n items into a local dictionary of size m .
- $\tau_{FD}(n, m)$ – denotes the time required to sequentially find n items in a local dictionary of size m .

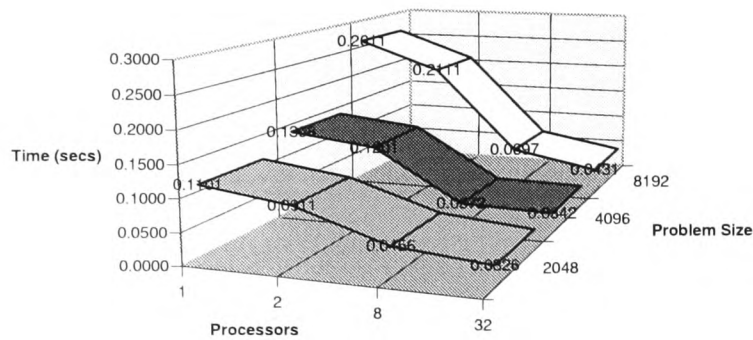


Figure 3.25: Execution time on a Cray T3D for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{16} .

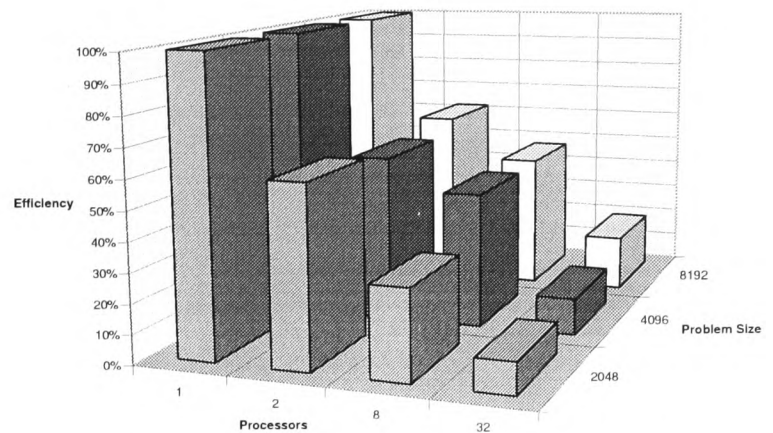


Figure 3.26: Efficiency on a Cray T3D for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{16} .

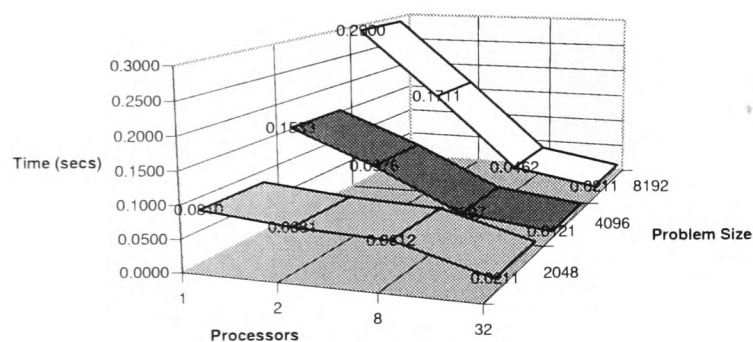


Figure 3.27: Execution time on a Cray T3D for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{18} .

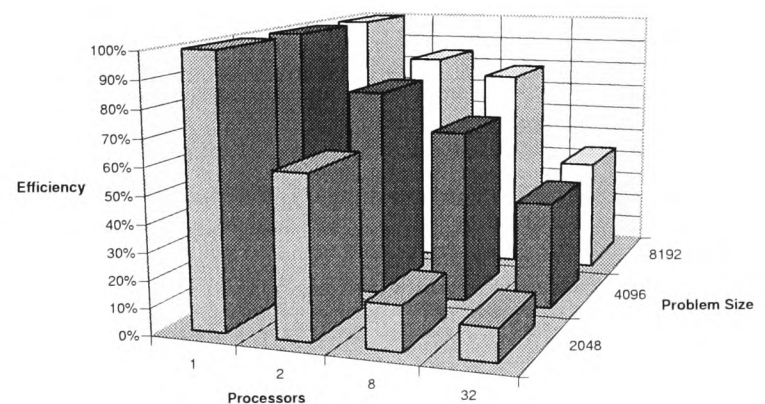


Figure 3.28: Efficiency on a Cray T3D for randomized heap deletion from an ordinary heap – uniform distribution (U), heap size 2^{18} .

m .

- $\tau_{DD}(n, m)$ – denotes the time required to sequentially delete n items from a local dictionary of size m .
- $\tau_{CD}(m)$ – denotes the time required to sequentially construct a dictionary of size m (we note that in general $\tau_{CD}(m) \leq \sum_{i=0}^{m-1} \tau_{ID}(1, i)$).
- $\tau_{JD}(m_1, m_2)$ – denotes the time required to sequentially join two dictionaries of size m_1 and m_2 into one of size $m_1 + m_2$.

The quasi-randomized methods we present in this section derive from the idea of quasi-random functions of section 3.2.6.3. The underlying algorithms are fairly simple and achieve optimality for a wide range of values of n , p , L and g .

Informal Description of Algorithm The optimal randomized insert, delete and search algorithms (on parallel dictionaries) are immediate adaptations of the parallel priority queue algorithms of section 3.3.1.

Our main result is summarized in the following theorem – additional results are presented later in this section.

Theorem 3.3.3 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete and find on a PD of size m , that for all integers n , m and p such that $m \geq n$ and $p = o(n/\lg n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$\begin{aligned} (c_{ins} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} & \quad \text{for insert,} \\ (c_{del} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} & \quad \text{for delete-min,} \\ (c_{srch} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} & \quad \text{for find-min,} \end{aligned}$$

and

$$L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) \quad \text{for insert/delete/find,}$$

for computation and communication respectively. The algorithm requires $O(\sqrt{n}/p + \lg p) + L \cdot O(\lg p) + g \cdot O(\lg p)$ preprocessing time.

The proof of the aforementioned result is diverged to the end of this section. The remainder of this section is devoted to the proof of the following proposition. Then, theorem 3.3.3 follows easily as a consequence of this result.

Proposition 3.3.2 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete and find on a PD of size m , that for all integers n and p such that $p = O(n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$C_{pd}(n, m, p) = \begin{cases} \tau_{ID}(S(n, n), S(n, m)) + C_{e_qrnd}(n, 1/2, 1, p) & \text{for insert,} \\ \tau_{DD}(S(n, n), S(n, m)) + C_{e_qrnd}(n, 1/2, 1, p) & \text{for delete,} \\ \tau_{FD}(S(n, n), S(n, m)) + C_{e_qrnd}(n, 1/2, 1, p) & \text{for find,} \end{cases}$$

and

$$M_{pd}(n, m, p) = \begin{cases} L \cdot O(1) + g \cdot O(S(n, n)) + M_{e_qrnd}(n, 1/2, 1, p) & \text{for insert/delete/find,} \end{cases}$$

for computation and communication respectively, where

$$S(n, k) = \begin{cases} O\left(\min\left\{\lg n, \frac{k}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{k}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{k}}\right)\right) \cdot \frac{k}{p} & \text{otherwise,} \end{cases}$$

The overall time complexity of the algorithm is denoted by $T_{pd}(n, m, p) = C_{pd}(n, m, p) + M_{pd}(n, m, p)$. The algorithm requires preprocessing time $T_{e_qrnd}(n, 1/2, p)$.

Proof The PD is maintained as a collection of disjoint local (sequential) dictionaries. For the insertion of n new items (at most $\lceil n/p \rceil$ per processor), each processor sends its (at most) $\lceil n/p \rceil$ items to uniformly at quasi-random chosen processors, where they are inserted in their local dictionaries. The procedure to manage the insert operation on a PD D is INSERTD($\{y_1, y_2, \dots, y_n\}, D$) and is outlined in figure 3.29.¹⁷

The correctness of procedure INSERTD is immediate; it therefore remains to bound the resources required by the aforementioned scheme. The following lemma bounds the number of items received by any processor as a result of the quasi-random distribution of items to processors (line 2).

¹⁷With reference to the pseudo-code of this section, the superscript notation $\langle k \rangle$ refers to processor p_k (or equivalently, processor $p^{(k)}$) and implies the “local segment” of that processor.

begin INSERTD ($\{y_1, y_2, \dots, y_n\}, D$)
 1. denote by Y set $\{y_1, y_2, \dots, y_n\}$;
 2. *distribute* quasi-randomly the items of Y among the p processors ;
 3. **for** each processor $p^{(k)}$, $0 \leq k < p$, in parallel
 4. **do** SEQ_INSERTD($Y^{(k)}, D^{(k)}$) ;
end INSERTD

Figure 3.29: Procedure INSERTD, p - number of processors.

Claim 3.3.3 *Let in procedure INSERTD $Y = \bigcup_k Y^{(k)}$ and $D = \bigcup_k D^{(k)}$, for all $0 \leq k < p$. Then, for any constant $\rho > 0$ and all $0 \leq k < p$,*

$$|Y^{(k)}| = \begin{cases} O\left(\min\left\{\lg n, \frac{n}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{n}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{n}}\right)\right) \cdot \frac{n}{p} & \text{otherwise,} \end{cases}$$

and

$$|D^{(k)}| = \begin{cases} O\left(\min\left\{\lg n, \frac{m}{p} \cdot \frac{\lg n}{\lg \lg n}\right\}\right) & \text{if } \frac{m}{p} = O(\lg n), \\ \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{m}}\right)\right) \cdot \frac{m}{p} & \text{otherwise,} \end{cases}$$

with probability $1 - O(n^{-\rho})$, where m denotes the total number of items in D prior to the insertion.

Proof In procedure INSERTD, the quasi-random distribution of items to processors implies that the results of sub-section 3.2.6.2 can be employed to bound the number of items per processor. Hence, by invoking corollary 3.2.1 (probability bound 3.1), for $n/p = 1$, it follows that, for all $0 \leq k < p$,

$$\Pr\left[|Y^{(k)}| = O\left(\frac{\lg n}{\lg \lg n}\right)\right] = 1 - O(n^{-\rho}).$$

Similarly, by invoking corollary 3.2.1 (probability bound 3.2), for $n/p = \omega(\lg n)$, it follows that, for all $0 \leq k < p$,

$$\Pr\left[|Y^{(k)}| = \left(1 + O\left(\sqrt{\frac{p \cdot \lg n}{n}}\right)\right) \cdot \frac{n}{p}\right] = 1 - O(n^{-\rho}).$$

The analysis of the remaining cases follows a line of arguments similar to that of the aforementioned cases and it is therefore omitted. The bound on $|D^{(k)}|$ follows similarly. $\square$

The resources required for procedure INSERTD are bounded as follows. For any constant $\rho > 0$, by way of corollary 3.2.6, the quasi-random distribution of items to processors (line 2) takes time $C_{e_qrand}(n, 1/2, 1, p)$ and $L + g \cdot O(|Y^{(k)}|) + M_{e_qrand}(n, 1/2, 1, p)$, for computation and communication respectively, with probability $1 - O(n^{-\rho})$. The insertion of the $|Y^{(k)}|$ items in the local dictionaries (line 4) requires time $\tau_{ID}(|Y^{(k)}|, |D^{(k)}|)$. Thus, insertion can be implemented efficiently by employing this simple quasi-randomized scheme in time

$$\tau_{ID}(|Y^{(k)}|, |D^{(k)}|) + C_{e_qrand}(n, 1/2, 1, p)$$

and

$$L + g \cdot O(|Y^{(k)}|) + M_{e_grnd}(n, 1/2, 1, p),$$

for computation and communication respectively.

For the deletion of n items (at most $\lceil n/p \rceil$ per processor), the following procedure is employed. Each processor sends its (at most) $\lceil n/p \rceil$ items to uniformly at quasi-random chosen processors, where they are deleted from their local dictionaries. The procedure to manage the delete operation on a PD D is DELETED($\{y_1, y_2, \dots, y_n\}, D$) and is outlined in figure 3.30. The find operation is implemented similarly.

```

begin DELETED ( $\{y_1, y_2, \dots, y_n\}, D$ )
1.  denote by  $Y$  set  $\{y_1, y_2, \dots, y_n\}$  ;
2.  distribute quasi-randomly the items of  $Y$  among the  $p$  processors ;
3.  for each processor  $p^{(k)}$ ,  $0 \leq k < p$ , in parallel
4.    do SEQ_DELETED( $Y^{(k)}, D^{(k)}$ ) ;
end DELETED

```

Figure 3.30: Procedure DELETED, p - number of processors.

Since the quasi-random distribution of items to processors results in identical distributions for both INSERTD and DELETED procedures, the correctness of procedure DELETED is easy to verify; it therefore remains to bound the resources required by the aforementioned scheme. The process of procedure DELETED is similar to that of INSERTD, and therefore, it is straightforward to verify that deletion can be implemented efficiently by employing this simple quasi-randomized scheme in time

$$\tau_{DD}(|Y^{(k)}|, |D^{(k)}|) + C_{e_grnd}(n, 1/2, 1, p)$$

and

$$L + g \cdot O(|Y^{(k)}|) + M_{e_grnd}(n, 1/2, 1, p),$$

for computation and communication respectively.

Analysis The correctness of the algorithm follows from its description. By summing up the computation and communication requirements of procedures INSERTD and DELETED we obtain the desired time bound. The analysis pertaining procedure FINDD is identical to that of procedure DELETED, and therefore, is omitted. The probability bound follows by way of claim 3.3.3 and corollary 3.2.6. Additionally, the methods employed in the aforementioned analysis make use of the quasi-random functions of section 3.2.6.3, and therefore, we assume an implicit preprocessing step for the construction of the underlying quasi-random function. By way of corollary 3.2.5, the time required for this preprocessing step is $T_{c_grnd}(n, 1/2, p)$.

The proposition follows. $\square$

Proof (of theorem 3.3.3) The theorem follows easily from proposition 3.3.2 by employing balanced search trees [54, 202] – as the underlying sequential PD – and substituting for T_{c_grnd} , T_{e_grnd} , T_{bcast} and

T_{ppf} the corresponding expressions in corollaries 3.2.5 and 3.2.6, and propositions 2.1.2 and 2.1.4, and letting $t = 2$ in these expressions. $\square$

We note that we can improve the performance of the INSERTD operation in several ways. For example, instead of implementing the insertion of $O(n/p)$ items per processor as a sequence of $O(n/p)$ single insert operations, we employ the following scheme on balanced search trees. To simplify the exposition, in the remainder of this section we assume that $n/p = \Omega(\lg n)$. Each processor constructs an auxiliary tree on the $(1+o(1)) \cdot (n/p)$ items it receives, as a result of the quasi-random distribution of items to processors (line 2), in time $\tau_{CD}((1+o(1)) \cdot (n/p)) = O((n/p) \cdot \lg(n/p))$ [54, 202], and subsequently joins this auxiliary tree with its local tree that is part of D . The latter operation requires time $\tau_{JD}((1+o(1)) \cdot (n/p), (1+o(1)) \cdot (m/p)) = O(\lg n/p + \lg m/p)$ [54, 202]. Overall, an improved $O((n/p) \cdot \lg(n/p) + \lg(m/p))$ computation time bound can be claimed for insertion. The following analogue of theorem 3.3.3 can then be derived.

Theorem 3.3.4 *For any constant $\rho > 0$, there exists a randomized p -processor BSP algorithm for supporting the operations insert, delete and find on a PD of size m , that for all integers n , m and p such that $m \geq n$ and $p = o(n/\lg n)$, requires time [with probability $1 - O(n^{-\rho})$]*

$$\begin{aligned} &O\left(\frac{n}{p} \cdot \lg \frac{n}{p} + \lg \frac{m}{p}\right) && \text{for insert,} \\ &(c_{del} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} && \text{for delete-min,} \\ &(c_{srch} + o(1)) \cdot \frac{n}{p} \cdot \lg \frac{m}{p} && \text{for find-min,} \end{aligned}$$

and

$$L \cdot O(1) + g \cdot O\left(\frac{n}{p}\right) \quad \text{for insert/delete/find,}$$

for computation and communication respectively. The algorithm requires $O(\sqrt{n}/p + \lg p) + L \cdot O(\lg p) + g \cdot O(\lg p)$ preprocessing time.

Remark Along the lines of PD operations, quasi-random functions can be employed to deliver optimal performance for PPQ operations. The significant advantage, however, of employing quasi-random hash functions is that the processor location of items can be easily determined – in contrast to purely random functions. For example, there exist applications involving parallel computations that generate tasks that have to be stored and subsequently scheduled for execution according to their priority (e.g., branch-and-bound applications [124, 125, 127]). As noted in [123], tasks are usually not independent of one another, and hence, they have to be able to communicate – employing quasi-random function allows the efficient communication of inter-dependent tasks.

3.3.4.1 Implementation and Experimental Results

In this section we discuss a *BSPlib* implementation – in C – of a generalized variant of the PD algorithms (INSERTD, DELETEmIND and FINDMIND) of the previous sections.

Implementation The prototypes adopted for the *BSPlib* dictionary procedures are outlined in figure 3.31.

The aim of functions (procedures) `BSP_insert_pd`, `BSP_delete_pd` and `BSP_find_pd` is to provide a data-type independent high-performance *BSPlib* dictionary implementation.¹⁸

¹⁸The terms generic and data-type independent imply that procedures `BSP_insert_pd`, `BSP_delete_pd` and `BSP_find_pd` can

```
void BSP_insert_pd (byte *src, pd **trg, int n, int *m, int sb, int (*cmp)()) ;
void BSP_delete_pd (byte **src, pd **trg, int n, int *m, int sb, int (*cmp)()) ;
void BSP_find_pd (byte **src, pd *trg, int n, int m, int sb, int (*cmp)()) ;
```

src - *input/output array.*
trg - *dictionary structure.*
n - *input size.*
m - *dictionary size.*
sb - *item size.*
cmp - *generic compare function.*

Figure 3.31: Procedure for BSP PD inserting, deleting and finding.

be employed to handle *any* data type. To this end, a comparison function `cmp` is used to facilitate the comparison of two elements e_1 and e_2 . In particular, the `cmp` function returns 1, 0 or -1 depending on whether e_1 is greater, equal or smaller than e_2 .

Chapter 4

Conclusion

4.1 Epilogue

The theme of this work has been the *architecture independent* study (*design, analysis* and *implementation*) of a multiplicity of fundamental parallel methods (*algorithms, data structures* and *techniques*). This work has neither been a comprehensive account of theoretical results on architecture independent parallel algorithms and data structures nor a thorough investigation of all available techniques. Still less has it been a programming manual containing a long catalogue of programs ready to be used directly on parallel machines to solve certain specific problems, but giving at best a concise abstraction of the ideas involved in their design and analysis. On the contrary, the aim of this work has been to give the reader some basic understanding of the principles involved in the architecture independent design and analysis of certain fundamental parallel algorithms and data structures. Although our approach has been rigorous and theoretical, we did not neglect the needs of practitioners.

We concentrated on the techniques used to design, analyze and subsequently implement efficient architecture independent algorithms and data structures. These architecture independent techniques included but were not limited to: *balanced trees* (e.g., broadcasting [section 2.1] and parallel-prefix [section 2.1]), *balancing* (e.g., load-balancing [section 2.1]), *pointer jumping* (e.g., list contraction [section 2.2]), *symmetry breaking* (e.g., list contraction [section 2.2]), *over-sampling* (e.g., merging [section 2.3], sorting, [section 2.6] and selection [section 2.8]), *partitioning* (e.g., merging [section 2.3], sorting [section 2.6]), *divide-and-conquer* (e.g., sorting [section 2.6] and computational geometry [section 2.10]), *cascading divide-and-conquer* (e.g., sorting [section 2.6]), *pipelining* (e.g., heaps [section 3.1] and search trees [section 3.2]) and *hashing* (e.g., hash tables [section 3.2]).

The main contribution of this work has been the unified investigation of the aforementioned techniques and the implementation of software that is transportable amongst pragmatic parallel machines having different number of processors, different periodicity of global synchronization and different bandwidth of inter-processor communication. The computational model adopted has been the bulk-synchronous parallel (BSP) model, which abstracts the characteristics of parallel machines into three numerical parameters p , L and g . The BSP parameters p , L and g , together with the problem size n , have been employed to measure the performance, and consequently, the transportability of parallel methods across machines having different values of these parameters.

We showed that optimality to within small multiplicative constant factors close to *one* can be achieved

for a multiplicity of fundamental computational problems by transportable algorithms and data structures that can be applied for a wide range of values of the BSP parameters. While these algorithms and data structures have been fairly simple themselves, description of their performance in terms of these parameters has been somewhat complicated. The main reward for quantifying these complications, has been that it enabled software to be written once and for all that could be migrated efficiently amongst a variety of parallel machines.

Appendix A

Probabilistic Results

A.1 Tail Inequalities

In this section we present some basic facts from probability theory. For a detailed account of the forthcoming material the reader should refer to [40, 165].

Theorem A.1.1 *Let $Y_1, Y_2, \dots, Y_n$ be independent random variables such that, for all $1 \leq i \leq n$, $0 \leq Y_i \leq z$. Then, for $Y = \sum_{i=1}^n Y_i$, $\mu = E[Y]$, and any $u > 0$,*

$$\Pr[Y \geq (1+u) \cdot \mu] \leq \left(\left(\frac{1}{1+u} \right)^{(1+u) \cdot (\mu/n)} \cdot \left(\frac{z - \mu/n}{z - (1+u) \cdot (\mu/n)} \right)^{z - (1+u) \cdot (\mu/n)} \right)^{n/z} \leq \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{\mu/z}.$$

Theorem A.1.2 *Let $X_1, X_2, \dots, X_n$ be independent Poisson trials such that, for all $1 \leq i \leq n$, $\Pr[X_i = 1] = P_i$, where $0 < P_i < 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = E[X] = \sum_{i=1}^n P_i$, and any $u > 0$,*

$$\Pr[X \geq (1+u) \cdot \mu] \leq \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^\mu.$$

The aforementioned tail inequalities (referred to as Chernoff's bounds) are attributed to [116] and [43] respectively. A simpler bound (due to [14]) is obtained for the special case $0 < u < 1$ and is summarized below [157].

Theorem A.1.3 *Let $X_1, X_2, \dots, X_n$ be independent Poisson trials such that, for all $1 \leq i \leq n$, $\Pr[X_i = 1] = P_i$, where $0 < P_i < 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = E[X] = \sum_{i=1}^n P_i$, and any $0 < \varepsilon < 1$,*

$$\Pr[X \geq (1+\varepsilon) \cdot \mu] \leq e^{-(1/3) \cdot \varepsilon^2 \cdot \mu},$$

and

$$\Pr[X \leq (1-\varepsilon) \cdot \mu] \leq e^{-(1/2) \cdot \varepsilon^2 \cdot \mu}.$$

Theorem A.1.4 *Let $Z_0, Z_1, \dots$ be a martingale sequence such that, for all i , $|Z_i - Z_{i-1}| \leq \vartheta_i$, where ϑ_i may depend on i . Then, for any $\lambda > 0$,*

$$\Pr[|Z_n - Z_0| \geq \lambda] \leq 2 \cdot e^{-\lambda^2 / (2 \cdot \sum_{i=1}^n \vartheta_i^2)}.$$

The aforementioned tail inequality (referred to as Azuma's inequality) is due to [19, 116], and various forms and applications are surveyed in [157].

A.1.1 Applications of Azuma's Inequality

Lemma A.1.1 *Let W be the number of bins containing exactly one ball when m balls are placed randomly into k bins. Then,*

$$\mu_1 = E[W] = \left(1 - \frac{1}{k}\right)^{m-1} \cdot m \sim e^{-(m-1)/k} \cdot m,$$

and for $\lambda > 0$,

$$\Pr[|W - \mu_1| \geq \lambda] \leq 2 \cdot e^{-(\lambda^2 / (2 + (m-1)/(k-1))^2) \cdot ((k-1/2)/(k^2 - (\mu_1 \cdot (k/m) \cdot (1-1/k))^2))}.$$

Proof The probability that a bin gets exactly i balls when m balls are placed randomly into k bins is

$$\binom{m}{i} \cdot \left(\frac{1}{k}\right)^i \cdot \left(1 - \frac{1}{k}\right)^{m-i}.$$

Therefore, the expected number of bins with exactly i balls is bounded above by

$$k \cdot \binom{m}{i} \cdot \left(\frac{1}{k}\right)^i \cdot \left(1 - \frac{1}{k}\right)^{m-i}.$$

Substituting in the above bound for $i = 1$ we obtain the desired expected number of bins with exactly one ball when m balls are placed randomly into k bins. In the remainder of this proof we concentrate on proving the tail bound.

Let $G_0 \subseteq G_1 \subseteq \dots \subseteq G_m$ be a filter, where G_t is generated by the random choice of the first t balls, that is, the set of events corresponding to the state of the bins at time t . Let W be the random variable denoting the number of bins with exactly one ball at time m , and let $W_t = E[W \mid G_t]$ denote the conditional expectation of W at time t . Then, the random variables $W_0, W_1, \dots, W_m$ form a martingale with $W_0 = E[W]$ and $W_m = W$.

Let $w(X, Y, t)$ denote the expectation of W given that X bins are empty at time t and Y bins contain exactly one ball. The probability that any of these empty bins receives exactly one ball during the last $m - t$ time units is given by $\binom{m-t}{1} \cdot (1/k) \cdot (1 - 1/k)^{m-t-1}$. Similarly, the probability that any of these bins containing exactly one ball does not receive a ball during the last $m - t$ time units is given by $(1 - 1/k)^{m-t}$. By linearity of expectations, we obtain that the number of these bins that contain exactly one ball at the end of the m placements is given by

$$w(X, Y, t) = X \cdot \binom{m-t}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t-1} + Y \cdot \left(1 - \frac{1}{k}\right)^{m-t}.$$

Let the random variable X_t denote the number of empty bins at time t , and, similarly, the random variable Y_t denote the number of bins containing exactly one ball at time t . Then,

$$W_{t-1} = w(X_{t-1}, Y_{t-1}, t-1) = X_{t-1} \cdot \binom{m-t+1}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} + Y_{t-1} \cdot \left(1 - \frac{1}{k}\right)^{m-t+1}.$$

At time $t-1$ the values of X_{t-1} , Y_{t-1} and W_{t-1} are known. Then, at time t , there are three possibilities. With probability X_{t-1}/k , the t -th ball goes into a currently empty bin. Then, $X_t = X_{t-1} - 1$, $Y_t = Y_{t-1} + 1$, and

$$W_t = w(X_{t-1} - 1, Y_{t-1} + 1, t) = (X_{t-1} - 1) \cdot \binom{m-t}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t-1} + (Y_{t-1} + 1) \cdot \left(1 - \frac{1}{k}\right)^{m-t}.$$

With probability Y_{t-1}/k , the t -th ball goes into a bin currently containing exactly one ball. Then, $X_t = X_{t-1}$, $Y_t = Y_{t-1} - 1$, and

$$W_t = w(X_{t-1}, Y_{t-1} - 1, t) = (X_{t-1} - 1) \cdot \binom{m-t}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t-1} + (Y_{t-1} - 1) \cdot \left(1 - \frac{1}{k}\right)^{m-t}.$$

With probability $1 - (X_{t-1} + Y_{t-1})/k$, the t -th ball goes into a bin currently containing more than one balls. Then, $X_t = X_{t-1}$, $Y_t = Y_{t-1}$, and

$$W_t = w(X_{t-1}, Y_{t-1}, t) = X_{t-1} \cdot \binom{m-t}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t-1} + Y_{t-1} \cdot \left(1 - \frac{1}{k}\right)^{m-t}.$$

We next focus on the difference random variable $\Delta_t = W_t - W_{t-1}$. Similarly to W_t , the distribution of Δ_t (given the state at time $t-1$) can be characterized as follows.

(1) With probability X_{t-1}/k , the value of Δ_t is

$$\begin{aligned} \delta_0 &= W_t - W_{t-1} \\ &= (X_{t-1} - 1) \cdot \binom{m-t}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t-1} + (Y_{t-1} + 1) \cdot \left(1 - \frac{1}{k}\right)^{m-t} - \\ &\quad X_{t-1} \cdot \binom{m-t+1}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} - Y_{t-1} \cdot \left(1 - \frac{1}{k}\right)^{m-t+1} \\ &= \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} \left(X_{t-1} \cdot \frac{m-t-k+1}{k-1} + Y_{t-1} - \frac{(m-t) \cdot k}{k-1} + k \right). \end{aligned}$$

Thus, since $1 \leq t \leq m-1$ and by definition $0 \leq X_{t-1} + Y_{t-1} \leq k$, it follows that the absolute value of δ_0 is bounded as follows:

$$\begin{aligned} |\delta_0| &= |W_t - W_{t-1}| \\ &\leq \left(2 + \frac{m-1}{k-1}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t}. \end{aligned}$$

(2) With probability Y_{t-1}/k , the value of Δ_t is

$$\begin{aligned}
 \delta_1 &= W_t - W_{t-1} \\
 &= X_{t-1} \cdot \binom{m-t}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t-1} + (Y_{t-1} - 1) \cdot \left(1 - \frac{1}{k}\right)^{m-t} - \\
 &\quad X_{t-1} \cdot \binom{m-t+1}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} - Y_{t-1} \cdot \left(1 - \frac{1}{k}\right)^{m-t+1} \\
 &= \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} \left(X_{t-1} \cdot \frac{m-t-k+1}{k-1} + Y_{t-1} - k \right).
 \end{aligned}$$

Thus, since $1 \leq t \leq m-1$ and by definition $0 \leq X_{t-1} + Y_{t-1} \leq k$, it follows that the absolute value of δ_1 is bounded as follows:

$$\begin{aligned}
 |\delta_1| &= |W_t - W_{t-1}| \\
 &\leq \left(2 + \frac{m-1}{k-1}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t}.
 \end{aligned}$$

(3) With probability $1 - (X_{t-1} + Y_{t-1})/k$, the value of Δ_t is

$$\begin{aligned}
 \delta_2 &= W_t - W_{t-1} \\
 &= X_{t-1} \cdot \binom{m-t}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t-1} + Y_{t-1} \cdot \left(1 - \frac{1}{k}\right)^{m-t} - \\
 &\quad X_{t-1} \cdot \binom{m-t+1}{1} \cdot \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} - Y_{t-1} \cdot \left(1 - \frac{1}{k}\right)^{m-t+1} \\
 &= \left(\frac{1}{k}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} \left(X_{t-1} \cdot \frac{m-t-k+1}{k-1} + Y_{t-1} \right).
 \end{aligned}$$

Thus, since $1 \leq t \leq m-1$ and by definition $0 \leq X_{t-1} + Y_{t-1} \leq k$, it follows that the absolute value of δ_2 is bounded as follows:

$$\begin{aligned}
 |\delta_2| &= |W_t - W_{t-1}| \\
 &\leq \left(1 + \frac{m-1}{k-1}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t}.
 \end{aligned}$$

It therefore follows that the value of the difference Δ_t is bounded by

$$-\left(2 + \frac{m-1}{k-1}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t} \leq \Delta_t \leq \left(2 + \frac{m-1}{k-1}\right) \cdot \left(1 - \frac{1}{k}\right)^{m-t}.$$

For $1 \leq t \leq m$, let $r = (m-1)/k$ and $\vartheta_t = (2 + (m-1)/(k-1)) \cdot (1 - 1/k)^{m-t}$. Then, we obtain that $|\Delta_t| = |W_t - W_{t-1}| \leq \vartheta_t$. By a straightforward calculation,

$$\begin{aligned}
\sum_{t=1}^m \vartheta_t^2 &= \left(2 + \frac{m-1}{k-1}\right)^2 \cdot \sum_{t=0}^{m-1} \left(1 - \frac{1}{k}\right)^{2 \cdot t} \\
&= \left(2 + \frac{m-1}{k-1}\right)^2 \frac{1 - (1 - (1/k))^{2 \cdot m}}{1 - (1 - (1/k))^2} \\
&= \left(2 + \frac{m-1}{k-1}\right)^2 \frac{k^2 - (\mu_1 \cdot (k/m) \cdot (1 - (1/k)))^2}{2 \cdot k - 1}.
\end{aligned}$$

The desired result is obtained by invoking theorem A.1.4 and substituting the above result for $\sum_{t=1}^m \vartheta_t^2$. $\square$

Lemma A.1.2 ([165]) *Let Z be the number of empty bins when m balls are placed randomly into k bins. Then,*

$$\mu_0 = \mathbb{E}[Z] = \left(1 - \frac{1}{k}\right)^m \cdot k \sim e^{-m/k} \cdot k,$$

and for $\lambda > 0$,

$$\Pr[|Z - \mu_0| \geq \lambda] \leq 2 \cdot e^{-\lambda^2 \cdot ((k-1)/2)/(k^2 - \mu_0^2)}.$$

Lemma A.1.3 ([62]) *Consider arbitrary $y_1, y_2, \dots, y_n > 0$ and let $Y_1, Y_2, \dots, Y_n$ be independent random variables such that, for any integer $k \geq 1$,*

$$\Pr[Y_i > k \cdot y_i] \leq 2^{-k}.$$

Then, for $Y = \sum_{i=1}^n Y_i$, $\mu = 2 \cdot \sum_{i=1}^n y_i$, $m = \max_i \{y_i\}$, and any integer $u \geq 0$,

$$\Pr[Y \geq 1.5 \cdot (1 + u) \cdot \mu] \leq e^{-u \cdot (\mu/(2m))}.$$

A.2 Randomized Results

In this section we present several randomized and probabilistic results on random sampling and hashing that provide the basis for the analysis of a multiplicity of efficient algorithms.

A.2.1 Random Sampling Applied to Sorting

In this section we present tight results related to random sampling – in the context of sorting – that generalize those of [75, 77]. In the discussion and proofs to follow we use the notation originally used in [75]. Let $X = \langle x_1, x_2, \dots, x_N \rangle$ be a set of size N , and for integers $k \geq 1$ and $s \geq 1$, let $Y = \langle y_1, y_2, \dots, y_{ks-1} \rangle$ be a randomly chosen sub-sequence of X , ordered such that $y_j < y_{j+1}$, for all $1 \leq j < k \cdot s - 1$. Without loss of generality, we assume that all the items appearing in sequence X are distinct – if the items are not distinct we can make them so by appending to them their index. The following three subsets are then defined.

$$\begin{aligned}
X_0 &= \{x \mid x \in X \text{ and } x < y_s\}, \\
X_i &= \{x \mid x \in X \text{ and } y_{is} < x < y_{(i+1)s}\}, \text{ for all } 1 \leq i \leq k-2, \\
X_{k-1} &= \{x \mid x \in X \text{ and } y_{(k-1)s} < x\}.
\end{aligned}$$

Let $Q_i(j)$ be the probability that $y_i = x_j$. Then, for all $1 \leq i \leq k \cdot s - 1$ and $1 \leq j \leq N$ the following equality holds [75].

$$Q_i(j) = \frac{\binom{j-1}{i-1} \cdot \binom{N-j}{k \cdot s - 1 - i}}{\binom{N}{k \cdot s - 1}},$$

For all i , $0 \leq i \leq k-1$, let b_i denote the size of X_i and let $P_i(j)$ be the probability that $b_i = j$. The following is shown in [75].

$$P_i(j) = P(j) = \frac{\binom{j}{s-1} \cdot \binom{N-j-1}{(k-1)s-1}}{\binom{N}{k \cdot s - 1}}.$$

In the remainder of this section we establish a tight upper and lower bounds on the size of the X_i 's. We next establish lower bounds on the size of the X_i 's.

Lemma A.2.1 *Consider positive integers n , N , k and s , and any ρ and ε such that $n \geq N$, $k \geq 2$, $k \cdot s < N/2$ and $0 < \varepsilon < 1$. Then, for any $\rho > 0$, if*

$$s \geq \frac{1-\varepsilon}{\varepsilon^2} \left(3 \cdot \rho \cdot \log n + \log \left(k^3 \cdot (2 \cdot \pi \cdot (k \cdot s - 1))^{3/2} \cdot e^{1/(2 \cdot (k \cdot s - 1))} \right) \right),$$

the probability that X_i , for all $0 \leq i \leq k-1$, is of size at least $\lfloor (1-\varepsilon) \cdot (N-k+1)/k \rfloor - 1$ is $1 - O(n^{-\rho})$.

Proof In order to establish an upper bound for the term

$$\sum_{j \leq \lfloor (1-\varepsilon) \cdot (N-k+1)/k \rfloor - 1} P(j),$$

we approximate $P(j)$ by a binomial term and use bounds on the right tail of the binomial distribution. Let $B = \lfloor (1-\varepsilon) \cdot (N-k+1)/k \rfloor - 1$ and $q = (k \cdot s - 1)/N$. The probability that in N Bernoulli trials, each with a probability of success q we get $s-1$ successes among the first j trials, a success at the $(j+1)$ -st trial, and $(k-1) \cdot s - 1$ successes in the remaining trials is

$$R(j) = \binom{j}{s-1} \cdot q^{s-1} \cdot (1-q)^{j-s+1} \cdot q \cdot \binom{N-j-1}{(k-1) \cdot s - 1} \cdot q^{(k-1) \cdot s - 1} (1-q)^{N-j-(k-1) \cdot s}.$$

Let A be the ratio of $R(j)$ over $P(j)$. The following then holds.

$$A = \frac{R(j)}{P(j)} = \binom{N}{k \cdot s - 1} \cdot q^{k \cdot s - 1} \cdot (1-q)^{N-k \cdot s + 1}.$$

The following inequality is shown in [40].

$$e^{-1/(6 \cdot u)} \cdot \frac{1}{\sqrt{2 \cdot \pi \cdot a \cdot b \cdot v}} \cdot (a^a \cdot b^b)^{-v} \leq \binom{v}{u}.$$

Substituting for $u = k \cdot s - 1$, $v = N$, $a = q = (k \cdot s - 1)/N$ and $b = 1 - a$ we obtain the following result.

$$\frac{e^{-1/(6 \cdot (k \cdot s - 1))} \cdot q^{-k \cdot s + 1} \cdot (1 - q)^{-N + k \cdot s - 1}}{\sqrt{2 \cdot \pi \cdot (k \cdot s - 1) \cdot (1 - (k \cdot s - 1)/N)}} \leq \binom{N}{k \cdot s - 1}.$$

It is then immediate that the following holds.

$$A \geq \frac{e^{-1/(6 \cdot (k \cdot s - 1))}}{\sqrt{2 \cdot \pi \cdot (k \cdot s - 1) \cdot (1 - (k \cdot s - 1)/N)}}. \quad (\text{A.1})$$

Thus, since $R(j) \leq \binom{j}{s-1} \cdot q^{s-1} \cdot (1 - q)^{j-s+1} \cdot q$ and $P(j) = R(j)/A$, we obtain the following result.

$$\sum_{j \leq B} P(j) \leq \frac{q}{A} \cdot \sum_B \binom{j}{s-1} \cdot q^{s-1} \cdot (1 - q)^{j-s+1}. \quad (\text{A.2})$$

In the remaining analysis we establish an upper bound for the following term by employing generating functions.

$$\sum_{j \leq B} \binom{j}{s-1} \cdot q^{s-1} \cdot (1 - q)^{j-s+1}.$$

Term $\binom{j}{s-1} \cdot q^{s-1} \cdot (1 - q)^{j-s+1}$ is the coefficient of x^{s-1} in the polynomial $((1 - q) + q \cdot x)^j$. Therefore, the sum over all the terms with $j \leq B$ is the coefficient of x^{s-1} in the following sum.

$$\sum_{j \leq B} ((1 - q) + q \cdot x)^j = \frac{((1 - q) + q \cdot x)^{B+1}}{-q \cdot (1 - x)} + \frac{1}{q \cdot (1 - x)}.$$

The coefficient of x^{s-1} in this expression is

$$\frac{1}{q} - \frac{1}{q} \cdot \sum_{i=0}^{s-1} \binom{B+1}{i} \cdot q^i \cdot (1 - q)^{B+1-i} = \frac{1}{q} \cdot \sum_{i=s}^{B+1} \binom{B+1}{i} \cdot q^i \cdot (1 - q)^{B+1-i}.$$

The sum in the term above gives the probability of having at least s successes in $B + 1$ independent Bernoulli trials with probability of success q . Therefore, by employing the inequality of theorem A.1.3 on the probability of having at least $(1 + \beta) \cdot (B + 1) \cdot q$ successes, we obtain an upper bound of $e^{-(\beta^2 \cdot (B+1) \cdot q)/3}$. By setting $s = (1 + \beta) \cdot (B + 1) \cdot q$ we further obtain $e^{-(\beta^2 \cdot (B+1) \cdot q)/3} = e^{-(s - (B+1) \cdot q)^2 / (3 \cdot (B+1) \cdot q)}$. We first derive a lower bound on the term $s - (B + 1) \cdot q$, for $0 < \varepsilon < 1$ and $k \cdot s \leq N/2$.

$$\begin{aligned} s - (B + 1) \cdot q &\geq s - \left(\left\lfloor (1 - \varepsilon) \cdot \frac{N - k + 1}{k} \right\rfloor \right) \frac{k \cdot s - 1}{N} \\ &\geq s - s \cdot (1 - \varepsilon) + \frac{(1 - \varepsilon) \cdot (k - 1) \cdot k \cdot s + (1 - \varepsilon) \cdot (N - k + 1)}{N \cdot k} \\ &\geq \varepsilon \cdot s. \end{aligned}$$

An upper bound on $(B + 1) \cdot q$ is derived similarly.

$$\begin{aligned} (B + 1) \cdot q &\leq \left(\left\lfloor (1 - \varepsilon) \cdot \frac{N - k + 1}{k} \right\rfloor \right) \frac{k \cdot s - 1}{N} \\ &\leq s \cdot (1 - \varepsilon) + \frac{-(1 - \varepsilon) \cdot (k - 1)k \cdot s - (1 - \varepsilon) \cdot (N - k + 1)}{N \cdot k} \\ &\leq (1 - \varepsilon) \cdot s. \end{aligned}$$

The combination of these two bounds gives $e^{-\beta^2 \cdot (B+1) \cdot q/3} \leq e^{-\varepsilon^2 \cdot s/(3 \cdot (1-\varepsilon))}$. We thus obtain the following inequality for the sum of interest.

$$\begin{aligned} \sum_{j \leq B} \binom{j}{s-1} \cdot q^{s-1} \cdot (1-q)^{j-s+1} &= \frac{1}{q} \cdot \sum_{i=s}^{B+1} \binom{B+1}{i} \cdot q^i \cdot (1-q)^{B+1-i} \\ &\leq \frac{1}{q} \cdot e^{-\varepsilon^2 \cdot s/(3 \cdot (1-\varepsilon))}. \end{aligned} \quad (\text{A.3})$$

The desired bound follows from equations A.1, A.2 and A.3, and is given by the following expression.

$$\sum_{j \leq B} P(j) \leq \sqrt{2 \cdot \pi \cdot (k \cdot s - 1)} \cdot e^{1/(6(k \cdot s - 1))} \cdot e^{-\varepsilon^2 \cdot s/(3 \cdot (1-\varepsilon))}.$$

Substituting for the value of s , gives the desired result. $\square$

Lemma A.2.2 ([75]) Consider positive integers n, N, k and s , and any ρ and ε such that $n \geq N, k \geq 2, k \cdot s < N/2$ and $0 < \varepsilon < 1$. Then, for any $\rho > 0$, if

$$s \geq \frac{1 + \varepsilon}{\varepsilon^2} \left(2 \cdot \rho \cdot \log n + \log \left(k^2 \cdot (2 \cdot \pi \cdot (k \cdot s - 1)) \cdot e^{1/(3 \cdot (k \cdot s - 1))} \right) \right),$$

the probability that X_i , for all $0 \leq i \leq k - 1$, is of size at most $\lfloor (1 + \varepsilon) \cdot (N - k + 1)/k \rfloor + 1$ is $1 - O(n^{-\rho})$.

A.2.2 Random Sampling Applied to Selection

In this section we present tight results related to random sampling – in the context of selection – that generalize those of [75]. In the discussion and proofs to follow we use the notation originally used in [75]. Let $X = \langle x_1, x_2, \dots, x_N \rangle$ be a set of size N , and for integers $\lambda \geq 1$ and $s \geq 1$, let $Y = \langle y_1, y_2, \dots, y_{2\lambda s - 1} \rangle$ be a randomly chosen sub-sequence of X , ordered such that $y_j < y_{j+1}$, for all $1 \leq j < 2 \cdot \lambda \cdot s - 1$. Without loss of generality, we assume that all the items appearing in sequence X are distinct – if the items are not distinct we can make them so by appending to them their index. The following three subsets are then defined for some positive integers μ_0, μ_2 and t , and any c such that $2 \cdot c \cdot \lambda \cdot s > \mu_0 \cdot (t/2)$ and $1/N \leq c \leq 1/2$.

$$\begin{aligned} X_0 &= \{x \mid x \in X \text{ and } x < y_{2c\lambda s - \mu_0(t/2)}\}, \\ X_1 &= \{x \mid x \in X \text{ and } y_{2c\lambda s - \mu_0(t/2)} < x < y_{2c\lambda s + \mu_2(t/2)}\}, \\ X_2 &= \{x \mid x \in X \text{ and } y_{2c\lambda s + \mu_2(t/2)} < x\}. \end{aligned}$$

For $i = 0, 2$, let $b_i = |X_i|$ and let $P_i(j)$ be the probability that $b_i = j$. The following is shown in [75].

For $i = 0, 2$

$$P_i(j) = P(j) = \frac{\binom{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1}{j} \cdot \binom{N - j - 1}{2 \cdot (1 - c) \cdot \lambda \cdot s + \mu \cdot (t/2) - 1}}{\binom{N}{2 \cdot \lambda \cdot s - 1}}.$$

Let $B = c \cdot N - f(N) - 1$, for some non-decreasing function $f(N)$ of N . In the remainder of this section we establish a tight upper and lower bounds on the size of sets X_0 and X_2 , or equivalently, on the term $\sum_{j \leq B} P_i(j)$, $i = 0, 2$. We next establish lower bounds on the size of sets X_0 and X_2 .

Lemma A.2.3 Consider positive integers n, N, λ, μ, s and t , and any ρ and c such that $n \geq N$, $2 \cdot \lambda \cdot s - 1 < N/2$, $2 \cdot c \cdot \lambda \cdot s > \mu \cdot (t/2)$ and $1/N \leq c \leq 1/2$. Then, for any $\rho > 1$, if $\mu \cdot t = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot c \cdot \lambda \cdot s \cdot \log n}$ and $f(N) = (N \cdot \mu \cdot t)/(2 \cdot \lambda \cdot s)$, the probability that sets X_0 and X_2 are of size at least $c \cdot N - f(N)$ is $1 - O(n^{1-\rho})$.

Proof We first observe that the combinatorial identity [132],

$$\sum_{a-u \leq v \leq r-b} \binom{u+v}{a} \cdot \binom{r-v}{b} = \binom{r+u+1}{a+b+1},$$

implies that

$$\sum_{j=2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1}^{N - 2 \cdot (1 - c) \cdot \lambda \cdot s - \mu \cdot (t/2)} \binom{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1}{j} \cdot \binom{N - j - 1}{2 \cdot (1 - c) \cdot \lambda \cdot s + \mu \cdot (t/2) - 1} = \binom{N}{2 \cdot \lambda \cdot s - 1}.$$

By a combinatorial interpretation argument it is not difficult to verify that the term $\sum_{j \leq B} P(j)$ expresses the number of ways we can choose $2 \cdot \lambda \cdot s - 1$ objects so that among them, the object with the $(2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2))$ -th highest rank is labeled at most $B = c \cdot N - (N \cdot \mu \cdot t)/(2 \cdot \lambda \cdot s) - 1$.

In order to establish an upper bound for $\sum_{j \leq B} P(j)$ we approximate $P(j)$ by a binomial term and use bounds on the right tail of the binomial distribution. Let $q = (2 \cdot \lambda \cdot s - 1)/N$. The probability that in N Bernoulli trials, each with a probability of success q we get $2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1$ successes among the first j trials, a success at the $(j + 1)$ -st trial, and $2 \cdot (1 - c) \cdot \lambda \cdot s + \mu \cdot (t/2) - 1$ successes in the remaining trials is

$$R(j) = \binom{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1}{j} \cdot q^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot (1 - q)^{j - 2 \cdot c \cdot \lambda \cdot s + \mu \cdot (t/2) + 1} \cdot q \cdot \binom{N - j - 1}{2 \cdot (1 - c) \cdot \lambda \cdot s + \mu \cdot (t/2) - 1} \cdot q^{2 \cdot (1 - c) \cdot \lambda \cdot s + \mu \cdot (t/2) - 1} \cdot (1 - q)^{N - j - 2 \cdot (1 - c) \cdot \lambda \cdot s - \mu \cdot (t/2)}.$$

Let A be the ratio of $R(j)$ over $P(j)$. The following then holds.

$$A = \frac{R(j)}{P(j)} = \binom{N}{2 \cdot \lambda \cdot s - 1} \cdot q^{2 \cdot \lambda \cdot s - 1} \cdot (1 - q)^{N - 2 \cdot \lambda \cdot s + 1}.$$

The following inequality is shown in [40].

$$e^{-1/(6 \cdot u)} \cdot \frac{1}{\sqrt{2 \cdot \pi \cdot a \cdot b \cdot v}} \cdot (a^a \cdot b^b)^{-v} \leq \binom{v}{u}.$$

By substituting for $u = 2 \cdot \lambda \cdot s - 1$, $v = N$, $a = q = (2 \cdot \lambda \cdot s - 1)/N$ and $b = 1 - a$ we obtain the following result.

$$\frac{e^{-1/(6 \cdot (2 \cdot \lambda \cdot s - 1))} \cdot q^{-2 \cdot \lambda \cdot s + 1} \cdot (1 - q)^{-N + 2 \cdot \lambda \cdot s - 1}}{\sqrt{2 \cdot \pi \cdot (2 \cdot \lambda \cdot s - 1) \cdot (1 - (2 \cdot \lambda \cdot s - 1)/N)}} \leq \binom{N}{2 \cdot \lambda \cdot s - 1}.$$

It is then immediate that the following holds.

$$A \geq \frac{e^{-1/(6(2\lambda s - 1))}}{\sqrt{2 \cdot \pi \cdot (2 \cdot \lambda \cdot s - 1) \cdot (1 - (2 \cdot \lambda \cdot s - 1)/N)}}. \quad (\text{A.4})$$

Thus, since $R(j) = \binom{j}{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot q^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot (1 - q)^{j - 2 \cdot c \cdot \lambda \cdot s + \mu \cdot (t/2) + 1} \cdot q$ and $P(j) = R(j)/A$, we obtain the following result.

$$\sum_{j \leq B} P(j) \leq \frac{q}{A} \cdot \sum_{j \leq B} \binom{j}{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot q^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot (1 - q)^{j - 2 \cdot c \cdot \lambda \cdot s + \mu \cdot (t/2) + 1}. \quad (\text{A.5})$$

In the remaining analysis we establish an upper bound for the following term by employing generating functions.

$$\sum_{j \leq B} \binom{j}{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot q^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot (1 - q)^{j - 2 \cdot c \cdot \lambda \cdot s + \mu \cdot (t/2) + 1}. \quad (\text{A.6})$$

Term $\binom{j}{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot q^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \cdot (1 - q)^{j - 2 \cdot c \cdot \lambda \cdot s + \mu \cdot (t/2) + 1}$ is the coefficient of $x^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1}$ in the polynomial $((1 - q) + q \cdot x)^j$. Therefore, the sum over all the terms with $j \leq B$ is the coefficient of $x^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1}$ in the following sum.

$$\sum_{j \leq B} ((1 - q) + q \cdot x)^j = \frac{((1 - q) + q \cdot x)^{B+1}}{-q \cdot (1 - x)} + \frac{1}{q \cdot (1 - x)}.$$

The coefficient of $x^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1}$ in this expression is

$$\frac{1}{q} - \frac{1}{q} \cdot \sum_{i=0}^{2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - 1} \binom{B+1}{i} \cdot q^i \cdot (1 - q)^{B+1-i} = \frac{1}{q} \cdot \sum_{i=2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2)}^{B+1} \binom{B+1}{i} \cdot q^i \cdot (1 - q)^{B+1-i}.$$

The sum in the term above gives the probability of having at least $2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2)$ successes in $B + 1$ independent Bernoulli trials with probability of success q . Thus, by employing the inequality in theorem A.1.3 on the probability of having at least $(1 + \beta) \cdot (B + 1) \cdot q$ successes, we obtain an upper bound of $e^{-(\beta^2 \cdot (B+1) \cdot q)/3}$. By setting $2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) = (1 + \beta) \cdot (B + 1) \cdot q$ we further obtain $e^{-(\beta^2 \cdot (B+1) \cdot q)/3} = e^{-(2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - (B+1) \cdot q)^2 / (3 \cdot (B+1) \cdot q)}$. By substituting function $B = c \cdot N - (N \cdot \mu \cdot t) / (2 \cdot \lambda \cdot s) - 1$, the following lower bound on $2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - (B + 1) \cdot q$ is derived.

$$2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - (B+1) \cdot q \geq \mu \cdot (t/2) - (\mu \cdot t)/(2 \cdot \lambda \cdot s).$$

An upper bound on $(B+1) \cdot q$ is derived similarly.

$$(B+1) \cdot q \leq (2 \cdot c \cdot \lambda \cdot s - \mu \cdot t) + (\mu \cdot t)/(2 \cdot \lambda \cdot s).$$

The combination of these two bounds gives $e^{-(2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - (B+1) \cdot q)^2 / (3 \cdot (B+1) \cdot q)} \leq e^{-(1/3) \cdot (\mu \cdot (t/2))^2 / (2 \cdot c \cdot \lambda \cdot s)}$. We thus obtain the following upper bound for the sum of interest (refer to expression A.6).

$$\frac{1}{q} \cdot \sum_{i=2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2)}^{B+1} \binom{B+1}{i} \cdot q^i \cdot (1-q)^{B+1-i} \leq e^{-(2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - (B+1) \cdot q)^2 / (3 \cdot (B+1) \cdot q)}. \quad (\text{A.7})$$

The desired bound follows from equations A.4, A.5 and A.7, and is given by the following expression.

$$\sum_{j \leq B} P(j) \leq \sqrt{2 \cdot \pi \cdot (2 \cdot \lambda \cdot s - 1)} \cdot e^{1/(6 \cdot (2 \cdot \lambda \cdot s - 1))} \cdot e^{-(2 \cdot c \cdot \lambda \cdot s - \mu \cdot (t/2) - (B+1) \cdot q)^2 / (3 \cdot (B+1) \cdot q)}.$$

Substituting for $\mu \cdot t = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot c \cdot \lambda \cdot s \cdot \log n}$ and noting that $\sqrt{2 \cdot \pi \cdot (2 \cdot \lambda \cdot s - 1)} \cdot e^{1/(6 \cdot (2 \cdot \lambda \cdot s - 1))} < 6 \cdot n$, gives the desired result. $\square$

We can similarly establish upper bounds on the size of sets X_0 and X_2 .

Lemma A.2.4 Consider positive integers n, N, λ, μ, s and t , and any ρ and c such that $n \geq N$, $2 \cdot \lambda \cdot s - 1 < N/2$, $2 \cdot c \cdot \lambda \cdot s > \mu \cdot (t/2)$ and $1/N \leq c \leq 1/2$. Then, for any $\rho > 1$, if $\mu \cdot t = 2 \cdot \sqrt{2 \cdot \rho \cdot 2 \cdot c \cdot \lambda \cdot s \cdot \log n}$, the probability that sets X_0 and X_2 are of size at most $c \cdot N - 1$ is $1 - O(n^{1-\rho})$.

Proof The proof follows that of lemma A.2.3 with $f(N) = 0$. $\square$

The following bounds on the size of X_0 and X_2 can then be derived for $s = t = 1$.

Claim A.2.1 Consider positive integers n, N, λ and μ_0 , and any ρ and c such that $n \geq N$, $2 \cdot \lambda - 1 < N/2$, $4 \cdot c \cdot \lambda > \mu_0$, $\mu_0 = 2 \cdot \sqrt{2 \cdot \rho \cdot 2 \cdot c \cdot \lambda \cdot \log n}$ and $1/N \leq c \leq 1/2$. Then, for any $\rho > 1$, the probability that set X_0 is of size at most $c \cdot N - 1$ is $1 - O(n^{1-\rho})$.

Proof The claim is a direct consequence of lemma A.2.4 by substituting for $s = t = 1$. $\square$

Claim A.2.2 Consider positive integers n, N, λ and μ_2 , and any ρ and c such that $n \geq N$, $2 \cdot \lambda - 1 < N/2$, $4 \cdot (1-c) \cdot \lambda > \mu_2$, $\mu_2 = 2 \cdot \sqrt{2 \cdot \rho \cdot 2 \cdot (1-c) \cdot \lambda \cdot \log n}$ and $1/N \leq (1-c) \leq 1/2$. Then, for any $\rho > 1$, the probability that set X_2 is of size at most $N - c \cdot N - 1$ is $1 - O(n^{1-\rho})$.

Proof The claim is a direct consequence of lemma A.2.4 by substituting for $s = t = 1$. $\square$

Claim A.2.3 Consider positive integers n, N, λ and μ , and any ρ and c such that $n \geq N$, $2 \cdot \lambda - 1 < N/2$, $4 \cdot c \cdot \lambda > \mu$, $\mu_0 = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot c \cdot \lambda \cdot \log n}$ and $1/N \leq c \leq 1/2$. Then, for any $\rho > 1$, the probability that set X_0 is of size at least $c \cdot N - N \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot c \cdot \log n} / \sqrt{\lambda}$ is $1 - O(n^{1-\rho})$.

Proof The claim is a direct consequence of lemma A.2.3 by substituting for $s = t = 1$ and $\mu = \mu_0 = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot c \cdot \lambda \cdot \log n}$. $\square$

Claim A.2.4 Consider positive integers n, N, λ and μ , and any ρ and c such that $n \geq N$, $2 \cdot \lambda - 1 < N/2$, $4 \cdot (1 - c) \cdot \lambda > \mu$, $\mu_0 = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1 - c) \cdot \lambda \cdot \log n}$ and $1/N \leq (1 - c) \leq 1/2$. Then, for any $\rho > 1$, the probability that set X_2 is of size at least $N - c \cdot N - N \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1 - c) \cdot \log n} / \sqrt{\lambda} - 1$ is $1 - O(n^{1-\rho})$.

Proof The claim is a direct consequence of lemma A.2.3 by substituting for $s = t = 1$ and $\mu = \mu_0 = 2 \cdot \sqrt{3 \cdot \rho \cdot 2 \cdot (1 - c) \cdot \lambda \cdot \log n}$. $\square$

A.2.3 Hashing

In the remainder of this section we present several of the central properties of random members of H_s^d and $H_{r,s}^{d_1,d_2}$ (refer to section 3.2.6.2 for definitions).

Definition Consider any $c > 0$ and integer $d \geq 1$. Then, multi-set $\mathcal{H} \subseteq \{h \mid h : U \rightarrow \{0, 1, \dots, s-1\}\}$ is called (c, k) -universal, if for all sequences $\langle x_1, x_2, \dots, x_d \rangle$ of distinct elements of U , all sequences $\langle y_1, y_2, \dots, y_d \rangle$ of distinct elements of $\{0, 1, \dots, s-1\}$, and randomly chosen $h \in \mathcal{H}$ the following holds.

$$\Pr[h(x_i) = y_i] \leq \frac{c}{s^d}, \quad 1 \leq i \leq d.$$

Lemma A.2.5 ([42, 158]) If multi-set $\mathcal{H}$ is (c, d) -universal and $r : U \rightarrow \{0, 1, \dots, s-1\}$ is such that $|r^{-1}(j)| \leq k$, for all $j \in \{0, 1, \dots, s-1\}$, then multi-set $\mathcal{H}' = \{r \circ h \mid h \in \mathcal{H}\}$ is (c', d) -universal, for $c' = c \cdot (k \cdot (s/|Q|))^d$.

Remark Lemma A.2.5 implies that sets H_s^d and $H_{r,s}^{d_1,d}$ are $(\gamma, d+1)$ -universal, for $\gamma = (\lceil q/s \rceil \cdot (s/q))^{d+1} \leq (1 + s/q)^{d+1}$ [158].

Definition Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and consider function $h : U \rightarrow \{0, 1, \dots, s-1\}$ to be chosen at random from some universal class of hash functions.

- The j -th bucket induced by h on S is $B_j^h = \{x \in S \mid h(x) = j\}$, $0 \leq j < s$.
- The size of the j -th bucket B_j^h is $b_j^h = |B_j^h|$, $0 \leq j < s$.
- h is called c -perfect for S if $b_j^h \leq c$, for all $0 \leq j < s$.

Lemma A.2.6 ([65]) Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and consider function $h : U \rightarrow \{0, 1, \dots, s-1\}$ to be chosen at random from $\mathcal{H}$. If $\mathcal{H}$ is $(\gamma, d+1)$ -universal, then

$$\Pr[b_j^h \geq u] \leq \begin{cases} \gamma \cdot \left(\frac{e^{u-1}}{u^u}\right) \cdot \left(\frac{n}{s}\right)^u & \text{if } 1 \leq u < d+1, \\ \gamma \cdot \left(\frac{e^d}{u^{d+1}}\right) \cdot \left(\frac{n}{s}\right)^{d+1} & \text{if } d+1 \leq u \end{cases} \quad 0 \leq j < s.$$

Lemma A.2.7 ([65]) *Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and consider function $h : U \rightarrow \{0, 1, \dots, s-1\}$ to be chosen at random from $\mathcal{H}$. If $\mathcal{H}$ is $(\gamma, d+1)$ -universal, then*

$$\Pr[h \text{ is } d\text{-perfect for } S] \geq 1 - \frac{\gamma}{(d+1)!} \cdot n \cdot \left(\frac{n}{s}\right)^d.$$

Lemma A.2.8 ([65]) *Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n \leq s$, and consider function $h : U \rightarrow \{0, 1, \dots, s-1\}$ to be chosen at random from $\mathcal{H}$. If $\mathcal{H}$ is $(\gamma, d+1)$ -universal, then*

$$\mathbb{E} \left[\sum_{j=0}^{s-1} \left(b_j^h\right)^k \right] \leq c_k \cdot n, \text{ for any } 1 \leq k \leq d+1,$$

where $c_k = (k-1)^{k-1} + \gamma \cdot e^{k-1}$.

Lemma A.2.9 ([137]) *Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and consider function $h : U \rightarrow \{0, 1, \dots, s-1\}$ to be chosen at random from $\mathcal{H}$. Assume any $0 < \delta < 1$ such that $s = n^{1-\delta}$ and $n \geq (d+1) \cdot s/2$. If $\mathcal{H}$ is $(\gamma, d+1)$ -universal, then, for any $u > 0$,*

$$\Pr \left[b_j^h \leq (1+u) \cdot n^\delta \right] \geq 1 - \alpha \cdot \left(\frac{1}{u}\right)^{d+1} \cdot n^{-\delta \cdot (d+1)/2}, \quad 0 \leq j < s.$$

The following theorem is a generalization of a result due to [62].

Theorem A.2.1 *Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n$, and consider function h to be chosen at random from $H_{r,s}^{d_1,d_2}$. Assume any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq (d_1+1)r/2$. Then, with probability $1 - O(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + (\gamma/(d_2+1)!) \cdot n^{\delta \cdot c+1} \cdot (2/s)^{d_2})$, for any $u > 0$ and $0 < \varepsilon < 1$,*

$$\Pr \left[b_j^h \geq (1+u) \cdot \frac{n}{s} \right] \leq \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n/(d_2 \cdot s)}, \quad 0 \leq j < s \quad (\text{A.8})$$

and

$$\Pr \left[b_j^h \geq (1+\varepsilon) \cdot \frac{n}{s} \right] \leq e^{-(1/3) \cdot \varepsilon^2 \cdot n/(d_2 \cdot s)}, \quad 0 \leq j < s, \quad (\text{A.9})$$

where $\alpha = ((d+1)/2)^{d+2}/((d+1)/2)!$ and $\gamma \leq (1+s/q)^{d_2+1}$.

Proof Let

$$R = \left\{ h \in H_{r,s}^{d_1,d_2} \mid b_i^f \leq 2 \cdot \frac{n}{r} \text{ and } \hat{f} \text{ is } d_2\text{-perfect for } B_i^f, \text{ for all } 0 \leq i < r \right\}.$$

Along the lines of [62], we show that the aforementioned class of hash functions has the desired properties. Assume $|S| = n$ (otherwise add some dummy elements). Consider fixed $f \in H_r^{d_1}$ and $\hat{f} \in H_s^{d_2}$ such that $b_i^f \leq 2 \cdot (n/r)$ and $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$. It suffices to show that the assertion holds for the probability space generated by the random choice of $a_0, a_1, \dots, a_{r-1}$. With respect to this probability space, denote the probability and expectation by $\Pr_{f,\hat{f}}[\dots]$ and $\mathbb{E}_{f,\hat{f}}[\dots]$ respectively. Fix j , $0 \leq j < s$, and define the random variable Y_i as follows:

$$Y_i = \left| \left\{ x \in B_i^f \mid \hat{f}(x) + a_i \equiv j \pmod{s} \right\} \right|, \quad 0 \leq i < r.$$

Then,

$$\mathbb{E}_{f,\hat{f}}[Y_i] = \frac{1}{s} \cdot \sum_{a=0}^{s-1} \left| \left\{ x \in B_i^f \mid \hat{f}(x) + a \equiv j \pmod{s} \right\} \right| = \frac{1}{s} \cdot b_i^f, \quad 0 \leq i \leq r.$$

The independence of the a_i s implies the independence of the Y_i s, and therefore, we obtain the following for the mean expected value μ of $\sum_i Y_i = b_j^h$.

$$\mu = \mathbb{E}_{f,\hat{f}} \left[\sum_{i=0}^{r-1} Y_i \right] = \sum_{i=0}^{r-1} \mathbb{E}_{f,\hat{f}}[Y_i] = \sum_{i=0}^{r-1} \left(\frac{1}{s} \cdot b_i^f \right) = \frac{|S|}{s} = \frac{n}{s}.$$

The desired result is obtained by observing that $0 \leq Y_i \leq d_2$ and applying theorem A.1.1 (the former observation follows from the fact that $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$).

$$\Pr_{f,\hat{f}} \left[b_j^h \geq (1+u) \cdot \frac{n}{s} \right] \leq \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n/(d_2 \cdot s)}, \quad 0 \leq j < s.$$

By the right half of the following inequality [106],

$$-(1/2) \cdot \varepsilon^2 \leq \varepsilon - (1+\varepsilon) \cdot \log(1+\varepsilon) \leq -(1/3) \cdot \varepsilon^2, \quad 0 < \varepsilon < 1,$$

we further conclude that

$$\Pr_{f,\hat{f}} \left[b_j^h \geq (1+\varepsilon) \cdot \frac{n}{s} \right] \leq e^{-(1/3) \cdot \varepsilon^2 \cdot n/(d_2 \cdot s)}, \quad 0 \leq j < s.$$

It remains to establish that for suitably chosen d_1 , d_2 , r and s , a randomly chosen $h \in H_{r,s}^{d_1,d_2}$ satisfies $h \in R$, that is, $b_i^f \leq 2 \cdot (n/r)$ and $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$. Fix $a_0, a_1, \dots, a_{r-1}$ and assume any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq (d_1 + 1) \cdot r/2$. Then, by way of lemma A.2.9, it follows that a randomly chosen $f \in H_r^{d_1}$ satisfies $b_i^f \leq 2 \cdot n^\delta$, for all $0 \leq i < r$, with probability

$$1 - O \left(n^{1-\delta} \cdot \alpha \cdot n^{-\delta \cdot (d_1+1)/2} \right) = 1 - O \left(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} \right),$$

where $\alpha = ((d+1)/2)^{d+2}/((d+1)/2)!$. Given such an f , by way of lemma A.2.7 and sub-additivity, a randomly chosen $\hat{f} \in H_s^{d_2}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$, with probability

$$1 - O \left(n^{1-\delta} \cdot \left(\frac{\gamma}{(d_2+1)!} \cdot 2 \cdot n^\delta \cdot \left(\frac{2 \cdot n^\delta}{s} \right)^{d_2} \right) \right) = 1 - O \left(\frac{\gamma}{(d_2+1)!} \cdot n^{\delta \cdot d_2+1} \left(\frac{2}{s} \right)^{d_2} \right),$$

where $\gamma \leq (1+s/q)^{d_2+1}$. The desired result follows trivially. $\square$

The following result is a non-trivial adaptation of a result due to [64].

Theorem A.2.2 Let q be prime, $S \subseteq U = \{0, 1, \dots, q-1\}$, $|S| \leq n \leq s$, and consider function h to be chosen at random from $H_{r,s}^{d_1,d_2}$. Assume any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq (d_1+1) \cdot r/2$. Then, with probability $1 - O(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + (\gamma/(d_2+1)!) \cdot n^{\delta \cdot d_2+1} \cdot (2/s)^{d_2} + n^{-(1+(\lg \beta + \lg s - \lg n)/\lg \lg n - o(1)) \cdot (\beta/d_2)} \cdot s)$, for any $\beta > 0$ and $u > 0$,

$$\Pr \left[\sum_{j=0}^{s-1} (b_j^h)^k \geq (c_k + u) \cdot n \right] \leq 2 \cdot e^{-u^2 \cdot n^{1-\delta-2 \cdot k \cdot (\lg \beta / \lg n + o(1))} / 8}, \text{ for any } k \geq 1,$$

where $\alpha = ((d+1)/2)^{d+2} / ((d+1)/2)!$, $\gamma \leq (1+s/q)^{d_2+1}$ and $c_k = (k-1)^{k-1} + \gamma \cdot e^{k-1}$.

Proof Let

$$R = \left\{ h \in H_{r,s}^{d_1,d_2} \mid b_i^f \leq 2 \cdot \frac{n}{r} \text{ and } \hat{f} \text{ is } d_2\text{-perfect for } B_i^f, \text{ for all } 0 \leq i < r \right\}.$$

We show that the aforementioned class of hash functions has the desired property. Recall that in theorem A.2.1 it is shown that for suitably chosen d_1 , d_2 , r and s , a randomly chosen $h \in H_{r,s}^{d_1,d_2}$ satisfies $h \in R$, with probability $1 - O(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + (\gamma/(d_2+1)!) \cdot n^{\delta \cdot d_2+1} \cdot (2/s)^{d_2})$. Assume $|S| = n$ (otherwise add some dummy elements), and consider fixed $f \in H_r^{d_1}$ and $\hat{f} \in H_s^{d_2}$ such that $b_i^f \leq 2 \cdot (n/r)$ and $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$. In view of lemma A.2.8, it suffices to show that the assertion holds for the probability space generated by the random choice of $a_0, a_1, \dots, a_{r-1}$. With respect to this probability space, denote the probability and expectation by $\Pr_{f,\hat{f}}[\dots]$ and $\mathbb{E}_{f,\hat{f}}[\dots]$ respectively. For any $\beta > 0$, by way of theorem A.2.1, we have that

$$\Pr_{f,\hat{f}} \left[b_j^h \geq \left(\beta \cdot \frac{\lg n}{\lg \lg n} \right) \right] = O \left(n^{-(1+(\lg \beta + \lg s - \lg n)/\lg \lg n - o(1)) \cdot (\beta/d_2)} \right), \quad 0 \leq j < s.$$

We subsequently define truncated versions of the bucket sizes. In particular, define

$$\hat{b}_j^h = \min \left\{ b_j^h, \left(\beta \cdot \frac{\lg n}{\lg \lg n} \right) \right\}, \quad 0 \leq j < s.$$

Similarly, define

$$M_k = \sum_{j=0}^{s-1} (b_j^h)^k \quad \text{and} \quad \hat{M}_k = \sum_{j=0}^{s-1} (\hat{b}_j^h)^k.$$

Then, $\Pr_{f,\hat{f}}[M_k \neq \hat{M}_k] = O(s \cdot n^{-(1+(\lg \beta + \lg s - \lg n)/\lg \lg n - o(1)) \cdot (\beta/d_2)})$. Therefore, since $\mathbb{E}_{f,\hat{f}}[\hat{M}_k] \leq \mathbb{E}_{f,\hat{f}}[M_k] \leq c_k \cdot n$ (refer to lemma A.2.8), it suffices to establish a bound on

$$\Pr_{f,\hat{f}} \left[\hat{M}_k \geq \mathbb{E}[\hat{M}_k] + u \cdot n \right].$$

Let $G_0 \subseteq G_1 \subseteq \dots \subseteq G_r$ be a filter, where G_t is generated by the random choice of the first t of the a_i 's, that is, the set of events corresponding to the state of the buckets given that the first t of the a_i 's have been fixed. Let Z be the random variable denoting $\sum_j (\hat{b}_j^h)^k$ given that all of the a_i 's have been fixed, and let $Z_t = \mathbb{E}[Z \mid G_t]$ denote the conditional expectation of Z given that the first t of the a_i 's have

been fixed. Then, the random variables $Z_0, Z_1, \dots, Z_r$ form a martingale with $Z_0 = \mathbb{E}[Z] = \mathbb{E}[\hat{M}_k]$ and $Z_r = Z = \hat{M}_k$.

By assumption, $b_t^f \leq 2 \cdot n^\delta$, and therefore, if a_t is changed, then at most $2 \cdot n^\delta$ of the $\hat{b}_j^h$'s are affected (increased or decreased). Furthermore, by definition, $\hat{b}_j^h \leq \beta \cdot (\lg n / \lg \lg n)$; thus, the overall change (increment or decrement) $\Delta_t = Z_t - Z_{t-1}$ of $\hat{M}_k$ in response to changing a_t is bounded as follows:

$$-2 \cdot n^\delta \cdot \left(\beta \cdot \frac{\lg n}{\lg \lg n} \right)^k \leq \Delta_t \leq 2 \cdot n^\delta \cdot \left(\beta \cdot \frac{\lg n}{\lg \lg n} \right)^k.$$

For $1 \leq t \leq r$, we set $\vartheta_t = 2 \cdot n^{\delta+k \cdot (\lg \beta / \lg n + o(1))}$, and we therefore have that $|\Delta_t| = |Z_t - Z_{t-1}| = \vartheta_t$. By a straightforward calculation,

$$\sum_{t=1}^r \vartheta_t^2 = n^{1-\delta} \cdot \left(2 \cdot n^{\delta+k \cdot (\lg \beta / \lg n + o(1))} \right)^2 = 4 \cdot n^{1+\delta+2 \cdot k \cdot (\lg \beta / \lg n + o(1))}.$$

The desired bound on M_k is obtained by invoking theorem A.1.4 and substituting the above expression for $\sum_{t=1}^m \vartheta_t^2$.

$$\Pr_{f, \hat{f}} \left[\hat{M}_k \geq \mathbb{E}[\hat{M}_k] + u \cdot n \right] \leq 2 \cdot e^{-(u \cdot n)^2 / (8 \cdot n^{1+\delta+2 \cdot k \cdot (\lg \beta / \lg n + o(1))})}.$$

The desired result follows trivially. $\square$

The following theorem is a generalization of a result due to [62].

Theorem A.2.3 *Let q be prime, $\bar{S} \subseteq S \subseteq U = \{0, 1, \dots, q-1\}$, $|\bar{S}| = \bar{n}$, $|S| \leq n$, and consider function h to be chosen at random from $H_{r,s}^{d_1, d_2}$. Assume any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq (d_1 + 1) \cdot r/2$. Then, with probability $1 - O(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + (\gamma/(d_2 + 1)!) \cdot n^{\delta \cdot d_2+1} \cdot (2/s)^{d_2})$, for any $u > 0$,*

$$\Pr \left[\left| \{x \in S \mid \exists \bar{x} \in \bar{S} \ (h(x) = h(\bar{x}))\} \right| \geq \left((1+u) \cdot \frac{n}{s} + d_2 \right) \cdot \bar{n} \right] \leq \frac{\gamma}{(d_2 + 1)!} \cdot \left(2 \cdot n^\delta \cdot \bar{n} \right)^{d_2+1} \cdot \left(\frac{1}{s} \right)^{d_2} + \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n \cdot \bar{n} / (2 \cdot d_2 \cdot n^\delta \cdot s)},$$

where $\alpha = ((d+1)/2)^{d+2} / ((d+1)/2)!$ and $\gamma \leq (1+s/q)^{d_2+1}$.

Proof Let

$$R = \left\{ h \in H_{r,s}^{d_1, d_2} \mid b_i^f \leq 2 \cdot \frac{n}{r} \text{ and } \hat{f} \text{ is } d_2\text{-perfect for } B_i^f, \text{ for all } 0 \leq i < r \right\}.$$

We show that the aforementioned class of hash functions has the desired property. Recall that in theorem A.2.1 it is shown that for suitably chosen d_1, d_2, r and s , a randomly chosen $h \in H_{r,s}^{d_1, d_2}$ satisfies $h \in R$, with probability $1 - O(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + (\gamma/(d_2 + 1)!) \cdot n^{\delta \cdot d_2+1} \cdot (2/s)^{d_2})$. Assume $|S| = n$ (otherwise add some dummy elements), and consider fixed $f \in H_r^{d_1}$ and $\hat{f} \in H_s^{d_2}$ such that $b_i^f \leq 2 \cdot (n/r)$ and $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$. It suffices to show that the assertion holds for the probability space generated by the random choice of $a_0, a_1, \dots, a_{r-1}$. With respect to this probability space, denote

the probability and expectation by $\Pr_{f,\hat{f}}[\dots]$ and $\mathbb{E}_{f,\hat{f}}[\dots]$ respectively. Define $\bar{I} = \{i \mid B_i^f \cap \bar{S} \neq \{\}\}$ and $I = \{0, 1, \dots, r-1\} - \bar{I}$. Furthermore, let

$$z_{\bar{I}} = \left| \left\{ x \in \bigcup_{i \in \bar{I}} B_i^f \mid \exists \bar{x} \in \bar{S} \ (h(x) = h(\bar{x})) \right\} \right|$$

and

$$z_I = \left| \left\{ x \in \bigcup_{i \in I} B_i^f \mid \exists \bar{x} \in \bar{S} \ (h(x) = h(\bar{x})) \right\} \right|.$$

The quantity to be estimated is $z_{\bar{I}} + z_I$. It is straightforward to verify that $|\bar{I}| \leq \bar{n}$, and thus, $|\cup_{i \in \bar{I}} B_i^f| \leq 2 \cdot n^\delta \cdot \bar{n}$ (note that by assumption $b_i^f \leq 2 \cdot n^\delta$). Fix a_i , for all $i \in \bar{I}$. Then, by invoking lemma A.2.7, we obtain that a randomly chosen $\hat{f} \in H_s^{d_2}$ is d_2 -perfect for $\cup_{i \in \bar{I}} B_i^f$, with probability at least

$$1 - \frac{\gamma}{(d_2 + 1)!} \cdot 2 \cdot n^\delta \cdot \bar{n} \cdot \left(\frac{2 \cdot n^\delta \cdot \bar{n}}{s} \right)^{d_2} = 1 - \frac{\gamma}{(d_2 + 1)!} \cdot (2 \cdot n^\delta \cdot \bar{n})^{d_2+1} \cdot \left(\frac{1}{s} \right)^{d_2}.$$

We therefore conclude that

$$\Pr_{f,\bar{f}}[z_{\bar{I}} \geq d_2 \cdot \bar{n}] \leq \frac{\gamma}{(d_2 + 1)!} \cdot (2 \cdot n^\delta \cdot \bar{n})^{d_2+1} \cdot \left(\frac{1}{s} \right)^{d_2}.$$

It remains to establish an upper bound on z_I . Consider fixed a_i , for all $i \in \bar{I}$, such that $\hat{f} \in H_s^{d_2}$ is d_2 -perfect for $\cup_{i \in \bar{I}} B_i^f$. It suffices to bound z_I for the probability space generated by the random choice of a_i , for $i \in I$. With respect to this probability space, denote the probability and expectation by $\Pr_{f,\hat{f},\bar{I}}[\dots]$ and $\mathbb{E}_{f,\hat{f},\bar{I}}[\dots]$ respectively. Define the random variable Y_i as follows:

$$Y_i = \left| \left\{ x \in B_i^f \mid \exists \bar{x} \in \bar{S} \ (h(x) = h(\bar{x})) \right\} \right|, \quad i \in I.$$

Then,

$$\mathbb{E}_{f,\hat{f},\bar{I}}[Y_i] = \frac{1}{s} \cdot \sum_{a=0}^{s-1} \left| \left\{ x \in B_i^f \mid \exists \bar{x} \in \bar{S} \ (\hat{f}(x) + a_{f(x)} \equiv \hat{f}(\bar{x}) + a \pmod{s}) \right\} \right| \leq \frac{1}{s} \cdot b_i^f \cdot |\bar{S}|, \quad i \in I.$$

The independence of the a_i s, for $i \in I$, implies the independence of the Y_i s, and therefore, we obtain the following for the mean expected value μ of $\sum_{i \in I} Y_i = z_I$.

$$\mu = \mathbb{E}_{f,\hat{f},\bar{I}} \left[\sum_{i \in I} Y_i \right] = \sum_{i \in I} \mathbb{E}_{f,\hat{f},\bar{I}}[Y_i] \leq \sum_{i \in I} \left(\frac{1}{s} \cdot b_i^f \cdot |\bar{S}| \right) \leq \frac{|S|}{s} \cdot |\bar{S}| = \frac{n}{s} \cdot \bar{n}.$$

The desired result is obtained by observing that $0 \leq Y_i \leq 2 \cdot d_2 \cdot n^\delta$ and applying theorem A.1.1 (the former observation follows from the facts that $B_i^f \leq 2 \cdot n^\delta$ and $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$).

$$\Pr_{f, \bar{f}, \bar{I}} \left[z_I \geq (1+u) \cdot \frac{n}{s} \cdot \bar{n} \right] \leq \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n \cdot \bar{n} / (2 \cdot d_2 \cdot n^\delta \cdot s)}.$$

□

Theorem A.2.4 *Let q be prime, $\bar{S} \subseteq S \subseteq U = \{0, 1, \dots, q-1\}$, $|\bar{S}| = \bar{n}$, $|S| \leq n$, $\bar{J} = \{j \mid 0 \leq j < s \text{ and } B_j^h \cap \bar{S} \neq \{\}\}$, and consider function h to be chosen at random from $H_{r,s}^{d_1, d_2}$. Assume any $0 < \delta < 1$ such that $r = n^{1-\delta}$ and $n \geq (d_1 + 1) \cdot r/2$. Then, with probability $1 - O(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + (\gamma/(d_2 + 1)!) \cdot n^{\delta \cdot d_2+1} \cdot (2/s)^{d_2})$, for any $u > 0$,*

$$\Pr \left[b_j^h \geq (1+u) \cdot \frac{n}{s} + d_2 \right] \leq \frac{\gamma}{(d_2 + 1)!} \cdot (2 \cdot n^\delta \cdot \bar{n})^{d_2+1} \cdot \left(\frac{1}{s} \right)^{d_2} + \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n/(d_2 \cdot s)}, \quad \bar{j} \in \bar{J},$$

where $\alpha = ((d+1)/2)^{d+2} / ((d+1)/2)!$ and $\gamma \leq (1 + s/q)^{d_2+1}$.

Proof Let

$$R = \left\{ h \in H_{r,s}^{d_1, d_2} \mid b_i^f \leq 2 \cdot \frac{n}{r} \text{ and } \hat{f} \text{ is } d_2\text{-perfect for } B_i^f, \text{ for all } 0 \leq i < r \right\}.$$

We show that the aforementioned class of hash functions has the desired property. Recall that in theorem A.2.1 it is shown that for suitably chosen d_1 , d_2 , r and s , a randomly chosen $h \in H_{r,s}^{d_1, d_2}$ satisfies $h \in R$, with probability $1 - O(\alpha \cdot n^{1-\delta \cdot ((d_1+1)/2+1)} + (\gamma/(d_2 + 1)!) \cdot n^{\delta \cdot d_2+1} \cdot (2/s)^{d_2})$. Assume $|S| = n$ (otherwise add some dummy elements), and consider fixed $f \in H_r^{d_1}$ and $\hat{f} \in H_s^{d_2}$ such that $b_i^f \leq 2 \cdot (n/r)$ and $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$. It suffices to show that the assertion holds for the probability space generated by the random choice of $a_0, a_1, \dots, a_{r-1}$. With respect to this probability space, denote the probability and expectation by $\Pr_{f, \hat{f}}[\dots]$ and $\mathbb{E}_{f, \hat{f}}[\dots]$ respectively. Define $\bar{I} = \{i \mid B_i^f \cap \bar{S} \neq \{\}\}$ and $I = \{0, 1, \dots, r-1\} - \bar{I}$. Furthermore, let

$$z_{\bar{I}} = \left| \left\{ x \in \bigcup_{i \in \bar{I}} B_i^f \mid \hat{f}(x) + a_i \equiv \bar{j} \pmod{s} \right\} \right|$$

and

$$z_I = \left| \left\{ x \in \bigcup_{i \in I} B_i^f \mid \hat{f}(x) + a_i \equiv \bar{j} \pmod{s} \right\} \right|.$$

The quantity to be estimated is $z_{\bar{I}} + z_I$. It is straightforward to verify that $|\bar{I}| \leq \bar{n}$, and thus, $|\cup_{i \in \bar{I}} B_i^f| \leq 2 \cdot n^\delta \cdot \bar{n}$ (note that by assumption $b_i^f \leq 2 \cdot n^\delta$). Fix a_i , for all $i \in \bar{I}$. Then, by invoking lemma A.2.7, we obtain that a randomly chosen $\hat{f} \in H_s^{d_2}$ is d_2 -perfect for $\cup_{i \in \bar{I}} B_i^f$, with probability at least

$$1 - \frac{\gamma}{(d_2 + 1)!} \cdot 2 \cdot n^\delta \cdot \bar{n} \cdot \left(\frac{2 \cdot n^\delta \cdot \bar{n}}{s} \right)^{d_2} = 1 - \frac{\gamma}{(d_2 + 1)!} \cdot (2 \cdot n^\delta \cdot \bar{n})^{d_2+1} \cdot \left(\frac{1}{s} \right)^{d_2}.$$

We therefore conclude that

$$\Pr_{f,\bar{f}}[z_{\bar{I}} \geq d_2] \leq \frac{\gamma}{(d_2 + 1)!} \cdot (2 \cdot n^\delta \cdot \bar{n})^{d_2+1} \cdot \left(\frac{1}{s}\right)^{d_2}.$$

It remains to establish an upper bound on z_I . Consider fixed a_i , for all $i \in \bar{I}$, such that $\hat{f} \in H_s^{d_2}$ is d_2 -perfect for $\cup_{i \in \bar{I}} B_i^f$. It suffices to bound z_I for the probability space generated by the random choice of a_i , for $i \in I$. With respect to this probability space, denote the probability and expectation by $\Pr_{f,\hat{f},\bar{I}}[\dots]$ and $E_{f,\hat{f},\bar{I}}[\dots]$ respectively. Fix $\bar{j}, \bar{j} \in \bar{J}$, and define the random variable Y_i as follows:

$$Y_i = \left| \left\{ x \in B_i^f \mid \hat{f}(x) + a_i \equiv \bar{j} \pmod{s} \right\} \right|, \quad i \in I.$$

Then,

$$E_{f,\hat{f},\bar{I}}[Y_i] = \frac{1}{s} \cdot \sum_{a=0}^{s-1} \left| \left\{ x \in B_i^f \mid \hat{f}(x) + a \equiv \bar{j} \pmod{s} \right\} \right| \leq \frac{1}{s} \cdot b_i^f, \quad i \in I.$$

The independence of the a_i s, for $i \in I$, implies the independence of the Y_i s, and therefore, we obtain the following for the mean expected value μ of $\sum_{i \in I} Y_i = z_I$.

$$\mu = E_{f,\hat{f},\bar{I}} \left[\sum_{i \in I} Y_i \right] = \sum_{i \in I} E_{f,\hat{f},\bar{I}}[Y_i] \leq \sum_{i \in I} \left(\frac{1}{s} \cdot b_i^f \right) \leq \frac{|S|}{s} = \frac{n}{s}.$$

The desired result is obtained by observing that $0 \leq Y_i \leq d_2$ and applying theorem A.1.1 (the former observation follows from the fact that $\hat{f}$ is d_2 -perfect for B_i^f , for all $0 \leq i < r$).

$$\Pr_{f,\hat{f}} \left[b_{\bar{j}}^h \geq (1+u) \cdot \frac{n}{s} \right] \leq \left(\frac{e^u}{(1+u)^{(1+u)}} \right)^{n/(d_2 \cdot s)}, \quad \bar{j} \in \bar{J}.$$

□

Appendix B

A Sample Program

B.1 Source Code

In this section we present the source code – in C – of the BSP deterministic sorting algorithm of subsection 2.6.6.2.

B.1.1 BSP Deterministic Sorting

BSP_det_sort.c – *A high-performance generic BSP implementation of the deterministic sorting algorithm of section 2.6.6 (non-recursive version).*

BSP_det_sort.c

	<code>#include ...</code>
	<code>#define byte char</code>
<i>over-sampling factor</i>	<code>#define W_N(N) max(1, (log2(log2(N))))</code>
	<code>int BSP_det_sort</code>
<i>input array</i>	<code>(byte *src,</code>
<i>output array</i>	<code>byte **trg,</code>
<i>input size (per proc)</i>	<code>int n,</code>
<i>item size</i>	<code>int sb,</code>
<i>merge function</i>	<code>void (*merge)(),</code>
<i>sort function</i>	<code>void (*sort)(),</code>
<i>compare function</i>	<code>int (*cmp)())</code>
	<code>{</code>
	<code>byte *aux, *smpl, *ssmpl, *splt ;</code>

```

int    *c, *tc, *t, ntrg, nsmpl, nt, stp, rnd ;
int    np, pid ;
int    i, j, k, l, m, N ;

if ((n <= 0) || (sb <= 0)) { *trg = NULL ; return n ; }

get proc id and          pid = BSP_pid() ; np = BSP_np() ;
  number of procs
compute global input     BSP_reduce(plus, 1, &n, &N, sizeof(int)) ;
  size
compute sample size      nsmpl = W_N(N) * np ;
                          if (nsmpl > n) BSP_abort("Sample too large.") ;

allocate buffers         new(aux, byte, n * sb) ;
                          new(smpl, byte, nsmpl * sb) ;
                          new(spl, byte, np * sb) ;
                          new(c, int, np) ;
                          new(tc, int, np) ;
                          new(t, int, np) ;

register input-output    BSP_pushregister(spl, np * sb) ;
  buffers                BSP_pushregister(t, np * sizeof(int)) ;
                          BSP_sync() ;

sort input array into    memcpy(aux, src, n * sb) ;
  auxiliary buffer       sort(aux, n, sb, cmp) ;

select sample            for (j = 0, stp = n / nsmpl ; j < (nsmpl - 1) ; ++j)
                          memcpy(&smpl[j*sb], &aux[((j+1)*stp-1)*sb], sb) ;
                          memcpy(&smpl[j*sb], &aux[(n-1)*sb], sb) ;

globally sort sample     nsmpl = BSP_bitonic_sort(smpl, &ssmpl, nsmpl, sb, cmp) ;

communicate splitters    for (i = 0 ; i < np ; ++i)
                          BSP_put(i, &ssmpl[(nsmpl-1)*sb], spl, pid * sb, sb) ;
                          BSP_sync() ;

initialize counter       memset(c, 0, np * sizeof(int)) ;
count items per bucket   for (i = np - 2, c[np-1] = n ; i >= 0 ; -i) {
                          c[i] = search(&spl[i*sb], aux, n, sb, cmp) ;
                          c[i+1] -= c[i] ;
                          }

```

```

compute prefix-sum      BSP_prefix(plus, np, c, tc, sizeof(int)) ;

communicate prefix-sum   for (i = 0 ; i < np ; ++i)
  results                BSP_put(i, &tc[i], t, pid * sizeof(int), sizeof(int)) ;
                        BSP_sync() ;

allocate target buffer   nt = t[np-1] ;
                        new(*trg, byte, nt * sb) ;

register target buffer    BSP_pushregister(*trg, nt * sb) ;
                        BSP_sync() ;

send items to target     for (i = 0, k = 0 ; i < np ; k += c[i], ++i)
  processor              BSP_put(i, &aux[k*sb], *trg, (tc[i] - c[i]) * sb, c[i] * sb) ;
                        BSP_sync() ;

de-allocate auxiliary    free(aux) ;
  buffer

merge input              for (i = 1 ; i < np ; i *= 2)
                        for (l = 0, k = 0 ; l < (np / 2 + 1) ; k = t[m], l += 2 * i) {
                          m = min(l + 2 * i, np) - 1 ;
                          merge(&(*trg)[k*sb], &(*trg)[t[l+i-1]*sb], (t[l+i-1] - k),
                                (t[m] - t[l+i-1]), sb, cmp) ;
                        }

load balance output      BSP_load_balance(*trg, &aux, nt, &ntrg, sb) ;

de-register buffers      BSP_popregister(*trg) ;
                        BSP_popregister(t) ;
                        BSP_popregister(spl t) ;

de-allocate buffers      free(*trg) ; *trg = aux ;
                        free(t) ;
                        free(tc) ;
                        free(c) ;
                        free(spl t) ;
                        free(smpl) ;
                        free(ssmpl) ;

                        BSP_sync() ;

return output size        return ntrg ;
}

```


Bibliography

- [1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, 1993.
- [2] G. M. Adel’son-Vel’skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.
- [3] A. Aggarwal, B. Chazelle, L. Guibas, C. O’Dunlaig, and C. Yap. Parallel computational geometry. In *Proceedings of the 26-th IEEE Symposium on Foundations of Computer Science*, pp. 468–477, 1985.
- [4] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.
- [5] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \cdot \lg n$ steps. *Combinatorica*, 2:1–19, 1983.
- [6] S. G. Akl. An optimal algorithm for parallel selection. *Information Processing Letters*, 19:47–50, 1984.
- [7] S. G. Akl. *Parallel Sorting Algorithms*. Academic Press, Florida, 1985.
- [8] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11):1367–1669, 1987.
- [9] S. G. Akl and K. A. Lyons. *Parallel Computational Geometry*. Prentice Hall, New Jersey, 1993.
- [10] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of the 7-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 129–136, 1995.
- [11] A. G. Alexandrakis, A. V. Gerbessiotis, and C. J. Siniolakis. Portable and scaleable algorithm design on the IBM SP2: The bulk-synchronous parallel paradigm. In *Proceedings of the Sup-Eur’96 Conference*, Krakow, Poland, September 1996.
- [12] A. G. Alexandrakis, A. V. Gerbessiotis, D. S. Lecomber, and C. J. Siniolakis. Bandwidth, space and computation efficient PRAM programming: The BSP approach. In *Proceedings of the Sup-Eur’96 Conference*, Krakow, Poland, September 1996.
- [13] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33:269–273, 1990.
- [14] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18:155–193, 1979.

- [15] M. J. Attalah and M. T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(4):535–548, 1988.
- [16] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal on Computing*, 18(3):499–532, 1989.
- [17] M. J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J. Tsay. Multi-search techniques for implementing data structures on a mesh-connected computer. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 204–214, 1991.
- [18] M. J. Attalah and M. T. Goodrich. Deterministic parallel computational geometry. In *Synthesis of Parallel Algorithms*, (J. H. Reif, ed.), Morgan Kaufmann, 1993.
- [19] K. Azuma. Weighted sums of certain dependent random variables. *Tohoku Mathematical Journal*, 19:357–367, 1967.
- [20] D. A. Bader, D. R. Helman, and J. JáJá. Practical parallel algorithms for personalized communication and integer sorting. Technical Report UMIACS-TR-95-101/CS-TR-3548, The University of Maryland Institute for Advanced Computer Studies, 1995.
- [21] C. K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. DB2 parallel edition. *IBM Systems Journal*, 34(2):292–322, 1995.
- [22] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, pp. 307–314, 1968.
- [23] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C-27(1):84–87, January 1978.
- [24] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c -Optimal multi-search for an extension of the BSP model. In *Proceedings of the European Symposium on Algorithms*, pp. 17–30, 1995.
- [25] A. Baumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-Optimal multi-search for an extension of the BSP model. Technical Report, Department of Mathematics and Computer Science, University of Paderborn, 1996.
- [26] A. Baumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP Model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, pp. 233–242, June 1996.
- [27] A. Baumker, W. Dittrich, F. Meyer auf der Heide, and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP* model. In *Proceedings of Euro-Par'96 Parallel Processing, Lecture Notes in Computer Science* 1124, Springer-Verlag, pp. 369–376, August 1996.
- [28] R. Bayer and E. M. McReight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.
- [29] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

- [30] R. Bayer and M. Schkolnick. Concurrency of operations on *B*-trees. *Acta Informatica*, 9(1):1–21, 1977.
- [31] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15-th ACM Symposium on Theory of Computing*, pp. 80–86, 1983.
- [32] J. L. Bentley and M. I. Shamos. A problem in multivariate statistics: Algorithms, data structure and applications. In *Proceedings of the 15-th Allerton Conference on Communication, Control, and Computing*, pp. 193–201, 1973.
- [33] B. Bergsten, M. Couprie, and P. Valduriez. Overview of parallel architectures for databases. *The Computer Journal*, 36(8):734–740, 1993.
- [34] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.
- [35] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs. LogP. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 25–32, 1996.
- [36] A. Biliris. Operation specific locking in *B*-trees. In *Proceeding of the ACM Symposium on the Principles of Database Systems*, pp. 159–169, 1987.
- [37] J. Biswas and J. C. Browne. Simultaneous update of priority structures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 124–131, 1987.
- [38] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, 1989.
- [39] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.
- [40] B. Bollobás. *Random Graphs*. Academic Press, New York, 1984.
- [41] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.
- [42] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proceedings of the 9-th ACM Symposium on Theory of Computing*, pp. 106–112, 1977.
- [43] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.
- [44] B. S. Chlebus. Parallel iterated bucket sort. *Information Processing Letters*, 31:181–183, 1989.
- [45] A. L. Chow. A parallel algorithm for determining convex hulls of sets of points in two dimensions. In *Proceedings of the 9-th Allerton Conference on Communication, Control and Computing*, Illinois, pp. 214–223, 1981.
- [46] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Computation*, 70:32–53, 1986.
- [47] R. Cole. Parallel Merge Sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

- [48] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17(1):128–142, 1988.
- [49] R. Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26:295–299, 1988.
- [50] R. Cole and M. T. Goodrich. Optimal parallel algorithms for point-set and polygon problems. *Algorithmica*, 7:3–23, 1992.
- [51] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
- [52] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7:354–375, 1973.
- [53] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.
- [54] H. T. Cormen, C. E. Leiserson, and L. R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [55] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical parallel model of computation. *Communications of the ACM*, 39(11):78–85, 1996.
- [56] W. Cunto and J. I. Munro. Average-case selection. In *Proceedings of the 16-th ACM Symposium on Theory of Computing*, pp. 369–375, 1984.
- [57] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hyper-cube multiprocessor, and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing*, 8:367–375, 1990.
- [58] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scaleable parallel geometric algorithms for coarse grained multi-computers. In *Proceedings of the 9-th ACM Symposium on Computational Geometry*, San Diego, pp. 298–307, 1993.
- [59] F. Dehne. Scaleable parallel computational geometry. In *Proceedings of Parallel and Distributed Computing, Lecture Notes in Computer Science 805*, (G. Goos and J. Hartmanis, eds.), Springer-Verlag, pp. 115–119, 1994.
- [60] N. Deo, A. Jain, and M. Medini. An optimal parallel algorithm for merging using multi-selection. *Information Processing Letters*, 55:295–299, 1994.
- [61] D. J. DeWitt and J. Gray. Parallel database systems: The future of high-performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.
- [62] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of ICALP'90: Automata, Languages and Programming, Lecture Notes in Computer Science 443*, Springer-Verlag, pp. 6–19, July 1990.
- [63] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proceedings of the 22-nd ACM Symposium on Theory of Computing*, pp. 117–127, 1990.

- [64] M. Dietzfelbinger and F. Meyer auf der Heide. High-performance universal hashing, with applications to shared memory applications. In *Proceedings of Data Structures and Efficient Algorithms, Lecture Notes in Computer Science* 594, Springer-Verlag, pp. 250–269, 1992.
- [65] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.
- [66] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.
- [67] C. S. Ellis. Concurrent search and insert in 2-3 trees. *Acta Informatica*, 14(1):63–86, 1980.
- [68] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proceedings of the 8-th ACM Symposium on Computational Geometry*, Berlin, pp. 74–82, 1992.
- [69] R. W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.
- [70] R. W. Floyd and R. L. Rivest. Expected-time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.
- [71] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10-th ACM Symposium on Theory of Computing*, pp. 114–118, 1978.
- [72] R. S. Francis and I. D. Mathieson. A benchmark parallel sort for shared memory multiprocessors. *IEEE Transactions on Computers*, C-37(12):1619–1626, 1988.
- [73] W. D. Frazer and A. C. McKellar. Sample-sort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496–507, 1970.
- [74] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst-case access time. *Journal of the ACM*, 31(3):538–544, 1984.
- [75] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.
- [76] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report PRG-TR-13-96, Computing Laboratory, Oxford University, May 1996.
- [77] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996.
- [78] A. V. Gerbessiotis and C. J. Siniolakis. Concurrent heaps on the BSP model. Technical Report PRG-TR-14-96, Computing Laboratory, Oxford University, June 1996.
- [79] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the bulk-synchronous parallel model with applications to priority queues. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA, August 1996.

- [80] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of Euro-Par'96 Parallel Processing, Lecture Notes in Computer Science* 1124, Springer-Verlag, August 1996.
- [81] A. V. Gerbessiotis and C. J. Siniolakis. Efficient deterministic sorting on the BSP model. Technical Report PRG-TR-19-96, Oxford University Computing Laboratory, October 1996.
- [82] A. V. Gerbessiotis and C. J. Siniolakis. Primitive operations on the BSP model. Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996.
- [83] A. V. Gerbessiotis and C. J. Siniolakis. Converting sorting algorithms to fast randomized ones via searching. Submitted for journal publication, 1996.
- [84] A. V. Gerbessiotis and C. J. Siniolakis. An experimental study of sorting algorithms on the BSP model: Theory, practice and experience. Technical Report PRG-TR-06-97, Computing Laboratory, Oxford University, February 1997.
- [85] A. V. Gerbessiotis and C. J. Siniolakis. Dynamic data structures: The BSP approach. Technical Report PRG-TR-11-97, Computing Laboratory, Oxford University, March 1997.
- [86] A. V. Gerbessiotis and C. J. Siniolakis. A note on probabilistic integer sorting. Technical Report PRG-TR-14-97, Computing Laboratory, Oxford University, March 1997.
- [87] A. V. Gerbessiotis and C. J. Siniolakis. A randomized sorting algorithm on the BSP model. In *Proceedings of International Parallel Processing Symposium (IPPS'97)*, April 1997.
- [88] A. V. Gerbessiotis, C. J. Siniolakis, and A. Tiskin. Parallel priority queues and list contraction: The BSP approach. In *Proceedings of Euro-Par'97 Parallel Processing, Lecture Notes in Computer Science*, Springer-Verlag, August 1997.
- [89] A. V. Gerbessiotis and C. J. Siniolakis. On probabilistic integer-sorting. In *Proceedings of Workshop on Randomized Algorithms in Sequential, Parallel and Distributed Computing (RALCOM'97)*, October 1997.
- [90] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan. A PRAM programming environment. Technical Report PRG-TR-21-97, Computing Laboratory, Oxford University, June 1997.
- [91] A. V. Gerbessiotis and C. J. Siniolakis. Converting sorting algorithms to fast randomized ones via searching. Technical Report PRG-TR-22-97, Computing Laboratory, Oxford University, June 1997.
- [92] A. V. Gerbessiotis and C. J. Siniolakis. Efficient deterministic sorting on the BSP model. To appear in *Parallel Processing Letters*, 1997-8.
- [93] A. V. Gerbessiotis and C. J. Siniolakis. Ordered h -level graphs on the BSP model. Submitted for journal publication, 1997.
- [94] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the BSP model with applications to parallel priority queues. Submitted for journal publication, 1997.

- [95] A. V. Gerbessiotis and C. J. Siniolakis. An experimental study of BSP sorting algorithms. Submitted for publication, 1997.
- [96] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan. PRAM programming: Theory vs. practice. Submitted for publication, 1997.
- [97] A. V. Gerbessiotis and L. G. Valiant. Personal communication, 1997.
- [98] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings of the 32-nd IEEE Annual Symposium on Foundations of Computer Science*, pp. 698–710, 1991.
- [99] M. Golin and R. Sedgewick. Analysis of a simple yet efficient convex hull algorithm. In *Proceedings of the 4-th ACM Symposium on Computational Geometry*, Illinois, pp. 153–162, 1988.
- [100] M. T. Goodrich. Finding the convex hull of a sorted point set in parallel. *Information Processing Letters*, 26:173–179, 1987.
- [101] M. T. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 26-th ACM Symposium on Theory of Computing*, May 1996.
- [102] A. Gottlieb and C. P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *Journal of the ACM*, 13(2):193–209, 1984.
- [103] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, 1989.
- [104] T. Hagerup. Hybrid-sort revisited and parallelized. *Information Processing Letters*, 32:35–39, 1989.
- [105] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.
- [106] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1990.
- [107] T. J. Harris. A survey of PRAM simulation techniques. Technical Report CSR-23-92, Department of Computer Science, The University of Edinburgh, 1992.
- [108] D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. Technical Report UMIACS-TR-96-53/CS-TR-3669, The University of Maryland Institute for Advanced Computer Studies, 1996.
- [109] D. R. Helman, J. JáJá, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. Technical Report UMIACS-TR-96-54/CS-TR-3670, The University of Maryland Institute for Advanced Computer Studies, 1996.
- [110] J. Hershberger. Finding the upper envelope of n line segments in $O(n \cdot \lg n)$ time. *Information Processing Letters*, 33:169–174, 1989.
- [111] J. M. D. Hill and D. B. Skillicorn. Lessons learned from implementing BSP. Technical Report PRG-TR-21-96, Computing Laboratory, Oxford University, November 1996.

- [112] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. *BSPlib: The BSP programming library*. Technical Report PRG-TR-16-97, Computing Laboratory, Oxford University, May 1997.
- [113] J. M. D. Hill, S. A. Jarvis, C. J. Siniolakis, and V. P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool: A case study in optimizing SQL. Technical Report PRG-TR-17-97, Computing Laboratory, Oxford University, May 1997.
- [114] J. M. D. Hill, S. A. Jarvis, C. J. Siniolakis, and V. P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. Submitted for publication, 1997.
- [115] C. A. R. Hoare. Quick-sort. *The Computer Journal*, 5:10–15, 1962.
- [116] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.
- [117] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the IEEE Computer Society's 7-th International Computer Software and Applications Conference*, pp. 627–631, 1983.
- [118] L. Hyafil. Bounds for selection. *SIAM Journal of Computing*, 5:109–114, 1976.
- [119] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [120] T. Johnson and D. Shasha. The performance of current *B*-tree algorithms. *ACM Transactions on Database Systems*, 18(1):51–101, 1993.
- [121] D. W. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, 1989.
- [122] B. H. H. Juurlink and H. A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters*, 58:303–310, 1996.
- [123] A. R. Karlin and E. Upfal. Parallel hashing – An efficient implementation of shared memory. In *Proceedings of the 18-th ACM Symposium on Theory of Computing*, pp. 160–168, 1986.
- [124] R. M. Karp, M. Saks, and A. Wigderson. On a search problem related to branch-and-bound procedures. In *Proceedings of the 27-th IEEE Symposium on Foundations of Computer Science*, pp. 19–28, 1986.
- [125] R. M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the 20-th ACM Symposium on Theory of Computing*, pp. 290–300, 1988.
- [126] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, (J. van Leeuwen, ed.), North Holland, Amsterdam, 1990.
- [127] R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765–789, 1993.
- [128] D. G. Kirkpatrick. A unified lower bound for selection and set partitioning problems. *Journal of the ACM*, 28:150–165, 1981.

- [129] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.
- [130] S. Knee. Program development and performance prediction on BSP machines using OPAL. Technical Report PRG-TR-18-94, Computing Laboratory, Oxford University, 1994.
- [131] D. E. Knuth. *The Art of Computer Programming. Volume I: Fundamental Algorithms*. Addison-Wesley, Reading, 1968.
- [132] D. E. Knuth. *The Art of Computer Programming. Volume II: Semi-numerical Algorithms*. Addison-Wesley, Reading, 1969.
- [133] D. E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching*. Addison-Wesley, Reading, 1973.
- [134] D. E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, 1976.
- [135] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for some graph problems. In *Proceedings of International Conference on Parallel Processing*, pp. 180–185, 1985.
- [136] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10):955–968, 1985.
- [137] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.
- [138] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.
- [139] Y. S. Kwong and D. Wood. A new method for concurrency in *B*-trees. *IEEE Transactions on Software Engineering*, SE-8(3):211–222, 1982.
- [140] R. E. Ladner and M. J. Fischer. Parallel-prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.
- [141] D. S. Lecomber. Object-oriented programming with BSP++. In *Proceedings of the PPECC Workshop on Parallel and Distributed Computing*, Abingdon, March 1995.
- [142] D. S. Lecomber and C. S. Siniolakis. A PRAM library: The BSP approach. Technical Report PRG-TR-18-97, Computing Laboratory, Oxford University, May 1997.
- [143] P. L. Lehman and S.B. Yao. Efficient locking for concurrent operations on *B*-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.
- [144] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, 1985.
- [145] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays – Trees – Hyper-cubes*. Morgan Kaufmann, California, 1991.
- [146] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

- [147] G. S. Lueker. A data structure for orthogonal range queries. In *Proceedings of the 19-th IEEE Symposium on Foundations of Computer Science*, pp. 28–34, 1978.
- [148] M. D. MacLaren. Internal sorting by radix plus sifting. *Journal of the ACM*, 13:404–411, 1966.
- [149] W. F. McColl. General purpose parallel computing. In *Lectures on parallel computation*, (A. Gibbons and P. Spirakis, eds.), Cambridge University Press, 1993.
- [150] W. F. McColl. Scaleable parallel computing: A grand unified theory and its practical development. In *Proceedings of IFIP World Congress*, 1:539–546, Hamburg, 1994.
- [151] W. F. McColl. An architecture independent programming model for scaleable parallel computing. *Portability and Performance for Parallel Processors*, (J. Ferrante and A. J. G. Hey, eds.), John Wiley and Sons, 1994.
- [152] W. F. McColl. BSP programming. In *Proceedings of DIMACS Workshop*, Princeton, pp. 21–35, 1994.
- [153] W. F. McColl. The BSP approach to architecture independent parallel programming. Technical Report, Computing Laboratory, Oxford University, December 1994.
- [154] W. F. McColl. Scaleable computing. *Computer Science Today: Recent Trends and Developments, Lecture Notes in Computer Science* 1000, (J. van Leeuwen, ed.), Springer-Verlag, pp. 46–61, 1995.
- [155] W. F. McColl and Q. Miller. Development of the GPL language. Technical Report, Computing Laboratory, Oxford University, October 1995.
- [156] W. F. McColl. Scaleability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12(4):265–272, 1996.
- [157] C. J. H. McDiarmid. On the method of bounded differences. *Surveys in Combinatorics: Invited Papers at the 12-th British Combinatorial Conference*, (J. Siemons, ed.), pp. 148–188, 1989.
- [158] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.
- [159] H. Meijer and S. G. Akl. The design and analysis of a new hybrid-sort algorithm. *Information Processing Letters*, 10:213–218, 1980.
- [160] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26-th IEEE Symposium on Foundations of Computer Science*, pp. 478–489, 1985.
- [161] R. Miller and Q. F. Stout. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, C-37(12):1605–1618, 1988.
- [162] R. Miller. A library for bulk-synchronous parallel programming. In *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, 1993.
- [163] R. Miller and J. L. Reed. The Oxford BSP library: User’s guide. Technical Report, Computing Laboratory, Oxford university, 1994.

- [164] C. Mohan, H. Pirahesh, W. G. Tang, and Y. Wang. Parallelism in relational database management systems. *IBM Systems Journal*, 33(2):349-371, 1994.
- [165] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1996.
- [166] J. I. Munro and P. V. Poblete. A lower bound for determining the median. Research Report CS-82-21, University of Waterloo, 1982.
- [167] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29(3):642-667, 1982.
- [168] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt. Parallel algorithms for the two-dimensional convex hull problem in two dimensions. Technical Report EE 8005, Department of Electrical Engineering, Indian Institute of Technology, October 1980.
- [169] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer Systems Science*, 23:166-204, 1981.
- [170] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceeding of ICALP'83: Automata, Languages, and Programming, Lecture Notes in Computer Science* 154, (J. Diaz, ed.), Springer-Verlag, pp. 597-609, 1983.
- [171] M. C. Pinotti and G. Pucci. Parallel priority queues. *Information Processing Letters*, 40:33-40, 1990.
- [172] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.
- [173] M. J. Quinn and P. J. Hatcher. On the utility of communication-computation overlap in data-parallel programs. *Journal of Parallel and Distributed Computing*, 33:197-204, 1996.
- [174] S. Rajasekaran and J. H. Reif. Optimal and sub-logarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18:594-607, 1989.
- [175] S. Rajasekaran. Randomized parallel selection. In *Proceedings of the 10-th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes on Computer Science* 472, Springer-Verlag, pp. 215-224, 1990.
- [176] S. Rajasekaran and S. Sen. On parallel integer-sorting. *Acta Informatica*, 29:1-15, 1992.
- [177] A. G. Ranade, S. Bhatt, and S. Johnson. The Fluent abstract machine. In *Proceedings of 5-th MIT Conference on Advanced Research in VLSI*, MIT Press, pp. 71-93, 1988.
- [178] A. G. Ranade. Maintaining dynamic ordered sets on processor networks. In *Proceedings of the 4-th ACM Symposium on Parallel Algorithms and Architectures*, pp. 127-137, 1992.
- [179] A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *Proceedings of the 6-th IEEE Symposium on Parallel and Distributed Processing*, pp. 97-103, 1994.
- [180] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, C-37(12):1657-1665, 1988.

- [181] V. N. Rao and V. Kumar. Concurrent access of priority queues. Technical Report TR88-06, Computer Science Department, University of Texas at Austin, February 1988.
- [182] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In *Synthesis of Parallel Algorithms*, (J. H. Reif, ed.), Morgan Kaufmann, 1993.
- [183] H. J. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60–76, January 1987.
- [184] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, San Mateo, 1993.
- [185] J. H. Reif and S. Sen. Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications. *SIAM Journal on Computing*, 23(3):633–651, 1994.
- [186] G. Reinelt. The traveling salesman: Computational solutions for TSP applications. *Lecture Notes in Computer Science* 840, (G. Goos and J. Hartmanis, eds.), Springer-Verlag, pp. 40–41, 1994.
- [187] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396–409, 1985.
- [188] B. Samadi. B-trees in a system with multiple users. *Information Processing Letters*, 5(4):107–112, 1976.
- [189] P. Sanders. Fast priority queues for parallel branch-and-bound. In *Proceedings of the 2-nd International Workshop Irregular'95*, *Lecture Notes in Computer Science* 980, Springer-Verlag, pp. 379–393, 1995.
- [190] J. B. Saxe and J.L.Bentley. Transforming static data structures into dynamic structures. In *Proceedings of the 20-th Annual IEEE Symposium on Foundations of Computer Science*, pp. 148–168, 1979.
- [191] A. Schonhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13:184–199, 1976.
- [192] R. Sedgewick. *Algorithms*. Addison-Wesley, 1983.
- [193] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.
- [194] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:362–372, 1992.
- [195] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.
- [196] C. J. Siniolakis. On the complexity of BSP sorting. Technical Report PRG-TR-09-96, Computing Laboratory, Oxford University, May 1996.
- [197] C. J. Siniolakis. Direct bulk-synchronous parallel algorithms in computational geometry. Technical Report PRG-TR-10-96, Computing Laboratory, Oxford University, May 1996. Also as *Qualifying Dissertation to Support an Application for Transfer of Status from a Probationary Research Student to a Student for the Degree of Doctor of Philosophy at the University of Oxford*, September 1995.

- [198] C. J. Siniolakis and K. R. Sujithan. BSP database primitives. Technical Report PRG-TR-19-97, Computing Laboratory, Oxford University, May 1997.
- [199] C. J. Siniolakis. Static and dynamic hashing strategies on the BSP model. Technical Report PRG-TR-25-97, Computing Laboratory, Oxford University, July 1997.
- [200] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. To appear in *Journal of Scientific Programming*, 1997.
- [201] K. R. Sujithan and J. M. D. Hill. Collection types for database programming in the BSP model. In *5-th Euromicro Workshop on Parallel and Distributed Processing*, 1997.
- [202] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983.
- [203] G. T. Toussaint. *Computational Geometry*. Elsevier Science Publishers B.V., North Holland, 1985.
- [204] L. G. Valiant. Bulk-synchronous parallel computers. In *Parallel Processing and Artificial Intelligence*, (M. Reeve and S. E. Zenith, eds.), John Wiley and Sons, 1989.
- [205] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science*, (J. van Leeuwen, ed.), North Holland, Amsterdam, 1990.
- [206] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.
- [207] L. G. Valiant. A combining mechanism for parallel computers. In *Proceedings of the First Heinz Nixdorf Symposium on Parallel Architectures and Their Efficient Use, Lecture Notes in Computer Science* 678, (F. Meyer auf der Heide, B. Monien, and L. Rosenberg, eds.), Springer-Verlag, pp. 1–10, 1993.
- [208] U. Vishkin. An optimal parallel algorithm for selection. *Advances in Computing Research*, JAI Press Inc., Greenwich, 1987.
- [209] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume I*. Computer Science Press, Rockville, Maryland, 1988.
- [210] D. E. Willard. Predicate-oriented database search algorithms. Technical Report TR-20-78, Aiken Computation Laboratory, Harvard University, 1978.
- [211] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347–348, 1964.
- [212] J. C. Wyllie. The complexity of parallel computation. Technical Report TR 79-387, Department of Computer Science, Cornell University, New York, 1979.

