

Two Approaches
to
Architecture - Independent
Parallel Computation

Richard Miller
Lady Margaret Hall
Michaelmas Term 1994



Thesis submitted for the degree of Doctor of Philosophy

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD

Two Approaches to Architecture-Independent Parallel Computation

Richard Miller
Lady Margaret Hall, Oxford

D.Phil. Thesis
Michaelmas Term, 1994

Abstract

Two approaches to architecture-independent parallel computation are investigated: a constructive functional notation for specifying implicitly parallel operations on multidimensional arrays, and an extension to imperative sequential programming languages for implementing bulk-synchronous parallel algorithms.

An algebra of multidimensional rectangular arrays is defined constructively, by means of an injective singleton operator which maps each value from a base type into a one-element array, and a set of join operators which map a pair of arrays into their concatenation along one of a set of dimensions. A repertoire of array operations is defined in the context of the Bird-Meertens Formalism, using array versions of polymorphic higher-order functions such as *map*, *reduce*, *zip* and *cross*. This approach gives rise to a collection of algebraic laws which can be used to guide the transformation of array expressions into different equivalent forms. In particular, the promotion laws have a natural interpretation as descriptions of different parallel realisations of an array computation. The use of the array algebra is illustrated by the derivation of two example algorithms: the L-U decomposition of a matrix, and the numerical solution of an elliptic partial differential equation.

The bulk-synchronous parallel model of an abstract general-purpose parallel computer is described, along with several variants of the BSP cost model. A refinement to the cost model is proposed, introducing a ‘half-bandwidth’ parameter to quantify the effect of data granularity on communication cost. A simple BSP programming model is defined, with semantics specified in CSP. The programming model is realised by extending sequential programming languages with a small set of primitives for process creation and termination, bulk synchronisation, and inter-process data access. The author has created implementations of these primitives in the Oxford BSP library, with versions for a variety of parallel systems including networked workstations, shared-memory multiprocessors and massively-parallel distributed-memory machines, and has used them to produce an architecture-independent parallel version of the molecular dynamics module of the UCSF AMBER 4.0 package.

Acknowledgements

The research reported here owes much to the inspiration, influence and ideas of many colleagues past and present. Seeds for the two themes of computing with arrays and architecture-independent systems programming were planted by Juris Reinfelds, who introduced me in 1973 to the expressive power of higher-dimensional arrays for scientific computation in the SIGMA programming language, and who encouraged and supported me in 1977 in the project (considered preposterous at the time) of ‘porting’ the UNIX operating system from the PDP-11 to the Interdata 7/32. My interest in the theory of parallel computation was kindled by Tony Hoare’s course of lectures on CSP in Santa Cruz in 1979; and Bill McColl later provided an opportunity for practical parallel programming with transputers at Oxford, and opened my eyes to the riches hidden within the BSP model. Gaétan Hains and Lenore Mullen set a valuable example in MoA of a formal semantics for arrays of arbitrary dimension; and David Skillicorn showed how to use the Bird-Meertens formalism as a model of parallel computation.

I have relied extensively on Richard Bird’s calculus of constructive functional programming and in particular his *above-beside* model of two-dimensional arrays, and on Leslie Valiant’s theory of bulk-synchronous parallel computation. Without their theoretical foundations the present work would not have been possible.

I am grateful to Paul Wesson, Richard Burroughs and Joy Reed, the pioneers who encouraged my development of the BSP library by using it in their own work. Thanks to other colleagues, friends and relations in the Computing Laboratory, for the stimulating exchange of ideas and viewpoints which enriches academic life; to my supervisors, Bernard Sufrin and David Mayers, for guidance and advice; and to Chris Edge for his unfailing encouragement.

Above all, thanks to Felicity for her love and patience through it all.

Part of this work has been assisted financially by a postgraduate research studentship (90312030) from the Science and Engineering Research Council. Parallel computing facilities for the development of the BSP library and parallel AMBER have been generously provided by Oxford Parallel (funded by the DTI and SERC) and by Cray Research, Inc.

In memoriam Frank Dunstan Miller, 1915-1990

Contents

1	Introduction	3
2	Multidimensional arrays	5
2.1	Constructive functional programming	5
2.2	Generalising <i>above</i> and <i>beside</i>	8
2.3	Formal definition of array algebra	9
2.4	Size and shape of arrays	12
3	Array homomorphisms	16
3.1	Map and reduce	16
3.2	Algebraic laws for homomorphisms	18
3.3	Flatten and transpose	20
3.4	Zip	22
3.5	Cross-product	24
4	Subarrays	26
4.1	Vectors and subarrays	26
4.2	Matrix algebra	29
4.3	Dot product and matrix multiplication	29
4.4	Parallel matrix multiplication	31
4.5	Directed reductions	32
4.6	Accumulations	34
5	Programming with arrays	35
5.1	Deriving an algorithm for LU decomposition	35
5.2	Partial differential equations	38
6	Bulk-synchronous parallel computation	49
6.1	The BSP computation model	49
6.2	BSP cost models	52
7	A BSP programming model	58

7.1	Design choices	58
7.2	Semantics of the programming model	60
8	The Oxford BSP library	66
8.1	Routines in the library	66
8.2	BSP library implementations	70
8.3	Library performance	72
9	Parallel molecular mechanics	75
9.1	Software acceleration	75
9.2	AMBER molecular mechanics	76
9.3	Execution profiling	76
9.4	Force field evaluation on transputers	78
9.5	Bulk-synchronous parallel AMBER	78
9.6	Parallel performance	80
10	Conclusions and future work	82
10.1	Further development of the array theory	82
10.2	Further development of the BSP library	84
10.3	Integrating the two approaches	84
A	The Oxford BSP Library Reference Manual	86
A.1	bsp(1)	87
A.2	bspcc(1)	89
A.3	bspf77(1)	90
A.4	bspprof(1)	91
A.5	bsp(3)	92
A.6	bsp_broadcast(3)	96
A.7	bsp_reduce(3)	97
A.8	bsp_time(3)	98
	Bibliography	99

Chapter 1

Introduction

The lack of a single unifying model of parallel computation has been a serious obstacle to the widespread adoption of parallel computers for general purpose programming. The growing diversity of parallel computer architectures has been accompanied by a proliferation of mutually incompatible programming models and language extensions. This means that the development of parallel applications programs requires a commitment to a particular hardware and software architecture. The more a program is tuned for high performance on a particular parallel system, the more extensive rewriting will be required to adapt it to another environment.

It is unreasonable for applications programmers to have to perform feats of architecture-specific systems programming to make their programs run efficiently on parallel machines. This is especially true in the case of scientific computation, where the need for parallelism is not inherent in the application, but is simply a means towards higher performance. Much work has been expended on the development of tools for automatic or computer-assisted restructuring of sequential programs to efficient parallel form, but this approach is hindered by a second obstacle. Because floating-point arithmetic abounds with non-associative operations, any reordering of a computation is likely to produce a result which is at least slightly different. Thus, parallel versions of a sequential program are not formal equivalents or even refinements of the original one; they must rather be viewed as variations on a theme. To decide whether a variation is acceptable requires a deep understanding of the meaning of a program, as well as its syntax and semantics, to distinguish where the stated order of sub-computations is essential to the algorithm, and where it is merely an artifact of the requirement that statements must be written in some arbitrary order.

The research reported in this thesis has been addressed towards overcoming these obstacles. Rather than writing sequential programs in the hope of transforming

them into parallel ones, or writing explicitly parallel programs for a particular architectural model in the fear of rewriting them repeatedly as standards for hardware and software evolve, I have sought a way to express algorithms in an abstract form which has no bias towards either sequential or parallel interpretation, explicitly identifying only the sequencing which is dictated by algorithmic dependencies. As a step in this direction, I have generalised from the constructive Theory of Lists to produce an algebra of multidimensional arrays, and shown that it can be used to describe some standard numerical algorithms in a way which exposes the potential for implicit parallelism.

If an implicitly parallel approach is to be useful for practical applications, there must eventually be a way to translate algorithm specifications into an explicitly parallel executable form. To avoid the multiplication of effort required in producing versions for each type of parallel machine, any transformation or compilation scheme should be based on an architecture-independent computation model. However the model must be realistic enough to allow as much as possible of the effective power of the underlying machine to be exploited.

I believe that the bulk-synchronous parallel model of computation offers just the right combination of generality and efficiency. The BSP theory provides a machine-independent way to specify explicitly parallel algorithms, whose performance in different environments can be predicted as a function of a small number of architectural parameters. As a step towards using the BSP machine as a target for the transformation of high-level implicitly parallel algorithms for execution on machines with a variety of architectures, I have formally specified a set of primitive BSP operations for process creation, remote data access and bulk synchronisation, and built an implementation in the form of a subroutine library callable from standard sequential languages such as FORTRAN, Pascal or C.

Although I first envisioned the BSP library as part of the infrastructure for the compilation or transformation of a (yet unspecified) high-level notation, it has taken on a life of its own as a tool for the direct implementation of portable parallel programs. The library has been used with encouraging success in a number of projects, in developing new parallel programs and in parallelising existing large applications. One such project, the creation of an architecture-independent parallel version of AMBER molecular dynamics, is described in the present thesis.

Chapter 2

Multidimensional arrays

2.1 Constructive functional programming

The task of transforming explicitly sequential programs into explicitly parallel form is made difficult by the inherent *sequential bias* of traditional imperative programming languages. In such languages, statements must be written one after another in an ordering which implies a sequential order of execution, even when the need for a particular sequence is not inherent in the algorithm.

Adopting a declarative notation, and writing programs as a set of unordered function declarations, would seem to provide a way to describe algorithms without over-specification of sequencing. But even functional programming languages are given a sequential bias by their free-algebra style of constructing aggregate datatypes. For example, in a language like Orwell [Wadler and Miller, 1990], with lists constructed using *nil* and *cons*, it is possible to define *foldl* and *foldr* functions, which insert a binary operator between adjacent elements of a list, parenthesising respectively from the left and from the right; but it is impossible to define a general *fold* which allows the expression to be parenthesised in any order.

To reduce this sequential bias, we turn to the *constructive* style of functional programming known as the Bird-Meertens formalism [Meertens, 1986, Bird, 1990], where constructors with algebraic laws (such as associative and commutative properties) allow aggregate datatypes to be built from their components in more than one way. For example, the required sequential-bias-free *fold* operator is a *reduction*, defined on non-empty lists by the equations

$$\begin{aligned}\oplus/[a] &= a \\ \oplus/(x \uplus y) &= (\oplus/x) \oplus (\oplus/y)\end{aligned}$$

When evaluating $\oplus/z$ for an arbitrary non-singleton list z , an implementation can apply the $\oplus$ operators in any order, corresponding to the different ways z can be ‘parsed’ as $(x \# y)$. Provided that $\oplus$ is associative, any ordering of the sub-computations will give the same result.

A notation based on Bird-Meertens-style datatypes should allow us to describe computations with maximum (potential) parallelism, by specifying only the sequencing which is required for the stability (one hesitates to use ‘correctness’ in connection with floating-point arithmetic) of the algorithm. An implementation (whether by compiler or human program-transformer) then has the maximum freedom to reorder or parallelise operations for efficiency on a given sequential or parallel architecture, while staying within the constraints of the programmer’s specification.

In [Skillicorn, 1990], Skillicorn argues that the Bird-Meertens theory of lists can serve as a universal model of parallel computation. For practical programming, however, it is useful to have a richer set of data structures than lists alone. In particular, numerical and scientific algorithms make much use of two- and three-dimensional arrays. Although these can be simulated with nested lists — the traditional approach in functional programming languages — the resulting asymmetry between the dimensions introduces a new form of sequential bias. Thus we seek to develop a constructive theory of multi-dimensional arrays, which generalises the two-dimensional array model of [Bird, 1988].

2.1.1 Notation and conventions

The notation in this paper follows the conventions of the Bird-Meertens formalism.

Functional notation Function application is denoted by juxtaposition, and associates to the left. Functions of two arguments are generally denoted by infix operators, which have a lower precedence in expressions than prefix function application, and associate to the right.

A binary operator can be *sectioned* by placing it in parentheses with one operand, to make a unary function, or *curried* by placing it in parentheses on its own, to make a second-order function. If $\oplus$ is any binary operator, then

$$\begin{aligned} (a \oplus) b &= a \oplus b \\ (\oplus b) a &= a \oplus b \\ (\oplus) a b &= a \oplus b \end{aligned}$$

In an abuse of notation, we will often omit the parentheses where context or type information makes it obvious that a section or currying is required.

In stating laws such as the definitions of sectioning above, all free variables are implicitly universally quantified over the appropriate type. We will use the following conventional types for variables:

$A\ B$	–	types or sets
$f\ g\ h$	–	unary functions
$\oplus\ \otimes$	–	binary operators
$u\ v\ w\ x\ y\ z$	–	arrays
$a\ b\ c\ d$	–	scalars (values from the base type)
$i\ j\ k\ l$	–	dimensions
$m\ n$	–	natural numbers

Some useful functions Function composition is denoted by an infix $\cdot$ operator, and has lower precedence than any other infix operator.

$$(f \cdot g)\ x = f(g\ x)$$

We denote by x° the constant function which maps everything to the value x .

$$(x^\circ)\ y = x$$

The *swap* operator $\sim$ causes its (binary operator) argument to take its operands in reverse order.

$$x \tilde{\oplus} y = y \oplus x$$

The $\times$ operator applies a pair of functions to a pair of arguments to give a pair of results.

$$(f \times g)(x, y) = (f\ x, g\ y)$$

The $\ll$ and $\gg$ operators return their left and right argument respectively.

$$\begin{aligned} x \ll y &= x \\ x \gg y &= y \end{aligned}$$

2.2 Generalising *above* and *beside*

In his series of lectures on Constructive Functional Programming [Bird, 1988], Richard Bird introduces a theory of two-dimensional arrays. In Bird's theory, an array is either a *singleton* (a 1×1 array containing a single element), or is formed by joining two smaller arrays together vertically or horizontally using the constructors *above* and *beside* (denoted by infix operators $\ominus$ and $\oplus$).

In order that all arrays be rectangular, the binary constructors are defined to be partial functions: arrays can be joined above one another only if they have the same width, and beside one another only if they have the same height. Both constructors are associative, and they are related by an interchange law, which Bird calls the *abiding* property. We say that $\ominus$ abides with $\oplus$, meaning that the identity

$$(x \oplus u) \ominus (y \oplus v) = (x \ominus y) \oplus (u \ominus v)$$

holds whenever all the parenthesised terms are defined: that is, when x has the same height as u and the same width as y , and v has the same height as y and the same width as u .

The associativity and abiding laws mean that an array has no memory of the way it was constructed: any two arrays having the same shape (equal width and equal height) and equal elements at each position are identified as equal. For example, we can show that

$$(|1| \ominus |4|) \oplus ((|2| \oplus |3|) \ominus (|5| \oplus |6|)) = (((|1| \oplus |2|) \oplus |3|) \ominus ((|4| \oplus |5|) \oplus |6|))$$

(where $|a|$ denotes the singleton array containing element a). Both sides of the equation denote the same array, which we can picture geometrically as

1	2	3
4	5	6

This two-dimensional array algebra has been explored further by Chris Wright [Wright, 1988]. It can be seen as a two-dimensional generalisation of the algebra of join-lists [Bird, 1987], with the associative constructor $\#$ replaced by the two abiding constructors $\ominus$ and $\oplus$. The subspace consisting of arrays of height 1 (*row vectors*) is isomorphic to the space of non-empty lists: the isomorphism in question maps the singleton array constructor to the singleton list constructor, and $\oplus$ to $\#$. A similar isomorphism holds with the subspace of arrays of width 1 (*column vectors*).

The relationship between list and array theories is strongly suggested by Bird's terminology: after defining the algebra of lists as the canonical instance of a monoid

(an algebra having a single associative binary operator), he coins the term *binoid* to describe an algebra such as that of arrays, with two associative and abiding operators.

To carry the generalisation further from two-dimensional to n -dimensional arrays, we replace $\oplus$ and $\otimes$ with n pairwise-abiding *join* constructors. Rather than inventing n different symbols for the constructors, we give them names of the form $\#_i$ where the index i is an element from a set of dimensions $\mathcal{D}$.

It is conventional for multidimensional array models to impose a hierarchy or ordering on dimensions. For example, in [Jeuring, 1992] an n -dimensional array is defined to be a list of $(n-1)$ -dimensional arrays; while in [G.Hains and Mullin, 1991] and [Banger, 1993] an array is defined in terms of a one-dimensional contents list containing the array's elements in row-major order. This asymmetry of dimensions is not strictly necessary; we will take a more abstract view and require only that $\mathcal{D}$ be a finite set on which an equality relation is defined. For example, taking $\mathcal{D} = \{\text{row}, \text{col}\}$ and abbreviating $\oplus = \#_{\text{row}}$ and $\otimes = \#_{\text{col}}$ gives the two-dimensional arrays of [Bird, 1988]. For a three-dimensional application, we might want to denote the constructors by $\#_x$, $\#_y$ and $\#_z$ to correspond to the axes in Euclidean space, defining $\mathcal{D} = \{x, y, z\}$.

To avoid the complication of dealing simultaneously with arrays of differing dimensionality, the set $\mathcal{D}$ will be regarded as a global constant of the theory, and seldom mentioned explicitly. Thus all arrays are treated as having the same fixed number of dimensions. This choice is justified by the implied spatial model of arrays: if we interpret dimensions geometrically as orthogonal axes in Cartesian hyperspace, we observe that while an $n \times 1$ array is a different shape from a $1 \times n$ array (because they are oriented in different directions), a $1 \times 1 \times \cdots \times 1 \times m \times n$ array ought to be indistinguishable from an $m \times n$ array.

2.3 Formal definition of array algebra

Partial functions and equality When dealing with partial functions such as the array join constructors, we must take some care over undefined terms. In particular, in order to maintain Leibnitz' principle ("equals may be substituted for equals"), an equation of the form

$$f\ x = g\ x$$

where f and g may be partial, must be interpreted to mean "either both sides denote the same value, or both sides are undefined". We cannot deduce from it

that a value $f x$ actually exists. For that purpose we introduce the relation of *strong equality*: the equation

$$f x \simeq g x$$

asserts that both sides are defined and denote equal values. Similarly, we use the symbol $\in$ to denote *strong membership* in a datatype or set: the assertion

$$x \in T$$

is interpreted to mean both that x is a term of type T and that it denotes a well-defined value.

Array axioms We denote by A^* the type of multidimensional arrays with elements from the *base type* A . (Note that A itself may be an array type, so that arrays can be nested; however, all elements of a single array must be of the same type.) To be pedantic, we should write $A_{\mathcal{D}}^*$, since the array datatype is parameterised by the set of dimensions; but since the same $\mathcal{D}$ pervades the theory, we will leave it implicit.

The type A^* is defined formally by a set of axioms, which we present as inference rules. The *introduction* rules say that an array value can be constructed by applying the singleton constructor $[]$ to a value of the base type, or by joining two singleton arrays along any dimension i with the constructor $\#_i$. (The notation for the constructors is chosen to remind us of the analogy with nonempty lists.)

$$\frac{a \in A}{[a] \in A^*} \quad [\text{singleton introduction}]$$

$$\frac{\begin{array}{l} a, b \in A \\ i \in \mathcal{D} \end{array}}{[a] \#_i [b] \in A^*} \quad [\text{join introduction}]$$

Note that the use of the symbol $\in$ implies that the constructors are *strict*: $[a]$ and $[a] \#_i [b]$ are asserted to be well-defined only if a and b are.

Joining of non-singleton arrays is constrained by a generalised rectangularity condition: the expression $x \#_i y$ is defined only if x and y have the same size in every dimension other than i . Rather than making this condition explicit we will prove

later that it can be derived from the following *congruence* rules, which state that the join constructors are associative and pairwise abiding:

$$\begin{array}{c}
 i \in \mathcal{D} \\
 (x \mathbin{+}_i y), (y \mathbin{+}_i z) \in A^* \\
 \hline
 (x \mathbin{+}_i y) \mathbin{+}_i z \simeq x \mathbin{+}_i (y \mathbin{+}_i z) \quad [\text{join associativity}]
 \end{array}$$

$$\begin{array}{c}
 i, j \in \mathcal{D} \\
 i \neq j \\
 (x \mathbin{+}_i y), (u \mathbin{+}_i v), (x \mathbin{+}_j u), (y \mathbin{+}_j v) \in A^* \\
 \hline
 (x \mathbin{+}_i y) \mathbin{+}_j (u \mathbin{+}_i v) \simeq (x \mathbin{+}_j u) \mathbin{+}_i (y \mathbin{+}_j v) \quad [\text{join abiding}]
 \end{array}$$

The above axioms give sufficient conditions for a value to be in A^* , and for two array values to be equal. We complete the definition of the type by stating that they are also necessary conditions. One way to do this would be via a *closure* or *induction* rule (any subset of A^* containing all the singletons and closed under every join constructor is equal to A^* itself), and an *individuality* rule (two array terms can only be equal if the same constructor is applied to equal operands, or by one of the two congruence rules). These rules can be summarised as “no junk, no confusion”.

Instead, we will declare A^* to be the *initial algebra* satisfying the first four axioms. By definition, this means that for every algebra $(B, f, \oplus)$ with the same structure as $(A^*, [-], \mathbin{+})$, there is a unique total function $h : A^* \rightarrow B$ such that

$$\begin{aligned}
 h[a] &= f a \\
 h(x \mathbin{+}_i y) &= h x \oplus_i h y \quad \dashv (x \mathbin{+}_i y) \in A^*
 \end{aligned}$$

A function satisfying these two equations for some f and $\oplus$ is called an *array homomorphism*; it is ‘structure-preserving’ in the sense that it replaces instances of $[-]$ with f , and replaces each $\mathbin{+}_i$ with the corresponding $\oplus_i$.

The turnstile symbol $\dashv$ is used above to indicate that the equation on the left holds when the condition on the right is satisfied. The qualified equation states that the functions $h \cdot \mathbin{+}_i$ and $\oplus_i \cdot (h \times h)$ are equal on the domain of $\mathbin{+}_i$.

We formalise the initiality property with the *Unique Extension* axiom:

$$\begin{array}{l}
 a \in A \vdash f a \in B \\
 a, b \in A \vdash (f a \oplus_i f b) \in B \\
 (p \oplus_i q), (q \oplus_i r) \in B \vdash \\
 \quad (p \oplus_i q) \oplus_i r \simeq p \oplus_i (q \oplus_i r) \\
 (p \oplus_i q), (r \oplus_i s), (p \oplus_j r), (q \oplus_j s) \in B \vdash \\
 \quad (p \oplus_i q) \oplus_j (r \oplus_i s) \simeq (p \oplus_j r) \oplus_i (q \oplus_j s) \\
 h \cdot [-] = g \cdot [-] = f \\
 (x \#_i y) \in A^* \vdash h(x \#_i y) = h x \oplus_i h y \\
 (x \#_i y) \in A^* \vdash g(x \#_i y) = g x \oplus_i g y \\
 \hline
 x \in A^* \vdash h x \simeq g x \quad \text{[unique extension]}
 \end{array}$$

The Unique Extension rule can be interpreted as follows: Let f be a total function from A to a type B , and for each dimension $i \in \mathcal{D}$ let $\oplus_i$ be a (possibly partial) binary operator on B . Suppose all the $\oplus_i$ are defined on all values in the range of f , and are associative and pairwise abiding. If functions h and g are both defined as homomorphisms with respect to f and $\oplus$, then the functions are equal, and defined for all values in A^* ; that is, the homomorphism is total and unique.

As a shorthand, the homomorphism defined above can be denoted by $\mathcal{H}(f, \oplus)$; we view $\mathcal{H}$ as a higher-order function which acts on the pair $(f, \oplus)$ to generate a function h which satisfies the equations

$$\begin{aligned}
 h \cdot [-] &= f \\
 h \cdot \#_i &\sqsubseteq \oplus_i \cdot (h \times h)
 \end{aligned}$$

The *refinement* symbol $\sqsubseteq$ indicates that whenever the left-hand side is defined, the right-hand side is defined and equal to it. The antecedents of the Unique Extension rule give the conditions that f and $\oplus$ must satisfy in order for a unique and total solution h to exist.

2.4 Size and shape of arrays

Size For any $i \in \mathcal{D}$, the function $\#_i$ gives the size of an array along dimension i . It is defined by the equations

$$\begin{aligned}
 \#_i[a] &= 1 \\
 \#_i(x \#_i y) &= \#_i x + \#_i y \\
 \#_i(x \#_j y) &= \#_j x \bullet \#_j y \quad \neg j \neq i
 \end{aligned}$$

where $\bullet$ is the partial binary operator defined by

$$m \bullet n = m \dashv m = n$$

In the context of two-dimensional arrays, we can use the abbreviations

$$\begin{aligned} width &= \#_{ROW} \\ height &= \#_{COL} \end{aligned}$$

Theorem $\#_i$ is a total function from A^* to $\mathbf{N}^+$ (positive integers).

Proof:

$\#_i$ is the homomorphism $\mathcal{H}(1^\circ, \oplus)$, where n° is the constant function which maps everything to the value n , and

$$\begin{aligned} \oplus_j &= (+) \quad , \text{if } j = i \\ &= (\bullet) \quad , \text{otherwise} \end{aligned}$$

To show that the conditions of the elimination rule are satisfied:

1. $a \in A \Rightarrow 1^\circ a = 1 \in \mathbf{N}^+$
2. $a, b \in A \Rightarrow 1^\circ a + 1^\circ b = 2 \in \mathbf{N}^+$
3. $a, b \in A \Rightarrow 1^\circ a \bullet 1^\circ b = 1 \in \mathbf{N}^+$
4. $+$ and $\bullet$ are both associative
5. $+$ abides with $\bullet$, since $(p \bullet q) + (r \bullet s) = (p + r) \bullet (q + s)$ provided $p = q$ and $r = s$

Rectangularity The formal definition of array size allows us to state and prove the rectangularity condition for the join constructors.

Theorem Let $\approx_i$ be the join conformance relation defined by

$$x \approx_i y \iff \forall j (j \neq i \Rightarrow \#_j x = \#_j y)$$

For any arrays x and y , the expression $(x \#_i y)$ is defined if and only if $x \approx_i y$.

Proof:

For $i \in \mathcal{D}$, let the operator $\oplus_i$ be defined by

$$x \oplus_i y = x \uplus_i y \dashv (x \uplus_i y) \in A^* \wedge x \approx_i y$$

That is, $\oplus_i$ agrees with $\uplus_i$ but is defined only on conformant pairs. Clearly, $\oplus_i$ is defined on singletons (because $\#_j [a] = 1$). Assuming $(x \approx_i y) \wedge (y \approx_i z)$, we have for $j \neq i$

$$\begin{aligned} & \#_j(x \oplus_i y) \\ = & \{\text{defn } \oplus\} \\ & \#_j(x \uplus_i y) \\ = & \{\text{defn } \#\} \\ & \#_j x \bullet \#_j y \\ = & \{x \approx_i y\} \\ & \#_j y \\ = & \{y \approx_i z\} \\ & \#_j z \end{aligned}$$

so that $(x \oplus_i y) \approx_i z$; and similarly $x \approx_i (y \oplus_i z)$. Assuming for $j \neq i$ that $(x \approx_i y) \wedge (u \approx_i v) \wedge (x \approx_j u) \wedge (y \approx_j v)$, we have

$$\begin{aligned} & \#_i(x \oplus_i y) \\ = & \{\text{defn } \oplus, \#\} \\ & \#_i x + \#_i y \\ = & \{(x \approx_j u) \wedge (y \approx_j v)\} \\ & \#_i u + \#_i v \\ = & \{\text{defn } \#, \oplus\} \\ & \#_i(u \oplus_i v) \end{aligned}$$

and for any k with $k \neq i \wedge k \neq j$, we have

$$\begin{aligned} & \#_k(x \oplus_i y) \\ = & \{\text{defn } \oplus, \#\} \\ & \#_k x \\ = & \{x \approx_i u\} \\ & \#_k u \\ = & \{\text{defn } \oplus, \#\} \\ & \#_k(u \oplus_i v) \end{aligned}$$

so that $(x \oplus_i y) \approx_j (u \oplus_i v)$; and similarly $(x \oplus_j u) \approx_i (y \oplus_j v)$. Thus the $\oplus_i$ are associative and pairwise abiding, so that the homomorphism $\mathcal{H}([\cdot], \oplus)$ is defined on the whole of A^* , and is equal to the *identity* homomorphism $\mathcal{H}([\cdot], \#)$.

Shape We can view $\#$ without a subscript as a higher-order operator, which maps a dimension to a specific size function such as *height* or *width*. If we swap the arguments, we obtain the *shape* operator, which we denote (following APL tradition [Iverson, 1962]) as ρ :

$$\rho x \ i \ = \ \#_i x$$

If x is an array, the section (ρx) is a finite mapping from each dimension i to the size of x along that dimension. Two arrays x and y have identical shape if and only if $\rho x = \rho y$.

Chapter 3

Array homomorphisms

3.1 Map and reduce

Map Any scalar function on the base type A can be lifted to act elementwise on arrays of A^* by the higher-order function *map*, denoted by an infix $*$. For example,

$$\text{if } x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \text{ then } f * x = \begin{bmatrix} f\ 1 & f\ 2 & f\ 3 \\ f\ 4 & f\ 5 & f\ 6 \end{bmatrix}.$$

Formally, map can be defined as the solution to the equations

$$\begin{aligned} f * [a] &= [f\ a] \\ f * (x \mathrel{+}_i y) &= (f * x) \mathrel{+}_i (f * y) \end{aligned}$$

that is, $(f*)$ is the homomorphism $\mathcal{H}([-] \cdot f, \mathrel{+})$.

Map obeys the following algebraic identities:

Map size rule: applying a function element-wise does not change the shape of an array.

$$\#_i \cdot f * = \#_i$$

Map composition rule: map distributes left through function composition; *i.e.* mapping with the composition of two functions is the same as composing the two maps.

$$(f \cdot g) * = f * \cdot g *$$

Reduce One way to lift a binary scalar operator to act on arrays is to insert the operator between adjacent elements of the array, thus collapsing or *reducing*

the array to a scalar result. We denote a reduction with a single operator by an infix $\not\!/\!.$ So that the result does not depend on the order in which we combine the elements, we require that the binary operator be associative and abide with itself.

For example, if $x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then

$$\begin{aligned} \oplus \not\!/\! x &= (1 \oplus 2 \oplus 3) \oplus (4 \oplus 5 \oplus 6) \oplus (7 \oplus 8 \oplus 9) \\ &= (1 \oplus 4 \oplus 7) \oplus (2 \oplus 5 \oplus 8) \oplus (3 \oplus 6 \oplus 9) \\ &= (((1 \oplus 2) \oplus (4 \oplus 5)) \oplus (3 \oplus 6)) \oplus (7 \oplus 8 \oplus 9) \\ &= \dots \end{aligned}$$

With the above restrictions on $\oplus$, we can define $\oplus \not\!/\!.$ as the homomorphism $\mathcal{H}(id, \oplus^\circ)$; i.e. the unique solution to

$$\begin{aligned} \oplus \not\!/\! [a] &= a \\ \oplus \not\!/\! (x \#_i y) &= (\oplus \not\!/\! x) \oplus (\oplus \not\!/\! y) \end{aligned}$$

Some examples of array reductions:

- $+\not\!/\!$ and $\times \not\!/\!$ the sum and product of elements in a numeric array
- $\uparrow \not\!/\!$ and $\downarrow \not\!/\!$ the the maximum and minimum elements of an array
- $\wedge \not\!/\!$ and $\vee \not\!/\!$ the conjunction (*all*) and disjunction (*any*) of elements in a Boolean array

General reduction The function $\not\!/\!$ performs a *uniform* reduction: the same binary operator is applied along all dimensions of an array. A more general reduction operator, denoted $/$, allows the use of a different operator for each dimension. In the expression $\oplus \not\!/\! x$, $\oplus$ is a higher-order function, providing a mapping from each dimension to a binary operator to be used along that dimension. Thus the type of $/$ is $(\mathcal{D} \rightarrow ((A \times A) \rightarrow A)) \rightarrow A^* \rightarrow A$.

We define $/$ to be the solution to

$$\begin{aligned} \oplus \not\!/\! [a] &= a \\ \oplus \not\!/\! (x \#_i y) &= (\oplus \not\!/\! x) \oplus_i (\oplus \not\!/\! y) \end{aligned}$$

that is, the homomorphism $\mathcal{H}(id, \oplus)$. To avoid ambiguity, the $\oplus_i$ are required to be associative and pairwise abiding.

Note that uniform reduction is just a special case of general reduction, since

$$\oplus \not\!/\! = \oplus^\circ \not\!/\!$$

3.2 Algebraic laws for homomorphisms

The promotion theorem for array homomorphisms In [Malcolm, 1988], a method is given for constructing a promotion theorem for any data type. For multidimensional arrays, the promotion theorem takes the following form;

Theorem *If $\mathcal{H}(f, \oplus)$ is an array homomorphism, and g a function such that for all $i \in \mathcal{D}$*

$$g(x \oplus_i y) = g x \otimes_i g y$$

then the composition of the two functions is also a homomorphism:

$$g \cdot \mathcal{H}(f, \oplus) = \mathcal{H}(g \cdot f, \otimes)$$

Proof:

We show that both sides have the same result on singleton arguments:

$$\begin{aligned} & (g \cdot \mathcal{H}(f, \oplus)) [a] \\ = & \{ \text{defn } \cdot \} \\ & g(\mathcal{H}(f, \oplus) [a]) \\ = & \{ \text{defn } \mathcal{H} \} \\ & g(f [a]) \\ = & \{ \text{defn } \cdot \} \\ & (g \cdot f) [a] \\ = & \{ \text{defn } \mathcal{H} \} \\ & \mathcal{H}(g \cdot f, \otimes) [a] \end{aligned}$$

We also have for any $i \in \mathcal{D}$ and $(x \uplus_i y) \in A^*$

$$\begin{aligned} & (g \cdot \mathcal{H}(f, \oplus))(x \uplus_i y) \\ = & \{ \text{defn } \cdot \} \\ & g(\mathcal{H}(f, \oplus)(x \uplus_i y)) \\ = & \{ \text{defn } \mathcal{H} \} \\ & g(\mathcal{H}(f, \oplus)x \oplus_i \mathcal{H}(f, \oplus)y) \\ = & \{ \text{hypothesis on } g \} \\ & g(\mathcal{H}(f, \oplus)x) \otimes_i g(\mathcal{H}(f, \oplus)y) \\ = & \{ \text{defn } \cdot \} \\ & (g \cdot \mathcal{H}(f, \oplus))x \otimes_i (g \cdot \mathcal{H}(f, \oplus))y \end{aligned}$$

so that the two sides are the same by the Unique Extension Property.

The map absorption theorem A special case of the promotion theorem occurs when $\oplus$ is $++$; that is, when $\mathcal{H}(f, \oplus)$ is a map.

Theorem *The composition of an array homomorphism and a map is another homomorphism:*

$$\mathcal{H}(g, \oplus) \cdot f* = \mathcal{H}(g \cdot f, \oplus)$$

Proof:

By definition of $\mathcal{H}$ the function $\mathcal{H}(g, \oplus)$ satisfies the condition

$$\mathcal{H}(g, \oplus)(x ++_i y) = \mathcal{H}(g, \oplus)x \oplus \mathcal{H}(g, \oplus)y$$

Thus we have

$$\begin{aligned} & \mathcal{H}(g, \oplus) \cdot f* \\ = & \quad \{ \text{defn } * \} \\ & \mathcal{H}(g, \oplus) \cdot \mathcal{H}([_] \cdot f, ++) \\ = & \quad \{ \text{promotion theorem} \} \\ & \mathcal{H}(\mathcal{H}(g, \oplus) \cdot [_] \cdot f, \oplus) \\ = & \quad \{ \text{defn } \mathcal{H} \} \\ & \mathcal{H}(g \cdot f, \oplus) \end{aligned}$$

The homomorphism lemma In any Bird-Meertens style theory of data types [Bird, 1988, Gibbons, 1990], analogues of *map* and *reduce* play a central role. The importance of these two functions arises from the fact that any homomorphism can be factored into the composition of a map and a reduction.

Theorem *An array function $h : A^* \rightarrow B$ is a homomorphism if and only if there exist functions $f : A \rightarrow B$ and $\oplus : \mathcal{D} \rightarrow (B \times B) \rightarrow B$ such that*

$$h = \oplus / \cdot f*$$

Proof:

$$\begin{aligned}
& \oplus / \cdot f * \\
= & \quad \{\text{defn } /\} \\
& \mathcal{H}(id, \oplus) \cdot f * \\
= & \quad \{\text{map absorption}\} \\
& \mathcal{H}(id \cdot f, \oplus) \\
= & \quad \{\text{defn } id\} \\
& \mathcal{H}(f, \oplus)
\end{aligned}$$

3.3 Flatten and transpose

Flatten A particular reduction which is frequently used is *flatten*, defined as $+/$. It is used to ‘flatten’ an array of arrays into an array which is nested less deeply, by reducing along each dimension with the corresponding join constructor. Note that since the $+_i$ constructors are partial, $+/x$ is defined only if the elements along each dimension of x have the same size in every other dimension.

We have array versions of the standard Bird-Meertens promotion laws for *map* and *reduce*:

$$\begin{aligned}
f * \cdot +/ &= +/ \cdot (f *) * \\
\oplus / \cdot +/ &\sqsubseteq \oplus / \cdot (\oplus /) *
\end{aligned}$$

Note that the second law is a refinement rather than an equation: because *flatten* is partial, the right-hand side may be defined when the left-hand side is not.

Consider, for example, the function

$$nestsum = +/ \cdot +/$$

which sums the base elements of a nested array of numerical arrays. Because it begins by flattening its argument, *nestsum* cannot be applied to an irregular

array such as

1 2	3 4
-----	--------

 whose subarrays are not conformant. However, the promoted version

$$nestsum' = +/ \cdot (+/)*$$

is total, because it reduces all subarrays to scalars first.

The map and reduce promotion laws can be interpreted as schemes for parallelisation, viewing a nested array as an abstraction for an array of data which is distributed amongst an array of processes. The map promotion law says that if each process (in parallel) applies the function $f*$ to its own segment, the result is the same as applying $f*$ to the whole array. The reduce promotion law says that if each process reduces its own segment with $\oplus/$, the resulting array (with one element in each process) can be reduced with $\oplus/$ to give the same result as reducing the original array.

The map and reduce promotion laws are special cases of the general *divide and conquer* law which applies to any array homomorphism:

$$\mathcal{H}(f, \oplus) \cdot \# / \sqsubseteq \oplus / \cdot \mathcal{H}(f, \oplus) *$$

Transpose A variation of *flatten* gives us a way to define the generalised transposition of an n -dimensional array. Note that applying the singleton constructor to each element of an array and flattening the result gives the original array; in other words, the identity function on arrays is the homomorphism:

$$id = \# / \cdot [-] *$$

We can transpose any pair of dimensions of an array by reducing along each of the dimensions with the join constructor for the other dimension; that is, for $m, n \in \mathcal{D}$

$$\begin{aligned} tr_{(m,n)} &= \oplus / \cdot [-] * \\ \text{where } \oplus_m &= \#_n \\ \oplus_n &= \#_m \\ \oplus_i &= \#_i \quad \neg i \neq m \wedge i \neq n \end{aligned}$$

The type of tr is $(\mathcal{D} \times \mathcal{D}) \rightarrow A^* \rightarrow A^*$.

The following laws relate tr to *map* and *reduce*:

$$\begin{aligned} f* \cdot tr_{(m,n)} &= tr_{(m,n)} \cdot f* \\ \oplus / \cdot tr_{(m,n)} &= \otimes / \\ \text{where } \otimes_m &= \oplus_n \\ \otimes_n &= \oplus_m \\ \otimes_i &= \oplus_i \quad \neg i \neq m \wedge i \neq n \end{aligned}$$

A consequence is that transposition is its own inverse:

$$tr_{(m,n)} \cdot tr_{(m,n)} = id$$

Proof:

$$\begin{aligned} & tr_{(m,n)} \cdot tr_{(m,n)} \\ = & \{ \text{defn } tr \} \\ & \oplus / \cdot [-] * \cdot tr_{(m,n)} \\ = & \{ \text{map transpose} \} \\ & \oplus / \cdot tr_{(m,n)} \cdot [-] * \\ = & \{ \text{reduce transpose} \} \\ & \# / \cdot [-] * \\ = & \{ id \text{ homo} \} \\ & id \end{aligned}$$

Other properties of tr follow directly from the definition:

$$\begin{aligned} tr_{(m,m)} &= id \\ tr_{(m,n)} &= tr_{(n,m)} \end{aligned}$$

In the context of two-dimensional arrays, the only non-trivial transposition is $tr_{(ROW,COL)} = tr_{(COL,ROW)}$, which we abbreviate to simply tr .

3.4 Zip

We have seen that *map* lifts a unary scalar function to act elementwise on a single array. We can similarly use *map* to lift a binary scalar function to act on the corresponding elements in a pair of arrays, by means of the operator *zip*, denoted by Υ , which transforms a pair of identically-shaped arrays to an array of pairs. For example, an elementwise addition of arrays x and y is denoted by the expression $(+) * (x \Upsilon y)$. Since this idiom is used frequently, we will abbreviate it to $x \Upsilon_+ y$ (pronounced “ x zipwith-plus y ”). For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Upsilon_+ \begin{bmatrix} 10 & 10 & 10 \\ 20 & 20 & 20 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 24 & 25 & 26 \end{bmatrix}$$

To define *zip* formally, we observe that although *zip* is not a homomorphism, it is the inverse of one. The function *unzip*, which transforms an array of pairs to a pair of identically-shaped arrays, can be defined as

$$\begin{aligned} \text{unzip} &= \otimes / \cdot \text{unpair} * \\ \text{where } \text{unpair}(a, b) &= ([a], [b]) \\ (x, y) \otimes_i (u, v) &= (x \uplus_i u, y \uplus_i v) \end{aligned}$$

We can now calculate the definition of *zip* from the specification $\text{zip} \cdot \text{unzip} = \text{id}$. For singletons we have

$$\begin{aligned} &[(a, b)] \\ &= \{ \text{hypothesis} \} \\ &\quad \text{zip}(\text{unzip} [(a, b)]) \\ &= \{ \text{defn unzip} \} \\ &\quad \text{zip}(\text{unpair}(a, b)) \\ &= \{ \text{defn unpair} \} \\ &\quad \text{zip}([a], [b]) \end{aligned}$$

For all $i \in \mathcal{D}$, if w and z are arrays of pairs such that $w \uplus_i z$ is defined, then let $(x, u) = \text{unzip } w$ and $(y, v) = \text{unzip } z$. We have

$$\begin{aligned} &w \uplus_i z \\ &= \{ \text{hypothesis} \} \\ &\quad \text{zip}(\text{unzip}(w \uplus_i z)) \\ &= \{ \text{defn unzip} \} \\ &\quad \text{zip}(\text{unzip } w \otimes_i \text{unzip } z) \\ &= \{ \text{defn } (x, u), (y, v) \} \\ &\quad \text{zip}((x, u) \otimes_i (y, v)) \\ &= \{ \text{defn } \otimes \} \\ &\quad \text{zip}(x \uplus_i y, u \uplus_i v) \end{aligned}$$

Thus *zip* is defined if its arguments are both singletons, or if they can be decomposed into joins $x \uplus_i y$ and $u \uplus_i v$, where the pairs (x, u) and (y, v) are in the range of *unzip* — that is, if x has the same shape as u , and y has the same shape as v . So we have as the definition for *zip*

$$\begin{aligned} [a] \vee [b] &= [(a, b)] \\ (x \uplus_i y) \vee (u \uplus_i v) &= (x \vee u) \uplus_i (y \vee v) \quad \neg (\rho x = \rho u) \wedge (\rho y = \rho v) \end{aligned}$$

Defining $\Upsilon_{\oplus} = \oplus * \Upsilon$ for any binary operator $\oplus$, a simple calculation gives

$$\begin{aligned} [a] \Upsilon_{\oplus} [b] &= [(a, b)] \\ (x \uplus_i y) \Upsilon_{\oplus} (u \uplus_i v) &= (x \Upsilon_{\oplus} u) \uplus_i (y \Upsilon_{\oplus} v) \quad \dashv (\rho x = \rho u) \wedge (\rho y = \rho v) \end{aligned}$$

Laws for *zip* The lifted operator $\Upsilon_{\oplus}$ inherits its algebraic properties from the original operator $\oplus$: the former is associative, commutative, or idempotent if (and only if) the latter is; and $\Upsilon_{\otimes}$ distributes over $\Upsilon_{\oplus}$ if $\otimes$ distributes over $\oplus$.

We can also give laws for absorbing a *map* into a *zip*:

$$\begin{aligned} f * \cdot \Upsilon_{\oplus} &= \Upsilon_{f \cdot \oplus} \\ \Upsilon_{\oplus} \cdot (f * \times g *) &= \Upsilon_{\oplus \cdot (f \times g)} \end{aligned}$$

3.5 Cross-product

$\Upsilon_{\oplus}$ combines arrays only if they are exactly the same shape. A more general way of lifting a binary operator $\oplus$ to act on arrays x and y is to form the cross-product $x \times_{\oplus} y$ (pronounced “ x crosswith-circleplus y ”), which uses $\oplus$ to combine

each element of x with every element of y . For example, if $x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, and

$$y = \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 10 \end{bmatrix}, \text{ then}$$

$$x \times_{+} y = \begin{bmatrix} 1+5 & 2+5 & 1+6 & 2+6 \\ 3+5 & 4+5 & 3+6 & 4+6 \\ 1+7 & 2+7 & 1+8 & 2+8 \\ 3+7 & 4+7 & 3+8 & 4+8 \\ 1+9 & 2+9 & 1+10 & 2+10 \\ 3+9 & 4+9 & 3+10 & 4+10 \end{bmatrix}$$

We can define the *cross* operator as the solution to the equations

$$\begin{aligned} x \times_{\oplus} [a] &= (\oplus a) * x \\ x \times_{\oplus} (y \uplus_i z) &= (x \times_{\oplus} y) \uplus_i (x \times_{\oplus} z) \end{aligned}$$

Alternatively, since for a given x the section $(x \times_{\oplus})$ is a homomorphism, we have the more concise definition

$$\begin{aligned} x \times_{\oplus} &= \text{++} / \cdot f_x * \\ &\text{where } f_x a = (\oplus a) * x \end{aligned}$$

Laws for cross The size in each dimension of a cross product is the product of the corresponding sizes of the arguments:

$$\#_i(x \times_{\oplus} y) = (\#_i x) \times (\#_i y)$$

If either argument is a singleton, a cross product degenerates to a map:

$$\begin{aligned} [a] \times_{\oplus} &= (a \oplus) * \\ \times_{\oplus} [a] &= (\oplus a) * \end{aligned}$$

A *cross* can absorb a *map* in the same way that a *zip* can:

$$\begin{aligned} f * \cdot \times_{\oplus} &= \times_{f \cdot \oplus} \\ \times_{\oplus} \cdot (f * \times g *) &= \times_{\oplus \cdot (f \times g)} \end{aligned}$$

If $\oplus$ is associative, then so is $\times_{\oplus}$.

However, it is not generally the case that $\times_{\oplus}$ is commutative if $\oplus$ is. A simple counterexample is

$$\begin{aligned} x &= \begin{bmatrix} 0 & 10 \end{bmatrix} \\ y &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\ x \times_+ y &= \begin{bmatrix} 1 & 11 & 2 & 12 & 3 & 13 \end{bmatrix} \\ y \times_+ x &= \begin{bmatrix} 1 & 2 & 3 & 11 & 12 & 13 \end{bmatrix} \end{aligned}$$

Because the cross product is asymmetrical, commutativity is obtained only in the special case that the arguments are orthogonal in shape — for example a row vector and a column vector.

Chapter 4

Subarrays

4.1 Vectors and subarrays

Vectors A *vector* is an array whose size in all but (at most) one dimension is 1. In particular, an array x is an i -*vector* (where $i \in \mathcal{D}$ is a dimension) if for all dimensions $j \neq i$ we have $\#_j x = 1$. For example, in the context of two-dimensional arrays where $\mathcal{D} = \{ROW, COL\}$, we speak of *row-vectors* (with height 1) and *column-vectors* (with width 1). By definition a singleton array is also an i -vector for all dimensions i .

The function $vecs_i$ applied to an n -dimensional array yields a nested array, each of whose elements is an i -vector of the original array. This generalises Bird's *rows* and *cols* functions, which give the rows and columns respectively of a two-dimensional array: we can define $rows = vecs_{ROW}$ and $cols = vecs_{COL}$. For example,

if $x = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$, then

$$vecs_{ROW} x = \begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \\ \begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix} \end{bmatrix}$$

$$vecs_{COL} x = \begin{bmatrix} \begin{bmatrix} 1 \\ 5 \end{bmatrix} & \begin{bmatrix} 2 \\ 6 \end{bmatrix} & \begin{bmatrix} 3 \\ 7 \end{bmatrix} & \begin{bmatrix} 4 \\ 8 \end{bmatrix} \end{bmatrix}$$

Formally, $vecs_i$ is the solution to the equations

$$\begin{aligned} vecs_i[a] &= [[a]] \\ vecs_i(x \uparrow_i y) &= vecs_i x \vee_{\uparrow_i} vecs_i y \\ vecs_i(x \uparrow_j y) &= vecs_i x \uparrow_j vecs_i y \quad \neg j \neq i \end{aligned}$$

Properties of $vecs_i$ The result of $vecs_i$ has the same shape as its argument, except that the size along dimension i is reduced to 1:

$$\begin{aligned} \#_i \cdot vecs_i &= 1^\circ \\ \#_j \cdot vecs_i &= \#_j \quad \neg j \neq i \end{aligned}$$

Each element of the resulting array is an i -vector, whose length is the size along dimension i of the original array:

$$\begin{aligned} \#_i * (vecs_i x) &= (\#_i x)^\circ * (vecs_i x) \\ \#_j * (vecs_i x) &= 1^\circ * (vecs_i x) \quad \neg j \neq i \end{aligned}$$

Flattening the result of $vecs_i$ yields the original array:

$$\uparrow / \cdot vecs_i = id$$

Vector reductions Using $vecs_i$ we can define a *single-axis reduction*, which collapses each i -vector of an n -dimensional array, leaving an $(n - 1)$ -dimensional result (strictly speaking, a result whose size along the i th axis is 1). An i -axis reduction with binary scalar operator $\oplus$, denoted by $\oplus/_i$, is simply a uniform reduction mapped to each i -vector of the array argument:

$$\oplus/_i = (\oplus /) * \cdot vecs_i$$

For example, the function $+/_{ROW}$ sums each row of a two-dimensional array, giving a column-vector result:

$$+/_{ROW} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 6 \\ 15 \\ 24 \end{bmatrix}$$

Expanding the above definition, we find that $\oplus/_i$ satisfies the equations

$$\begin{aligned} \oplus/_i[a] &= [a] \\ \oplus/_i(x \uparrow_i y) &= (\oplus/_i x) \vee_{\oplus} (\oplus/_i y) \\ \oplus/_i(x \uparrow_j y) &= (\oplus/_i x) \uparrow_j (\oplus/_i y) \quad \neg j \neq i \end{aligned}$$

Because vector reduction has an array rather than a scalar as a result, it has further algebraic properties which are not shared by general reduction. Vector reduction is idempotent:

$$\oplus/_i \cdot \oplus/_i = \oplus/_i$$

Composition of orthogonal vector reductions is commutative:

$$\oplus/_i \cdot \otimes/_j = \otimes/_j \cdot \oplus/_i \quad \neg j \neq i$$

Subarrays The function $vecs_i$ effectively decomposes an array into vector components oriented along the i th axis. A dual of this operation is the decomposition of an array into $(n - 1)$ -dimensional components, by ‘slicing’ across the i th axis; we will call these subcomponents, whose size along dimension i is 1, the *i-axis subarrays* of the original array. A geometric interpretation is that *i-axis subarrays* are (hyper)-planes normal to the i axis.

The function $subs_i$ applied to an n -dimensional array gives an i -vector, each of whose elements is a subarray with size 1 in dimension i . The definition is very similar to that of $vecs_i$:

$$\begin{aligned} subs_i[a] &= [[a]] \\ subs_i(x \uplus_i y) &= subs_i x \uplus_i subs_i y \\ subs_i(x \uplus_j y) &= subs_i x \downarrow_{\uplus_j} subs_i y \quad \neg j \neq i \end{aligned}$$

Algebraic properties are closely related to those of $vecs_i$. The result of $subs_i$ is an i -vector:

$$\begin{aligned} \#_i \cdot subs_i &= \#^i \\ \#_j \cdot subs_i &= 1^\circ \quad \neg j \neq i \end{aligned}$$

Each element of the resulting array has the same shape as the original array, except that the size along dimension i is 1:

$$\begin{aligned} \#_i * (subs_i x) &= 1^\circ * (subs_i x) \\ \#_j * (subs_i x) &= (\#_j x)^\circ * (subs_i x) \quad \neg j \neq i \end{aligned}$$

Flattening the result of $subs_i$ yields the original array:

$$\uplus/ \cdot subs_i = id$$

4.2 Matrix algebra

In this section, we deal only with two-dimensional numeric arrays, which we use to model *matrices*. In this context we can make some notational abbreviations:

$$\begin{aligned}\ominus &= \#_{COL} \\ \Phi &= \#_{ROW} \\ ht &= \#_{COL} \\ wd &= \#_{ROW} \\ cols &= vecs_{COL} \\ rows &= vecs_{ROW} \\ tr &= tr_{(ROW, COL)}\end{aligned}$$

4.3 Dot product and matrix multiplication

Definitions In matrix algebra, the *dot product* of two equal-length vectors is the sum of the elementwise product of the vectors. This is easily expressed in our notation (denoting dot product by $\boxtimes$) as

$$\boxtimes = +/\cdot \Upsilon_x$$

This general definition holds for any pair of arrays of the same shape, although we will use it only with row or column vectors.

The *matrix sum* (denoted by $\boxplus$) of two equal-shape arrays is simply their elementwise sum:

$$\boxplus = \Upsilon_+$$

The *matrix product* of an $m \times n$ matrix x with an $n \times p$ matrix y is an $m \times p$ matrix z , whose elements are given by

$$z_{ij} = \sum_k x_{ik} \times y_{kj}$$

That is, z_{ij} is the dot product of the i th row of x with the j th column of y . We can express this without the use of indices (denoting matrix multiplication by $\boxtimes$) as

$$x \boxtimes y = rows\ x \times_{\boxtimes} (tr * cols\ y)$$

This definition holds provided each row of x is the same shape as the transpose of each column of y ; *i.e.* when the width of x is the same as the height of y .

Laws for matrix operations The usual algebraic identities for matrix algebra can be proved from our definition:

$$\begin{aligned}
 x \boxtimes (y \boxtimes z) &= (x \boxtimes y) \boxtimes z \\
 x \boxtimes (y \boxplus z) &= (x \boxtimes y) \boxplus (x \boxtimes z) \\
 [a] \times_{\times} (x \boxtimes y) &= ([a] \times_{\times} x) \boxtimes y \\
 \mathbf{0} \boxtimes x &= x \boxtimes \mathbf{0} = \mathbf{0} \\
 \mathbf{1} \boxtimes x &= x \boxtimes \mathbf{1} = x
 \end{aligned}$$

- matrix product is associative (but not commutative)
- matrix product distributes through matrix sum
- scalar multiplication (*i.e.* multiplication of each element by a scalar) associates with matrix product
- the *zero matrix* acts as a zero for matrix product. Here $\mathbf{0}$ represents an array whose elements are all 0, and whose shape is determined by context; to be more pedantic we ought to write

$$\begin{aligned}
 \mathbf{0}_x \boxtimes x &= \mathbf{0}_x \\
 &\text{where } \mathbf{0}_x = 0^\circ * x
 \end{aligned}$$

- the *identity matrix* is a unit (identity) for matrix product. $\mathbf{1}$ represents an array with elements equal to 1 on the diagonal and 0 elsewhere; that is

$$\begin{aligned}
 \mathbf{1}_{[a]} &= [1] \\
 \mathbf{1}_{((x \oplus y) \ominus (u \oplus v))} &= (\mathbf{1}_x \oplus \mathbf{0}_y) \ominus (\mathbf{0}_u \oplus \mathbf{1}_v) \\
 &\quad \dashv (width\ x = height\ x) \wedge (width\ v = height\ v)
 \end{aligned}$$

The constructive approach also gives various shape-related identities, which are less familiar:

$$\begin{aligned}
 x \boxtimes y &= x \boxdot y && \dashv (ht\ x = wd\ y = 1) \\
 x \boxtimes y &= x \times_{\times} y && \dashv (wd\ x = ht\ y = 1) \\
 (x \oplus y) \boxtimes z &= (x \boxtimes z) \oplus (y \boxtimes z) \\
 x \boxtimes (y \oplus z) &= (x \boxtimes y) \oplus (x \boxtimes z) \\
 (x \oplus y) \boxtimes (u \oplus v) &= (x \boxtimes u) \boxplus (y \boxtimes v)
 \end{aligned}$$

The last three equations generalise by induction to promotion laws:

$$\begin{aligned}
 (\boxtimes x) \cdot \ominus /_i &= \ominus /_i \cdot (\boxtimes x) * \\
 (x \boxtimes) \cdot \oplus /_i &= \oplus /_i \cdot (x \boxtimes) * \\
 (\oplus /_i x) \boxtimes (\ominus /_i y) &= \boxplus /_i (x \times_{\times} y)
 \end{aligned}$$

4.4 Parallel matrix multiplication

There are a number of well-known algorithms for performing matrix multiplication in parallel. By transforming the definition of $\boxtimes$, we can obtain equivalent forms which have different interpretations as parallel control structures.

Beginning with the original definition

$$\begin{aligned} x \boxtimes y &= \text{rows } x \times_{\boxtimes} (\text{tr} * \text{cols } y) \\ &= \boxtimes * (\text{rows } x \times (\text{tr} * \text{cols } y)) \end{aligned}$$

where x and y have shape $m \times n$ and $n \times p$ respectively, we see that the outermost operation is a *map*, operating on an array of shape $m \times p$. This can be performed in parallel on $m \times p$ processors, each independently evaluating the dot product of one row of x with one column of y — an operation which takes n sequential steps.

Decomposing y by columns gives

$$\begin{aligned} &x \boxtimes y \\ = &\quad \{\text{flatten vecs}\} \\ &x \boxtimes (\phi /_{\text{ROW}} \text{cols } y) \\ = &\quad \{\boxtimes - \phi \text{ promotion}\} \\ &\phi /_{\text{ROW}} (x \boxtimes * \text{cols } y) \end{aligned}$$

This is a *map*, which can be performed independently in parallel on p processors, assigning each a column of y and a copy of the whole of x . Each processor takes $m \times n$ sequential steps to form the matrix product of x with its column of y .

Similarly, decomposing x by rows gives

$$\begin{aligned} &x \boxtimes y \\ = &\quad \{\text{flatten vecs}\} \\ &(\ominus /_{\text{COL}} \text{rows } x) \boxtimes y \\ = &\quad \{\boxtimes - \ominus \text{ promotion}\} \\ &\ominus /_{\text{COL}} (\boxtimes y * \text{rows } x) \end{aligned}$$

This *map* can be performed on m processors, each taking $n \times p$ steps to form the matrix product of y with one row of x .

Another decomposition is

$$\begin{aligned} &x \boxtimes y \\ = &\quad \{\text{flatten vecs}\} \end{aligned}$$

$$\begin{aligned}
& (\phi /_{ROW} cols\ x) \boxtimes (\ominus /_{COL} rows\ y) \\
= & \quad \{\text{vector reduce} - \text{transpose}\} \\
& (\phi /_{ROW} cols\ x) \boxtimes (\ominus /_{ROW} tr(rows\ y)) \\
= & \quad \{\boxtimes - \phi, \ominus \text{ promotion}\} \\
& \boxplus /_{ROW} (cols\ x\ \curlywedge \boxtimes tr(rows\ y)) \\
= & \quad \{\text{column } \boxtimes \text{ row}\} \\
& \boxplus /_{ROW} (cols\ x\ \curlywedge_{\times} tr(rows\ y)) \\
= & \quad \{\text{defn } \boxplus\} \\
& \curlyvee_+ /_{ROW} (cols\ x\ \curlywedge_{\times} tr(rows\ y))
\end{aligned}$$

In this version, n processors each form an $m \times p$ array which is the ‘outer product’ of one column of x with one row of y ; this operation takes $m \times p$ steps. (Alternatively, if $m \times n \times p$ processors are available, the $\curlywedge_{\times}$ can be performed in parallel, in a single step.) The resulting n arrays must be summed with a $\curlyvee_+$ reduction.

4.5 Directed reductions

The reduction schemes we have defined so far require the binary operator arguments to be associative and pairwise abiding, so that the result will be independent of the order of evaluation. To reduce an array using a non-associative operator $\oplus$, it is necessary to specify how to parenthesise the expression; *i.e.* to give a direction for the reduction.

If, furthermore, $\oplus$ does not abide with itself, it is also necessary to give an ordering on the dimensions. To reduce complexity, we define only directed vector reductions (reductions along a single axis), since more general reductions can be built as the composition of orthogonal vector reductions.

The *left i -reduction* and *right i -reduction*, denoted by $\not\curlyvee_i$ and $\not\curlyvee_i$, reduce an array by inserting a binary operator between adjacent elements along dimension i , associating to the left and right respectively. For example, if $x =$

1	2	3	4
5	6	7	8

, left

and right row-reductions with the (non-associative) subtraction operator give

$$\begin{aligned}
 - \not\vdash_{\text{ROW}} x &= \begin{bmatrix} ((1-2)-3)-4 \\ ((5-6)-7)-8 \end{bmatrix} = \begin{bmatrix} -8 \\ -16 \end{bmatrix} \\
 - \not\vdash_{\text{ROW}} x &= \begin{bmatrix} 1-(2-(3-4)) \\ 5-(6-(7-8)) \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \end{bmatrix}
 \end{aligned}$$

The following equations define $\not\vdash$ and $\not\vdash$ uniquely:

$$\begin{aligned}
 \oplus \not\vdash_i [a] &= [a] \\
 \oplus \not\vdash_i (x \#_i [a]) &= [b \oplus a] \\
 &\quad \text{where } [b] = \oplus \not\vdash_i x \\
 \oplus \not\vdash_i (x \#_j y) &= (\oplus \not\vdash_i x) \#_j (\oplus \not\vdash_i y) \quad \dashv j \neq i \\
 \oplus \not\vdash_i [a] &= [a] \\
 \oplus \not\vdash_i ([a] \#_i x) &= [a \oplus b] \\
 &\quad \text{where } [b] = \oplus \not\vdash_i x \\
 \oplus \not\vdash_i (x \#_j y) &= (\oplus \not\vdash_i x) \#_j (\oplus \not\vdash_i y) \quad \dashv j \neq i
 \end{aligned}$$

The operations are well-defined because

- Every non-singleton array can be uniquely constructed as a subarray of i -vectors, for any $i \in \mathcal{D}$.
- Every non-singleton i -vector can be uniquely constructed as an i -vector joined to a singleton, and as a singleton joined to an i -vector.

An alternative definition of directed reduction is based on the fact that every non-singleton array can be constructed as an array joined along dimension i to an i -axis subarray; this view is analogous to the *cons* form of list construction.

$$\begin{aligned}
 \oplus \not\vdash_i [a] &= [a] \\
 \oplus \not\vdash_i (x \#_i y) &= (\oplus \not\vdash_i x) \vee_{\oplus} y \quad \dashv \#_i y = 1 \\
 \oplus \not\vdash_i (x \#_j y) &= (\oplus \not\vdash_i x) \#_j (\oplus \not\vdash_i y) \quad \dashv j \neq i
 \end{aligned}$$

Left and right reductions are related by a duality law:

$$\begin{aligned}
 \oplus \not\vdash_i &= \tilde{\oplus} \not\vdash_i \cdot \text{refl}_i \\
 \oplus \not\vdash_i &= \tilde{\oplus} \not\vdash_i \cdot \text{refl}_i
 \end{aligned}$$

where the *refl* operator is a generalisation of the list *reverse* function: $refl_i$ is a homomorphism which reverses each i -vector in an array.

$$\begin{aligned} refl_i &= \oplus / \cdot [-] * \\ \text{where } \oplus_i &= \widetilde{\#}_i \\ \oplus_j &= \#_j \quad \neg j \neq i \end{aligned}$$

If the scalar operator $\oplus$ is associative, left and right reductions with $\oplus$ give the same result as the non-directed vector reduction.

$$\oplus \#_i = \oplus \#_i = \oplus /_i \quad \neg \oplus \text{ associative}$$

Thus the directed reductions can be seen as more sequential or deterministic ‘implementations’ of the operation specified by the more general reduction.

Conversely, in some cases a directed reduction — even with a non-associative operator — can be replaced by a non-directed vector reduction, resulting in an operation which can be performed in parallel in $O(\log n)$ time.

4.6 Accumulations

An *accumulation* operator (also known as *scan* or *prefix*) can be defined for arrays by applying a vector accumulation along a given dimensional axis. The *left* and *right i -accumulations*, denoted $\#_i$ and $\#_i$, are equivalent to applying a directed reduction to each of the (non-empty) initial or tail segments respectively of the i -vectors of an array.

Formally,

$$\begin{aligned} \oplus \#_i [a] &= [a] \\ \oplus \#_i (x \#_i y) &= z \#_i ((\gg /_i z) \vee_{\oplus} y) \quad \neg \#_i y = 1 \\ &\quad \text{where } z = \oplus \#_i x \\ \oplus \#_i (x \#_j y) &= (\oplus \#_i x) \#_j (\oplus \#_i y) \quad \neg j \neq i \end{aligned}$$

Chapter 5

Programming with arrays

5.1 Deriving an algorithm for LU decomposition

Given a square nonsingular matrix X , its LU decomposition is a pair of square matrices $\text{lud } X = (L, U)$ such that

- $L \boxtimes U = X$
- L is lower triangular: all elements above the diagonal are zero
- U is upper triangular: all elements below the diagonal are zero

The first two conditions are easy to satisfy, by setting $(L, U) = (\mathbf{1}, X)$. Our strategy will be to transform this pair to satisfy the third condition, without invalidating the first two. That is, we want to make U ‘more upper triangular’ (changing elements below the diagonal to 0), while maintaining the invariant $L \boxtimes U = X$ by changing only below-diagonal elements of L .

We can characterise upper and lower triangular matrices recursively by a decomposition into quadrants. If we take

$$x = (x_1 \oplus x_2) \ominus (x_3 \oplus x_4)$$

where x_1 and x_4 are square matrices (not necessarily the same size), then x is upper triangular if x_1 and x_4 are upper triangular, and x_3 is all zero. Similarly, x is lower triangular if x_1 and x_4 are, and x_2 is all zero.

Decomposing L , U and X into quadrants, we write the invariant as

$$\begin{aligned} & ((L_1 \oplus L_2) \ominus (L_3 \oplus L_4)) \boxtimes ((U_1 \oplus U_2) \ominus (U_3 \oplus U_4)) \\ & = ((X_1 \oplus X_2) \ominus (X_3 \oplus X_4)) \end{aligned}$$

The shape-related identities for matrix product allow the left-hand side to be rewritten as

$$\begin{aligned} & ((L_1 \oplus L_2) \ominus (L_3 \oplus L_4)) \boxtimes ((U_1 \oplus U_2) \ominus (U_3 \oplus U_4)) \\ = & \{ \boxtimes - \ominus \} \\ & ((L_1 \oplus L_2) \boxtimes ((U_1 \oplus U_2) \ominus (U_3 \oplus U_4))) \\ & \ominus \\ & ((L_3 \oplus L_4) \boxtimes ((U_1 \oplus U_2) \ominus (U_3 \oplus U_4))) \\ = & \{ \ominus, \phi \text{ abide} \} \\ & ((L_1 \oplus L_2) \boxtimes ((U_1 \ominus U_3) \oplus (U_2 \ominus U_4))) \\ & \ominus \\ & ((L_3 \oplus L_4) \boxtimes ((U_1 \ominus U_3) \oplus (U_2 \ominus U_4))) \\ = & \{ \boxtimes - \phi \} \\ & (((L_1 \oplus L_2) \boxtimes (U_1 \ominus U_3)) \oplus ((L_1 \oplus L_2) \boxtimes (U_2 \ominus U_4))) \\ & \ominus \\ & (((L_3 \oplus L_4) \boxtimes (U_1 \ominus U_3)) \oplus ((L_3 \oplus L_4) \boxtimes (U_2 \ominus U_4))) \\ = & \{ \boxtimes - \phi, \ominus \} \\ & (((L_1 \boxtimes U_1) \boxplus (L_2 \boxtimes U_3)) \oplus ((L_1 \boxtimes U_2) \boxplus (L_2 \boxtimes U_4))) \\ & \ominus \\ & (((L_3 \boxtimes U_1) \boxplus (L_4 \boxtimes U_3)) \oplus ((L_3 \boxtimes U_2) \boxplus (L_4 \boxtimes U_4))) \end{aligned}$$

Now that left-hand and right-hand sides are decomposed into quadrants the same way, we can use the individuality law

$$((x \boxplus_i y) = (u \boxplus_i v)) \iff (x = u \wedge y = v) \dashv (\rho x = \rho u \wedge \rho y = \rho v)$$

to rewrite the invariant as four independent equations

$$\begin{aligned} X_1 &= (L_1 \boxtimes U_1) \boxplus (L_2 \boxtimes U_3) \\ X_2 &= (L_1 \boxtimes U_2) \boxplus (L_2 \boxtimes U_4) \\ X_3 &= (L_3 \boxtimes U_1) \boxplus (L_4 \boxtimes U_3) \\ X_4 &= (L_3 \boxtimes U_2) \boxplus (L_4 \boxtimes U_4) \end{aligned}$$

Making L and U lower and upper triangular respectively requires L_2 and U_3 to be all zero, giving

$$\begin{aligned} X_1 &= (L_1 \boxtimes U_1) \boxplus (\mathbf{0} \boxtimes \mathbf{0}) = L_1 \boxtimes U_1 \\ X_2 &= (L_1 \boxtimes U_2) \boxplus (\mathbf{0} \boxtimes U_4) = L_1 \boxtimes U_2 \\ X_3 &= (L_3 \boxtimes U_1) \boxplus (L_4 \boxtimes \mathbf{0}) = L_3 \boxtimes U_1 \\ X_4 &= (L_3 \boxtimes U_2) \boxplus (L_4 \boxtimes U_4) \end{aligned}$$

Setting L_1 to the identity matrix $\mathbf{1}$ allows us to simplify the first two equations

$$\begin{aligned} X_1 &= \mathbf{1} \boxtimes U_1 = U_1 \\ X_2 &= \mathbf{1} \boxtimes U_2 = U_2 \end{aligned}$$

To solve the third equation for L_3 , given X_3 and U_1 requires that we find the *inverse* of U_1 : that is, the matrix U_1^{-1} such that $U_1 \boxtimes U_1^{-1} = \mathbf{1}$. Given such an inverse, we could then calculate

$$\begin{aligned} &L_3 \\ &= \{\text{identity matrix}\} \\ &L_3 \boxtimes \mathbf{1} \\ &= \{\text{inverse}\} \\ &L_3 \boxtimes U_1 \boxtimes U_1^{-1} \\ &= \{X_3 = L_3 \boxtimes U_1\} \\ &X_3 \boxtimes U_1^{-1} \end{aligned}$$

Now, there is one special case in which inverting a matrix is very easy: if U_1 is a singleton array $[a]$, then matrix multiplication by U_1 degenerates to scalar multiplication, so that the inverse is $\left[\frac{1}{a}\right]$. So we constrain the quadrant decomposition to make U_1 (and hence L_1 and X_1) a singleton; which means that U_2 must be a row vector and U_3 a column vector.

Finally, we want to solve the fourth equation for L_4 and U_4 , given L_3 and U_2 . Since the inverse of elementwise addition is elementwise subtraction, this means finding (L_4, U_4) such that

- $L_4 \boxtimes U_4 = X_4 \curlywedge (L_3 \boxtimes U_2)$
- L_4 is lower triangular
- U_4 is upper triangular

But this is just a copy of the initial problem, reduced in size by one row and one column; its solution is therefore

$$\begin{aligned}(L_4, U_4) &= lud(X_4 \vee_- (L_3 \boxtimes U_2)) \\ &= lud(X_4 \vee_- (L_3 \times_\times U_2))\end{aligned}$$

Before putting it all together into a recursive definition for *lud*, we make a space optimisation: since *L* always has ones on the diagonal and zeroes below, and *U* always has zeroes above the diagonal, the ‘interesting’ parts of *L* and *U* can be packed together into a single matrix. We have

$$\begin{aligned}LU_1 &= U_1 = X_1 \\ LU_2 &= U_2 = X_2 \\ LU_3 &= L_3 = X_3 \times_\div U_1 \\ LU_4 &= lud(X_4 \vee_- (L_3 \times_\times U_2))\end{aligned}$$

Thus the function *lud* will calculate $LU = (L \vee_- \mathbf{1}) \vee_+ U$, which can easily be decomposed to (L, U) .

The complete definition is

$$\begin{aligned}lud [a] &= [a] \\ lud (([x_1] \oplus x_2) \ominus (x_3 \oplus x_4)) &= ([x_1] \oplus x_2) \ominus (x'_3 \oplus x'_4) \\ &\quad \text{where } x'_3 = x_3 \times_\div x_1 \\ &\quad x'_4 = lud (x_4 \vee_- (x'_3 \times_\times x_2))\end{aligned}$$

5.2 Partial differential equations

Partial differential equations are ubiquitous in the physical sciences. Many of the ‘laws of nature’ can be stated as differential equations, in which the value of some quantity as a function of spatial position (in one, two or three dimensions) or of position and time (up to four dimensions) is defined implicitly by a relation on the function’s partial derivatives.

The Poisson equation As a specific example, we will consider a boundary value problem: an instance of the *Poisson equation* in three dimensions

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2}(x, y, z) + \frac{\partial^2 u}{\partial y^2}(x, y, z) + \frac{\partial^2 u}{\partial z^2}(x, y, z) &= q(x, y, z) \quad , \\ (0 < x < 1) \wedge (0 < y < 1) \wedge (0 < z < 1) \\ u(x, y, z) &= f(x, y, z) \quad , \\ (x = 0) \vee (x = 1) \vee (y = 0) \vee (y = 1) \vee (z = 0) \vee (z = 1)\end{aligned}$$

which specifies a real-valued function u in a unit cube of space in terms of its second partial derivatives. The given functions f and q are respectively the *boundary condition* (fixing the value of u on the surfaces of the cube) and the *source term* (which accounts for interior inhomogeneities). This equation can be given a variety of physical interpretations: for example, u models the steady-state temperature distribution in a block of material, where the surfaces are held at constant temperatures given by f , and q defines the distribution of heat sources within the block.

While one-dimensional ordinary differential equations can often be solved symbolically, this is seldom the case for PDE's. The Poisson equation has an explicit closed-form solution only for extremely simple cases of f and q . In general, we must instead use numerical techniques to solve a discrete approximation of the equation at a set of points which form a grid over the region in question, and then use interpolation if necessary to find values of u at other points.

In this section we consider *finite difference* methods, in which partial derivatives are approximated by discrete difference operators. These methods fit naturally into our array model: the differential equation to be solved is replaced by an *array equation*, which can be transformed using algebraic laws into a program to generate its solution.

Finite difference approximations For each dimension $i \in \{x, y, z\}$, we define a *central difference* operator δ_i (assuming a uniform grid spacing of h), which has type $(\mathbf{R}^3 \rightarrow \mathbf{R}) \rightarrow (\mathbf{R}^3 \rightarrow \mathbf{R})$:

$$\begin{aligned}(\delta_x f)(x, y, z) &= f(x + \tfrac{h}{2}, y, z) - f(x - \tfrac{h}{2}, y, z) \\(\delta_y f)(x, y, z) &= f(x, y + \tfrac{h}{2}, z) - f(x, y - \tfrac{h}{2}, z) \\(\delta_z f)(x, y, z) &= f(x, y, z + \tfrac{h}{2}) - f(x, y, z - \tfrac{h}{2})\end{aligned}$$

The first partial derivative $\frac{\partial f}{\partial x}$ is approximated by $\frac{\delta_x f}{h}$ to any required degree of accuracy by using a fine enough grid, since

$$\lim_{h \rightarrow 0} \frac{\delta_x f(x, y, z)}{h} = \frac{\partial f(x, y, z)}{\partial x}$$

This is easily proved by subtracting the Taylor series expansions of $f(x + \frac{h}{2}, y, z)$ and $f(x - \frac{h}{2}, y, z)$.

Similarly, we approximate the second partial derivative $\frac{\partial^2 f}{\partial x^2}$ by the second central difference $\frac{\delta_x^2 f}{h^2}$, where

$$\begin{aligned} (\delta_x^2 f)(x, y, z) &= (\delta_x(\delta_x f))(x, y, z) \\ &= (\delta_x f)(x + \frac{h}{2}, y, z) - (\delta_x f)(x - \frac{h}{2}, y, z) \\ &= (f(x + h, y, z) - f(x, y, z)) - (f(x, y, z) - f(x - h, y, z)) \\ &= f(x + h, y, z) - 2f(x, y, z) + f(x - h, y, z) \end{aligned}$$

Thus by replacing derivatives with difference operators, we can approximate the Poisson equation with a finite difference equation:

$$\frac{\delta_x^2 u(x, y, z)}{h^2} + \frac{\delta_y^2 u(x, y, z)}{h^2} + \frac{\delta_z^2 u(x, y, z)}{h^2} = q(x, y, z)$$

or, equivalently,

$$\delta_x^2 u(x, y, z) + \delta_y^2 u(x, y, z) + \delta_z^2 u(x, y, z) - h^2 \times q(x, y, z) = 0$$

Discretising to an array equation To assist in transforming the difference equation to an array equation, we will define a new ‘application specific operator’, and introduce some notational abbreviations.

The *frame* operator, denoted by $\boxdot$, builds an array from the boundary elements of its left argument and the interior elements of its right argument. Note that the notion of ‘boundary’ depends on dimensionality: for example a row vector (of length > 2) considered as a one-dimensional array has only the two end-points as boundary elements, whereas the same vector considered as a two- or three-dimensional array is all boundary and no interior. Thus a generic definition of *frame* must be explicitly parameterised by $\mathcal{D}$. In the present context, we will define only a three-dimensional version:

$$\begin{aligned} x \boxdot &= x \boxdot_x \cdot x \boxdot_y \cdot x \boxdot_z \\ \text{where } x \boxdot_i y &= \uplus / (vecs_i x) \downarrow_{\oplus} (vecs_i y) \\ [a] \oplus [b] &= [a] \\ ([a] \uplus_j x \uplus_j [b]) \oplus ([c] \uplus_j y \uplus_j [d]) &= [a] \uplus_j y \uplus_j [b] \end{aligned}$$

To make the concrete syntax of our array expressions less busy, we introduce some notational simplifications. We allow the arithmetic operators $(+, -, \times, \div)$ to be *overloaded* and used between pairs of arrays or between an array and a scalar, with the obvious element-wise interpretation: using uppercase letters to represent

arrays, and lowercase letters (or numeric constants) for scalars, if $\oplus$ is an arithmetic operator, then

$$\begin{aligned} X \oplus Y & \text{ means } X \vee_{\oplus} Y \\ a \oplus X & \text{ means } (a \oplus) * X \\ X \oplus a & \text{ means } (\oplus a) * X \end{aligned}$$

In a further abuse of notation, we use numeric constants to represent constant-valued arrays whose shape is determined by context in the expression; for example

$$0 \boxtimes X \text{ means } (0^\circ * X) \boxtimes X$$

Finally, the expressions X_{i+} and X_{i-} denote respectively left and right zero-fill shifts of the array X by one position along dimension i ; or, more formally

$$\begin{aligned} X_{i-} &= (0^\circ * X) \quad \vdash \#_i X = 1 \\ (X \#_i Y)_{i-} &= (0^\circ * Y) \#_i X \quad \vdash \#_i Y = 1 \\ X_{i+} &= (0^\circ * X) \quad \vdash \#_i X = 1 \\ (X \#_i Y)_{i+} &= Y \#_i (0^\circ * X) \quad \vdash \#_i X = 1 \end{aligned}$$

Now we proceed towards an array version of the Poisson equation by dividing the unit cube into n^3 smaller cubes, with diameter $h = \frac{1}{n}$, and defining the array X to carry the grid of (x, y, z) values at the vertices of each cube:

$$\begin{aligned} X &= (h \times) * ((0 \dots_x n) \times (0 \dots_y n) \times (0 \dots_z n)) \\ &\text{where } n = 1 \div h \end{aligned}$$

The difference equation is promoted to an array equation using the law $(f = g) \iff (f* = g*)$ and combining the conditions on interior and boundary points with *frame*:

$$(u * X) \boxtimes ((\delta_x^2 u * X) + (\delta_y^2 u * X) + (\delta_z^2 u * X)) = (f * X) \boxtimes h^2 \times (q * X)$$

Since X is a grid of evenly-spaced values, we can show that for any A

$$\begin{aligned} A \boxtimes (((-h) \times id \times id) * X) &= A \boxtimes X_{x-} \\ A \boxtimes (((+h) \times id \times id) * X) &= A \boxtimes X_{x+} \end{aligned}$$

and hence we can rewrite

$$\begin{aligned}
& A \boxminus (\delta_x^2 u * X) \\
= & \{ \text{unfold and promote } \delta_x^2 u \} \\
& A \boxminus ((u * ((-h) \times id \times id) * X) - (2 \times (u * X)) \\
& \quad + (u * ((+h) \times id \times id) * X)) \\
= & \{ \text{promote } A \boxminus \} \\
& A \boxminus ((u * A \boxminus ((-h) \times id \times id) * X) - (2 \times (u * X)) \\
& \quad + (u * A \boxminus ((+h) \times id \times id) * X)) \\
= & \{ \text{above result} \} \\
& A \boxminus ((u * A \boxminus X_{x-}) - (2 \times (u * X)) + (u * A \boxminus X_{x+})) \\
= & \{ \text{promote } A \boxminus \} \\
& A \boxminus ((u * X_{x-}) - (2 \times (u * X)) + (u * X_{x+})) \\
= & \{ \text{shift promotes through } map \} \\
& A \boxminus ((u * X)_{x-} - (2 \times (u * X)) + (u * X)_{x+})
\end{aligned}$$

Using similar results for δ_y^2 and δ_z^2 , we can factor $u * X$ from the difference equation, giving

$$\begin{aligned}
U \boxminus ((U_{x-} - 2 \times U + U_{x+}) + (U_{y-} - 2 \times U + U_{y+}) + (U_{z-} - 2 \times U + U_{z+})) \\
= B \boxminus Q
\end{aligned}$$

where the arrays B , Q , and U represent the values of the boundary function, the source term (multiplied by h^2), and the function to be solved for, each mapped over the grid:

$$\begin{aligned}
U &= u * X \\
B &= f * X \\
Q &= h^2 \times (q * X)
\end{aligned}$$

In place of the original partial differential equation, we now have an array problem, which we will call the *Poisson array equation*: given arrays B and Q , we want to solve for U .

5.2.1 The point Jacobi method

A *direct* solution of the Poisson array equation is theoretically possible, but impractical: it would require finding the inverse of a matrix of size $n^3 \times n^3$. Instead,

we use an iterative or *relaxation* method. Suppose we have an equation in the form

$$x = F x$$

which we must solve for x given a known function F . We start with an initial ‘guess’ x_0 , and compute successive approximations using the recurrence

$$x_{i+1} = F x_i$$

If for some value of i we find that $x_{i+1} = x_i$, then x_i is a *fixed-point* of F , and thus satisfies the equation $x = F x$.

Casting the Poisson array equation into fixed-point form, we obtain

$$\begin{aligned} & U \boxminus ((U_{x-} - 2 \times U + U_{x+}) + (U_{y-} - 2 \times U + U_{y+}) \\ & \quad + (U_{z-} - 2 \times U + U_{z+})) = B \boxminus Q \\ \iff & \quad \{ \text{rearrange interior terms} \} \\ & U \boxminus (U_{x-} + U_{x+} + U_{y-} + U_{y+} + U_{z-} + U_{z+} - Q) = B \boxminus (6 \times U) \\ \iff & \quad \{ \text{multiply interior terms by } \tfrac{1}{6} \} \\ & U \boxminus \tfrac{1}{6} \times (U_{x-} + U_{x+} + U_{y-} + U_{y+} + U_{z-} + U_{z+} - Q) = B \boxminus U \\ \iff & \quad \{ \text{rearrange boundary terms} \} \\ & B \boxminus \tfrac{1}{6} \times (U_{x-} + U_{x+} + U_{y-} + U_{y+} + U_{z-} + U_{z+} - Q) = U \boxminus U \\ \iff & \quad \{ \boxminus \text{ is idempotent} \} \\ & B \boxminus \tfrac{1}{6} \times (U_{x-} + U_{x+} + U_{y-} + U_{y+} + U_{z-} + U_{z+} - Q) = U \end{aligned}$$

This form of the equation gives rise to the *point Jacobi* relaxation method. Considering the left-hand side of the equation as a function of U , which we will call *jacobistep*, we take B as our initial approximation, and apply *jacobistep* iteratively until convergence is reached.

How do we recognise convergence? Since we are solving only an approximation to the original partial differential equation (and since in the real world we will be dealing with imprecise floating-point arithmetic rather than real numbers), it is too strict to require that *jacobistep* $U = U$ exactly. Instead, we require only that the difference between *jacobistep* U and U , with respect to some norm $\| \cdot \|$, is less than a given tolerance ϵ . A suitable norm for arrays is the maximum of the elementwise absolute differences; that is,

$$\begin{aligned} \text{diff}(X, Y) &= \|X - Y\| \\ &= \uparrow \text{abs} * (X \text{ } \vee \text{ } Y) \end{aligned}$$

We can now write the relaxation procedure as a function, which takes a tolerance ϵ and a pair of arrays giving boundary and source values, and solves the Poisson array equation by iterating the Jacobi step function:

$$\text{solve } \epsilon (B, Q) = \text{relax } \epsilon (\text{jacobistep } (B, Q)) B$$

$$\begin{aligned} \text{relax } \epsilon f x &= x' && , \text{ if } \text{diff}(x, x') < \epsilon \\ &= \text{relax } \epsilon f (f x') && , \text{ otherwise} \\ &\text{where } x' = f x \end{aligned}$$

$$\text{jacobistep } (B, Q) u = B \boxtimes \frac{1}{6} \times (u_{x-} + u_{x+} + u_{y-} + u_{y+} + u_{z-} + u_{z+} - Q)$$

To show that we can expect the relaxation process to converge, we define the *error term* for a given approximation U to be $\|U - W\|$, where W is the (unknown) exact solution. By definition of $\| \cdot \|$, the error term is bounded below by 0. We calculate the error term for $F U$ to be

$$\begin{aligned} &\|F U - W\| \\ &= \{ F W = W \} \\ &\|F U - F W\| \\ &= \{ \text{defn } F \} \\ &\|B \boxtimes \frac{1}{6} \times (U_{x-} + U_{x+} + U_{y-} + U_{y+} + U_{z-} + U_{z+} - Q) \\ &\quad - (B \boxtimes \frac{1}{6} \times (W_{x-} + W_{x+} + W_{y-} + W_{y+} + W_{z-} + W_{z+} - Q))\| \\ &= \{ \text{rearrange terms} \} \\ &\|0 \boxtimes \frac{1}{6} \times ((U - W)_{x-} + (U - W)_{x+} + (U - W)_{y-} + (U - W)_{y+} \\ &\quad + (U - W)_{z-} + (U - W)_{z+})\| \\ &\leq \{ \uparrow \text{abs} * (0 \boxtimes x) \leq \uparrow \text{abs} * x \} \\ &\|\frac{1}{6} \times ((U - W)_{x-} + (U - W)_{x+} + (U - W)_{y-} + (U - W)_{y+} \\ &\quad + (U - W)_{z-} + (U - W)_{z+})\| \\ &\leq \{ \|x + y\| \leq \|x\| + \|y\| \} \\ &\frac{1}{6} \times (\|(U - W)_{x-}\| + \|(U - W)_{x+}\| + \|(U - W)_{y-}\| + \|(U - W)_{y+}\| \\ &\quad + \|(U - W)_{z-}\| + \|(U - W)_{z+}\|) \\ &\leq \{ \|x_i\| \leq \|x\| \} \\ &\frac{1}{6} \times (\|U - W\| + \|U - W\| + \|U - W\| + \|U - W\| \\ &\quad + \|U - W\| + \|U - W\|) \\ &= \{ \text{algebra} \} \\ &\|U - W\| \end{aligned}$$

Thus F is monotonic with respect to the error function.

5.2.2 Line relaxation

The point Jacobi procedure is a simple *explicit* method: each approximation is computed directly from the previous one, giving the new value of U' in terms of old values of the U_{x-} , U_{x+} , and so on. However, in practice it converges too slowly to be useful for fine grids: in a single iterative step, the new value at each grid point is affected only by the old values at its immediate neighbours, so information propagates very slowly. More rapid convergence can be obtained through *implicit* methods, in which a whole group of new values — a line or a plane — are computed simultaneously by solving an implicit equation.

An implicit method generalises the fixed-point equation $F x = x$ to an equation in two variables $G(x, y) = y$, where G is chosen so that $G(x, x) = F x$. As in explicit relaxation, we begin with a guess x_0 ; successive approximations are found by solving for x_{i+1} in the implicit equation

$$G(x_i, x_{i+1}) = x_{i+1}$$

Again, convergence is reached when $x_{i+1} = x_i$.

We can make an implicit form of the Poisson equation by using new values for the terms which are shifted along one particular dimension, say x :

$$\begin{aligned} & U \boxminus ((U_{x-} - 2 \times U + U_{x+}) + (U_{y-} - 2 \times U + U_{y+}) \\ & \quad + (U_{z-} - 2 \times U + U_{z+})) = B \boxminus Q \\ \Leftarrow & \quad \{ \text{generalise} \} \\ & U' \boxminus ((U'_{x-} - 2 \times U' + U'_{x+}) + (U_{y-} - 2 \times U' + U_{y+}) \\ & \quad + (U_{z-} - 2 \times U' + U_{z+})) = B \boxminus Q \quad \wedge \quad (U' = U) \\ \Longleftrightarrow & \quad \{ \text{rearrange interior terms} \} \\ & U' \boxminus ((U'_{x-} - 6 \times U' + U'_{x+})) \\ & \quad = B \boxminus (Q - (U_{y-} + U_{y+} + U_{z-} + U_{z+})) \quad \wedge \quad (U' = U) \end{aligned}$$

We now have at each iteration to solve an equation of the form

$$U \boxminus ((a \times U_{x-} + b \times U + c \times U_{x+})) = D$$

where the array D represents all the known information on the right-hand side. This process is known as *line relaxation*, since the new value at each interior grid point is related to all the other new values along the same x line.

We can simplify the equation to be solved by a generalisation. The $U \boxminus$ term can be eliminated by replacing the scalar factors a , b , and c by arrays of coefficients

A , B , and C , in such a way that the boundary elements of $A \times U_{x-}$ and $C \times U_{x+}$ are all zero; that is

$$\begin{aligned} A \times U_{x-} + B \times U + C \times U_{x+} &= D \\ \text{where } A &= 0 \boxtimes a \\ B &= 1 \boxtimes b \\ C &= 0 \boxtimes c \end{aligned}$$

Solving the implicit line equation The traditional way to view the implicit equation above is as a collection (one for each x line) of systems of simultaneous linear equations. Expanding the array equation element-wise, using the conventional subscript notation for array indexing, we obtain for each fixed choice of y and z indices j and k a set of interrelated equations

$$\begin{aligned} A_{0,j,k} \times 0 + B_{0,j,k} \times U_{0,j,k} + C_{0,j,k} \times U_{1,j,k} &= D_{0,j,k} \\ A_{1,j,k} \times U_{0,j,k} + B_{1,j,k} \times U_{1,j,k} + C_{1,j,k} \times U_{2,j,k} &= D_{1,j,k} \\ &\dots = \dots \\ A_{n,j,k} \times U_{n-1,j,k} + B_{n,j,k} \times U_{n,j,k} + C_{n,j,k} \times 0 &= D_{n,j,k} \end{aligned}$$

Solving this system is equivalent to finding the inverse of the tridiagonal matrix of coefficients

$$\begin{pmatrix} B_{0,j,k} & C_{0,j,k} & & & 0 \\ A_{1,j,k} & B_{1,j,k} & C_{1,j,k} & & \\ & & \ddots & \ddots & \ddots \\ 0 & & & A_{n,j,k} & B_{n,j,k} \end{pmatrix}$$

for which a number of well-known algorithms exist.

Instead, we will develop a solution for the implicit array equation in array terms, avoiding the use of indices and element-wise reasoning. The algorithm we derive is a multidimensional generalisation of the *Thomas method* for solving a tridiagonal system.

The Thomas method in N dimensions We seek an array u which satisfies the equation

$$au_{i-} + bu + cu_{i+} = d$$

where i is some dimension in $\mathcal{D}$, and $a - d$ are given arrays of coefficients having the same shape as u .

Dealing with the two shifted terms simultaneously makes matters difficult, so we make the hopeful assumption that we can find further coefficient arrays e and f such that

$$u = eu_{i+} + f$$

This is consistent with the original equation if

$$\begin{aligned} & au_{i-} + bu + cu_{i+} = d \\ \Leftarrow & \quad \{ \text{substituting } u_{i-} = e_{i-}u + f_{i-} \} \\ & a(e_{i-}u + f_{i-}) + bu + cu_{i+} = d \\ \Leftrightarrow & \quad \{ \text{isolating terms in } u \} \\ & (ae_{i-} + b)u = -cu_{i+} + d - af_{i+} \\ \Leftrightarrow & \quad \{ \text{dividing} \} \\ & u = \frac{-c}{ae_{i-} + b}u_{i+} + \frac{d - af_{i+}}{ae_{i-} + b} \\ \Leftarrow & \quad \{ \text{assumption } u = eu_{i+} + f \} \\ & e = \frac{-c}{ae_{i-} + b} \wedge f = \frac{d - af_{i+}}{ae_{i-} + b} \end{aligned}$$

Now we have reduced the problem into the simpler cases of solving for e and f .

The equation for e has the form

$$e = e_{i-} \vee_{\oplus} A$$

where $A = a \vee (b \vee c)$ and $\oplus$ represents the operation

$$x \oplus (a, (b, c)) = \frac{-c}{ax + b}$$

Such an equation can be solved easily by what a numerical analyst would call *forward substitution*. We have the simple recurrence

$$\begin{aligned} e_0 &= 0 \oplus A_0 \\ e_{i+1} &= e_i \oplus A_i \end{aligned}$$

But this is precisely the pattern of a left accumulation —

$$e = (0, \oplus) \#_i A$$

Similarly, having found the value of e we can compute f as

$$\begin{aligned} f &= (0, \oplus) \#_i A \\ \text{where } A &= a \vee (b \vee (d \vee e_{i-})) \\ x \oplus (a, (b, (d, e))) &= \frac{d - ax}{ae + b} \end{aligned}$$

Now we can use the values of e and f to solve for u in

$$u = eu_{i+} + f$$

this time by *backward substitution* (right accumulation) —

$$\begin{aligned} u &= (0, \oplus) \#_i (e \vee f) \\ \text{where } (e, f) \oplus x &= ex + f \end{aligned}$$

Line relaxation defined We can now give a complete definition of the function *linestep*, which can be used in place of *jacobistep* to perform a single iteration of line relaxation.

$$\begin{aligned} \text{linestep}(B, Q) u &= \text{thomas}_x(a, b, a) d \\ \text{where } a &= 0 \boxtimes 1 \\ b &= 1 \boxtimes (-6) \\ d &= B \boxtimes (Q - (u_{y-} + u_{y+} + u_{z-} + u_{z+})) \end{aligned}$$

$$\begin{aligned} \text{thomas}_i(a, b, c) d &= (0, \oplus) \#_i (e \vee f) \\ \text{where } (e, f) \oplus x &= e \times x + f \\ f &= (0 \oplus) \#_i (a \vee (b \vee (d \vee e_{i-}))) \\ \text{where } x \oplus (a, (b, (d, e))) &= (d - a \times x) \div (a \times e + b) \\ e &= (0 \oplus) \#_i (a \vee (b \vee c)) \\ \text{where } x \oplus (a, (b, c)) &= -c \div (a \times x + b) \end{aligned}$$

Chapter 6

Bulk-synchronous parallel computation

The bulk-synchronous parallel model of computation was proposed by Leslie Valiant [Valiant, 1989] as an architecture-independent ‘bridge’ between software and hardware for parallel computation. It is intended to play the same part as the traditional von Neumann model does for sequential computation: to be a unifying abstraction of a general-purpose computer, which can serve as an effective target for a variety of high-level programming languages, while being efficiently realisable by a variety of hardware architectures.

6.1 The BSP computation model

In the CACM article *A Bridging Model for Parallel Computation* [Valiant, 1990a], Valiant defines a BSP computer as

“the combination of three attributes:

1. A number of *components*, each performing processing and/or memory functions;
2. A *router* that delivers messages point to point between pairs of components; and
3. Facilities for synchronizing all or a subset of the components at regular intervals of L time units where L is the *periodicity* parameter.”

As the term *bulk-synchronous* suggests, the BSP model lies between the extremes of asynchronous and synchronous parallelism. In an asynchronous parallel computing system, processors can work independently at different rates, and are synchronised implicitly in pairs when a communication takes place. In a fully synchronous system, processors proceed in lock-step, synchronised by a global clock on every instruction. The processors of a BSP computer likewise proceed together synchronously through a computation, but the granularity of synchronisation is not a single instruction step but a larger program segment which Valiant calls a *superstep*.

“In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components.”

While the sequencing of supersteps in a BSP computation is synchronous, operation within each superstep can be entirely asynchronous. If each processor operates only on local data which was present at the beginning of the superstep, its computational progress during the superstep will be independent of the other processors. Any exchange of data between processors can also proceed asynchronously, provided that the data communicated during a superstep is not used by the recipient until a subsequent superstep. Successive supersteps are separated by a global barrier synchronisation, which ensures that all processors have reached the end of one superstep (and all asynchronous data exchanges have taken place) before any process proceeds to the next.

Valiant suggests two variations of the mechanism for barrier synchronisation. In the first, barriers are assumed to occur at fixed time intervals:

“After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep.”

If the synchronisation is performed by hardware which is independent of both the processors and the router, then local computation and global communication can be perfectly overlapped with the check for superstep completion. Thus the elapsed time for each superstep will be an exact multiple of L . The apparent waste of time implied by this fixed-period synchronisation model (on average $L/2$ if the actual time required for computation and communication is uniformly distributed) is avoided by the second variant:

“the system could continuously check whether the current superstep is completed, and allow it to proceed to the next superstep as soon as completion is detected.”

In this version the synchronisation period is variable, and L represents the minimum elapsed time between successive barriers.

Valiant’s original model is general enough to encompass the whole range of Flynn’s taxonomy [Flynn, 1972] of computer architectures. The SISD (single instruction, single data) sequential machine represents the degenerate case of a single component, with no need for communication or synchronisation. A SIMD (single instruction, multiple data) machine is modelled by setting $L = 1$: each instruction is a superstep, in which every active processor performs an identical operation on local values, or a global synchronous shift operation redistributes data between processors. The general case of a MIMD (multiple instruction, multiple data) machine is provided for by allowing extra pairwise synchronisations during a single superstep, for example using message-passing or shared-memory locking protocols. If only these mechanisms are used, we again have a degenerate case, where the entire computation is a single superstep. However, any MIMD machine can also be programmed to act as a ‘pure’ BSP computer, where application programs are written to use explicitly only barrier synchronisations, and the barriers are implemented using the native synchronisation mechanisms of the underlying machine.

Later expositions of BSP by Valiant, McColl and other writers have slightly simplified and specialised the model, favouring the pure BSP approach.

- The components are taken to be simple Von Neumann machines consisting of a single sequential processor and a local memory [Gerbessiotis and Valiant, 1992, McColl, 1993a, Miller, 1993]. This definition is less applicable to the more exotic ‘hybrid’ architectures such as parallel vector machines (where a cluster of SIMD vector processors are linked by a communication system to form a single MIMD system), but suggests a particularly simple programming model.
- The router is referred to as a ‘communications network’ [McColl, 1993a] or ‘communication system’ [Miller, 1993]. The original term implied an unnecessary bias towards message-passing architectures. In [Valiant, 1990a] it is noted that the router is “specifically intended for implementing storage accesses between distinct components”. On a shared-memory machine, these accesses are made directly by addressing non-local memory locations; here the function of the communication system is performed by the shared address and data bus.

- The notion of periodicity, with the tacit assumption that barrier synchronisation can be perfectly overlapped with computation and communication functions, has been replaced by “minimal time ... between successive synchronization operations” [Gerbessiotis and Valiant, 1992], or more simply “barrier synchronization cost” [Cheatham *et al.*, 1994]. This appears to be a more realistic model: on most current parallel machines, synchronisation must be performed by the communication system (distributed memory) or the processors (shared memory); even on systems which provide independent hardware for barrier synchronisation (for example, the Cray T3D), the barrier function implicitly includes the flushing of asynchronous memory requests through the communication system.

6.2 BSP cost models

The BSP model abstracts from many details of machine architecture. There is no notion of hierarchies of memory caches, or of locality within the communication system. The model does not distinguish explicitly between different network topologies for distributed-memory machines, or even between distributed memory and shared memory. The characteristics which determine the performance of any particular ensemble of hardware and software acting as a BSP machine are captured quantitatively in a small set of numerical parameters. Valiant [Valiant, 1990a] specifies three of these: the number of processors p , the synchronisation period L , and the ratio g of global computation capacity to global communication capacity. McColl [McColl, 1993a] introduces another: the speed s of individual processors. The present thesis proposes a refinement of g to a pair of parameters: the asymptotic computation/communication ratio g_∞ for ‘infinitely large’ data transfers, and the half-bandwidth data size $N_{\frac{1}{2}}$.

The BSP architecture parameters form the basis of a cost model for predicting the performance of parallel algorithms on different parallel machines. Or rather, they form the basis of a family of cost models: several variant models have appeared in the literature, reflecting the variety of purposes for which a cost model may be used, and differing implicit assumptions about the capabilities of actual parallel systems. There is general agreement on the need for three classes of parameters (processor count and speed, communication cost, synchronisation cost) corresponding to the three separate constituents of the abstract machine model (processor/memory components, communication system, synchronisation mechanism). The variation between models is in the precise definition of the parameters, and in the way that computation, communication, and synchronisation costs are

combined to estimate the cost of a superstep.

6.2.1 Communication cost

The cost of global communication (per unit of data communicated by each processor) is expressed by the BSP machine parameter g . It is common [Hey, 1991] to describe the communication power of parallel machines in terms of peak point-to-point bandwidth between a single pair of processors in megabytes per second. BSP practice differs from this custom in two important ways. First, because parallel efficiency is determined by the balance of computation and communication, the elapsed-time cost to communicate an item of data is stated not in absolute terms but in terms of the time taken by a ‘basic computation step’ performed on that data. Normalising time costs by processor speed in this way allows a more direct comparison of parallel performance between systems having processors of different power. For scientific computing it is convenient to consider the basic computation step to be one floating-point operation or *flop*, and the unit of data size to be one floating-point word.

Second, to account for the effect of message congestion in a communications network or of bus contention in a shared-memory multiprocessor, g expresses the cost of a global exchange of data, in which all processors may be participating simultaneously. Clearly the efficiency of such a global exchange can depend in an architecture-dependent way on the communication pattern. A conservative way to allow for this effect would be to use for g the upper bound for communication cost per word across all possible communication patterns. Unfortunately, for distributed-memory architectures with a significant element of communication locality, this worst case can be very bad indeed. However, theoretical results [Valiant, 1982, Valiant, 1990b] show that such worst-case behaviour can be avoided with high probability using *two-phase randomised routing*, which makes communication patterns on packet-switched networks more uniform by routing every point-to-point message through a randomly chosen intermediate point anywhere in the network.

With the assumption of randomised routing where appropriate, all communication patterns can be classified for cost purposes as *h-relations*, which Valiant [Valiant, 1990a] defines as “supersteps in which each component sends and is sent at most h messages”. This description can be read two ways: does each component send up to h messages and receive up to h others, or is h the maximum total of messages sent plus messages received? It appears that the first meaning is intended, as the same article refers to the routing of a permutation as a 1-relation, but *h-relations* are defined differently elsewhere.

Suppose we use s_i and r_i to denote the number of messages sent and received respectively by component i , with $0 \leq i < p$. The definition in [Valiant, 1990a] is equivalent to

$$h = \max(\max\{s_i \cdot 0 \leq i < p\}, \max\{r_i \cdot 0 \leq i < p\})$$

A later paper [Gerbessiotis and Valiant, 1992] repeats this definition, but suggests as an alternative

$$h = \max\{s_i \cdot 0 \leq i < p\} + \max\{r_i \cdot 0 \leq i < p\}$$

The choice of definition may depend on whether a lower or upper bound on cost is sought; in particular the lower bound can only be achieved if communication is *full-duplex*, so that components are able to send and receive simultaneously. In practice the communication system of many current-generation parallel machines is so fast that the effective bandwidth is limited by time taken by the processor in copying data between operating system and application program buffers. In this case a more realistic definition for an h -relation would use

$$h = \max\{s_i + r_i \cdot 0 \leq i < p\}$$

Formally, g is defined [Valiant, 1990a] such that an h -relation can be communicated in no more than gh time units, provided h is greater than some threshold value h_0 . Valiant states that

“This g can be regarded as the ratio of the number of local computational operations performed per second by all the processors, to the total number of data words delivered per second by the router.”

A possible argument for this equivalence is as follows: Suppose a particular BSP machine has p processors which can each execute s floating-point operations per second, and a communication system which has a global aggregate bandwidth of r words per second (that is, a total of r words can be delivered to all processors in a second). Then for a ‘random’ h -relation, each processor can use a share of up to $1/p$ of the global bandwidth to communicate its own h words, so the elapsed time overall should be no more than hp/r seconds, or the equivalent of $s(hp/r)$ flops. Thus we can consider g to be at most ps/r , which is the ratio of the global computation throughput of the machine to its global communication throughput.

The theoretical bound on g depends on having enough data to exchange so that the communication system can achieve the stated capacity. That is, the threshold

h_0 must be large enough for communications to be aggregated into messages of efficient size, amortising the start-up costs which make many short messages slower to communicate than a few longer ones.

6.2.2 Synchronisation cost

The BSP parameter L specifies the minimum amount of time (measured in basic computation steps) which must elapse between successive synchronisations. In the original BSP model with periodic synchronisations, this parameter can be adjusted to suit particular algorithms:

“The value of the periodicity L may be controlled by the program, even at runtime.”

The minimum size for a superstep is limited by the frequency with which barrier synchronisations can be performed on a particular hardware/software system. The maximum size is a property of the algorithm, since by definition a superstep can only contain as much work as can be done without waiting for an exchange of data between processors. For ‘embarrassingly parallel’ applications, where each processor can be given a self-contained task with no interaction at all, the whole program is effectively a single superstep. At the other extreme an iterative relaxation algorithm, where processors fetch updated values from their neighbours at each iteration, will not speed up at all by running in parallel unless there are at least L operations to perform per iteration, since each iteration must involve at least one superstep and therefore take at least the time of L flops.

In later versions of the model L becomes a property of the architecture, giving the minimum superstep size determined by the cost of performing a barrier synchronisation.

6.2.3 Combining the cost terms

In [Valiant, 1990a], it is stated that

“In a superstep of periodicity L , L local operations and a $\lfloor L/g \rfloor$ -relation message pattern can be realized.”

In other words, a superstep in which each processor performs a maximum of x local operations, and a global h -relation is communicated, can be assigned a cost

C (in basic computation steps) of

$$C = \max(L, x, gh)$$

This simple formula allows the constituent costs of synchronisation, computation, and communication to be considered independently in the complexity analysis of parallel algorithms. However, the assumption of independence appears unrealistic; for many architectures it is not the case that computation and communication can be perfectly overlapped. In [Gerbessiotis and Valiant, 1992] the cost for a superstep is given by an upper bound

$$C \leq \max(L, x + gh)$$

with the remark that the choice between the two combining formulas is “not crucial” if algorithms are designed so that every superstep contains either pure computation or pure communication. An even more conservative upper bound is used in [Bisseling and McColl, 1993, Cheatham *et al.*, 1994], equivalent to

$$C \leq L + x + gh$$

which allows for the necessity on most current parallel machines of using the communication system and/or the processors to realise a barrier synchronisation operation.

From the point of view of complexity analysis, where superstep cost calculations are used to prove asymptotic time bounds for algorithms as the number of processors or the problem size increases, the differences in the three combining formulas are insignificant, amounting at most to a ‘small constant factor’. For the purpose of software engineering, where an accurate prediction of execution time is needed for a given machine and problem size, a multiplicative factor of 2 or 3 may be important enough to compel a choice between combining formulas based on the architecture’s capabilities for overlapping of operations. However, such a choice will be significant only for an unusually well balanced algorithm. The relative difference between $\max(x, y)$ and $(x + y)$ is large only when x and y are approximately equal, while in practice most supersteps are likely to be dominated by one particular cost constituent: computation, or communication or synchronisation.

6.2.4 Scalability

Much of the elegant simplicity of the BSP model comes from the clear separation of processing from communication, and communication from synchronisation. However, this does not mean that the corresponding cost parameters are independent.

There may, for example, be an engineering tradeoff between the values of L and g achievable on a given architecture: the highest potential bandwidth will only be obtained if supersteps are long enough to contain large messages.

More important, although parameters L and g are used in formulas as if they were constants, on most real machines they are functions of p : both synchronisation and communications costs are likely to grow (sometimes alarmingly) with the number of processors. It might have been better to use the notation L_p and g_p as a reminder of this dependency.

For example, consider a two-dimensional mesh of processors with a packet-switched communication system. If we double the number of processors, the total number of communication links is doubled; but the number of separate links required to transmit a message between an arbitrary pair of processors also increases with the diameter of the network; *i.e.* by a factor of $\sqrt{2}$. Since the total point-to-point message capacity increases only as $\sqrt{p}$, while the CPU capacity increases linearly with p , the value of g for a mesh of p processors will be proportional to $\sqrt{p}$.

In contrast, a hypercube network adds more links per processor as the number of processors is increased. A hypercube with $p = 2^d$ processors has a total of $2^d d$ communication links, while the maximum path length for a message is d links; so the total point-to-point message capacity and the CPU capacity grow together as 2^d , and g is a constant independent of p .

The goal of constant L and g serves as a useful yardstick by which to evaluate manufacturers' claims of 'scalable' machines.

Chapter 7

A BSP programming model

The BSP computation model does not dictate any particular style of programming. Just as the traditional von Neumann computer allows the efficient implementation of many different programming language paradigms — imperative, functional, logic programming, and so on — the BSP computer can support a variety of parallel programming models. In particular, Valiant[Valiant, 1990a] contrasts two modes of BSP programming, according to the amount of control the programmer exerts over the partitioning of data between the processors: *automatic* mode, in which the locations of data objects are implicitly allocated to local memories using a pseudo-random scheme such as hashing, to simulate a single global address space; and *direct* mode, in which the programmer is responsible for the allocation of objects between processors, and non-local data references must explicitly identify the processor in whose local memory each object resides.

7.1 Design choices

This chapter introduces one particular example of a direct BSP programming model, intended as a minimal extension to the imperative model underlying familiar sequential languages such as FORTRAN, C and Pascal. In choosing a semantics for this extension, the primary design goals have been

1. simplicity
2. portability
3. efficiency

in that order of importance. In order to be useful in production applications programming, rather than an academic toy, the model must be capable of efficient implementation on current high-performance parallel computers. Efficiency must not be achieved at the expense of portability: architecture-specific details such as communication buffer size and the selection of routing and synchronisation protocols are optimisations to be performed by the implementation, and should not be visible at the programming level. The overriding consideration is simplicity: keeping the semantics as simple as possible and restricting the primitive operations to a minimal set helps to make programs more straightforward to write, read, and understand, and reduces the effort required in building and tuning multiple implementations of the programming model on different machine architectures.

The programming model is characterised by the following design choices:

Data parallel program structure. The need for large-scale parallelism seems most likely to arise from very large problems, which can only be solved in reasonable time by partitioning the data into pieces of manageable size and working on them concurrently. It is harder to imagine a computation so complex that the algorithm could be partitioned into thousands of distinct concurrent procedures. (We do not address the problem of constructing *reactive* programs, where control parallelism is dictated by a need to interact with a parallel external environment.)

SPMD program text. A data-parallel control structure is expressed naturally in the single program, multiple data programming style, where all processes are executing the same program text, but each has its own independent copy of program variables. The dynamic structure of the computation — proceeding sequentially through a series of phases, with concurrent operations on disjoint partitions of data within each phase — follows clearly the lexical structure of supersteps in the source program. The alternative *master-slave* paradigm can easily be accommodated by distinguishing one process as the controlling ‘master’, and making the code within each superstep conditional on the local process identifier; the corresponding actions of master and slaves are presented clearly together, interleaved in a single source text.

Direct mode of global memory management. All data is local to a particular processor; there is no implicitly shared global data space. The alternative *automatic mode* shows great promise as a basis for future high-level parallel languages; but its efficient implementation depends on values of L and g considerably smaller than those of most machines currently available.

Remote assignment semantics for data access between processors. Access to non-local data takes the form of a remote assignment operation, which indicates both the source of the data and its destination in one syntactic unit. We prefer this to the alternative choice of having symmetric *send* and *receive* operations performed explicitly by pairs of processes. Explicit message passing can be less efficient, requiring in a shared memory implementation the artificial introduction of extra buffering and per-message synchronisation. Separate sends and receives also lead to verbose applications code which is difficult to read, and to the risk of deadlock if the operations are not properly matched, or if system buffer space is exhausted. Because we specify remote assignments to be asynchronous, communication and synchronisation are decoupled, and deadlock is eliminated provided all processes follow the same superstep structure. However, shared access to remote data introduces the possibility of non-determinism, which the programmer must avoid by observing simple noninterference rules.

Static processor allocation. The number of processors taking part in a program is determined at the beginning of execution, and does not vary dynamically during the computation.

Single threaded process management. Within each processor there is a single sequential flow of control; in this paper we rather imprecisely use the terms ‘processor’ and ‘process’ interchangeably. Since an important aspect of the theoretical BSP model is *parallel slackness* — having more threads of control (virtual processors) than the number of physical processors — this can be considered a serious limitation. However, the provision of asynchronous communication primitives represents a kind of parallel slackness, as multiple remote memory operations can be overlapped with local computation in a single superstep.

7.2 Semantics of the programming model

In this section the semantics of the programming model is specified, using the formalism of CSP as defined in Hoare’s textbook *Communicating Sequential Processes*[Hoare, 1985]. We begin with the definition of sequential processes with variables and assignment, as defined in Section 5.5 of [Hoare, 1985].

7.2.1 Sequential processes

Let P and Q stand for any sequential process, x for a variable name, and e for an expression. A sequential process is one of

$SKIP$	empty process
$x := e$	assignment
$P; Q$	sequential composition
$P \triangleleft e \triangleright Q$	conditional
$e * P$	loop

We assume that all loops are finite, so we need not deal with divergence.

7.2.2 Supersteps

A data-parallel superstep can be generated from a sequential process by replicating it n times in parallel. We associate with each replicant process an index or *process-id* (an integer in the range $[0 .. n - 1]$), and give each replicant a private copy of all variables by systematically prefixing all variable names in assignments and expressions with the process-id. A superstep comprising n parallel copies of process P can be written as

$$\parallel_{i < n} P^{(i)}$$

where the relabelling operation is defined on processes by

$$\begin{aligned} (x := e)^{(i)} &= i.x := e|_i \\ (P; Q)^{(i)} &= P^{(i)}; Q^{(i)} \\ (P \triangleleft e \triangleright Q)^{(i)} &= P^{(i)} \triangleleft e|_i \triangleright Q^{(i)} \\ (e * P)^{(i)} &= e|_i * P^{(i)} \end{aligned}$$

and $e|_i$ represents the renaming of all variables occurring within the expression e by prefixing them with i .

In order to allow parallel processes within a superstep to follow different execution paths, we introduce a distinguished identifier *self* which evaluates to the local process-id. That is,

$$self|_i = i$$

7.2.3 Communication

The CSP $?$ and $!$ operators (*input* and *output*) require the simultaneous cooperation of a pair of processes to communicate a value. This behaviour is not suitable for a BSP programming model, which is intended to separate communication from synchronisation. Therefore communication within a BSP superstep will be defined by means of two new operators $?=$ and $!=$ (*fetch* and *store*), which act as asynchronous inter-process assignments. Because these operators are meaningless in a sequential process, they may only appear within an expression denoting a parallel superstep. They are defined by extending the equations for the relabelling operator:

$$\begin{aligned} (x \text{ ?= } p.y; Q)^{(i)} &= (i.x := p|_i.y) \parallel Q^{(i)} \\ (p.x \text{ != } e; Q)^{(i)} &= i.temp := e|_i; ((p|_i.x := i.temp) \parallel Q^{(i)}) \end{aligned}$$

Here x and y represent variable names, e and p are expressions, and $temp$ is a ‘fresh’ variable which does not occur elsewhere. Informally, $x \text{ ?= } p.y$ means “asynchronously fetch the value of the remote variable y from process p , and assign it to local variable x ”, and $p.x \text{ != } e$ means “compute the value of expression e using local variables, and assign it asynchronously to the remote variable x in process p ”.

To avoid the nondeterminism inherent in concurrent access to shared variables, the semantics of CSP makes the restriction that a parallel composition of sequential processes is valid only if neither process refers to any variable which is assigned to by the other process. Formally, $(P \parallel Q)$ is permitted only if

$$\begin{aligned} var(P) \cap acc(Q) &= \emptyset \\ acc(P) \cap var(Q) &= \emptyset \end{aligned}$$

where $var(P)$ is the set of all variables which could possibly be assigned to by any execution of process P , and $acc(P)$ is the set of variables which could be accessed (*i.e.* assigned to or referred to in an expression) by P . This rule requires a corresponding restriction on inter-process assignments in a BSP superstep. The superstep $\parallel_{i < n} P^{(i)}$ is valid only if

$$\forall i, j \in [0 .. n - 1] \cdot (i \neq j) \Rightarrow var(P^{(i)}) \cap acc(P^{(j)}) = \emptyset$$

Since access to nonlocal variables can only result from inter-process assignments, this condition can be guaranteed by the following rules:

- the source of a remote fetch, or the target of a remote store, must not be modified through an assignment or fetch by its owner or through a remote store by another process

- the target of a remote store must not be accessed in an expression by its owner, or fetched by another process

Because the asynchronous semantics of inter-process assignments introduces further parallelism, another condition is necessary to prevent fetches and stores from interfering with other operations within the same process. The sequence $P; Q$, where P is a fetch or store and Q is any process, is valid only if

$$\forall i \in [0 .. n - 1] \cdot \text{var}(P^{(i)}) \cap \text{acc}(Q^{(i)}) = \text{acc}(P^{(i)}) \cap \text{var}(Q^{(i)}) = \emptyset$$

This condition can be guaranteed for a superstep by the following rules:

- a process may not store to the same remote variable more than once
- a process may not store to and fetch from the same remote variable
- the target of a fetch may not subsequently be referenced in an expression

7.2.4 BSP programs

A BSP program is constructed from supersteps, with the same notation for sequential composition, conditionals and loops as we use to construct sequential processes. If P is a sequential process (possibly containing remote fetches and stores), we use the abbreviation $\{ P \}$ to denote the superstep $\|_{i < n} P^{(i)}$. The abstract syntax of a BSP program is one of

$\{ P \}$	superstep
$S; T$	sequential composition of supersteps
$S \triangleleft e \triangleright T$	conditional
$e * S$	loop

where P is a sequential process (not containing a nested superstep), and S and T are BSP programs. The meaning of a complete BSP program is defined by the semantic function BSP as follows:

$$\begin{aligned}
 BSP(\{ P \}) &= \|_{i < n} P^{(i)} \\
 BSP(S; T) &= BSP(S); BSP(T) \\
 BSP(S \triangleleft e \triangleright T) &= BSP(S) \triangleleft \bigwedge_{i < n} e|_i \triangleright BSP(T) \\
 BSP(e * S) &= (\bigwedge_{i < n} e|_i) * BSP(S)
 \end{aligned}$$

7.2.5 Barrier synchronisation

The BSP semantics given above corresponds to a direct, data-parallel view of the programming model. The execution of a BSP program proceeds synchronously through a sequence of supersteps, each of which is a parallel composition of n copies of a local computation. The top-level structure of the program has the form

$$(\parallel_i P1_i); (\parallel_i P2_i); (\parallel_i P3_i); \dots$$

Each parallel superstep implies the dynamic creation and termination of n processes. This is likely to be inefficient or even impossible on typical parallel computer systems. For more efficient implementation we would like to express the program structure instead as

$$\parallel_i (P1_i; P2_i; P3_i; \dots)$$

But the equation

$$(P_1 \parallel P_2); (Q_1 \parallel Q_2) = (P_1; Q_1) \parallel (P_2; Q_2)$$

does not hold in general. A counterexample is

$$\begin{aligned} P_1 &= x := 1 \\ P_2 &= y := 1 \\ Q_1 &= y := 2 \\ Q_2 &= x := 2 \end{aligned}$$

where the left-hand side results in (x, y) having the final values $(2, 2)$, and the right-hand side is nondeterministic.

The solution is to introduce an explicit shared event *sync* to serve as a global barrier synchronisation between adjacent supersteps. We now have the equivalence

$$(P_1 \parallel P_2); (Q_1 \parallel Q_2) = ((P_1; \text{sync} \rightarrow Q_1) \parallel (P_2; \text{sync} \rightarrow Q_2)) \backslash \text{sync}$$

provided, of course, that the P_i and Q_i do not themselves contain any *sync* events.

The use of explicit barrier synchronisation allows an alternative semantics for our BSP program notation, which can be described as control-parallel rather than data-parallel. With an eye to efficient implementation, we make one further restriction.

The data-parallel semantics of conditionals and loops at the outer level of a BSP program requires that the *true* branch be executed if and only if the conditional expression is true in all parallel processes. To avoid the extra communication and synchronisation cost of a global $\wedge$ -reduction, we stipulate that the constructs $P \triangleleft e \triangleright Q$ or $e * P$ are valid at the outer level of a BSP program only in contexts where the expression e can be guaranteed to have the same value in every parallel process. Thus each process need only evaluate its local copy of e .

The alternative control-parallel meaning of our BSP program syntax is given by the semantic function BSP' below.

$$\begin{aligned}
 BSP'(S) &= (\|_{i < n} B_i(S)) \setminus \{sync\} \\
 B_i(\{ \mid P \mid \}) &= P^{(i)}; sync \rightarrow SKIP \\
 B_i(S; T) &= B_i(S); B_i(T) \\
 B_i(S \triangleleft e \triangleright T) &= B_i(S) \triangleleft e|_i \triangleright B_i(T) \\
 B_i(e * S) &= e|_i * B_i(S)
 \end{aligned}$$

Chapter 8

The Oxford BSP library

The appearance of a new model of parallel computation represents an opportunity to create an entirely new parallel programming language [McColl, 1993b, Cheatham *et al.*, 1994]. A more conservative goal is to extend familiar sequential languages with a minimal set of new operations with which to write parallel programs in bulk synchronous form. Rather than making syntactic additions to existing languages, our approach is to design a small library of subroutines [Miller and Reed, 1993] for the primitive operations of process creation, remote data access, and bulk synchronisation. As our main application domain is numerical and scientific programming, the most common choice of sequential language is FORTRAN; but the library has also been used with programs in C and Pascal.

While the specification of the primitive operations is portable, implementations of the library are not necessarily so. Instances of the library have been created for different parallel platforms, ranging from workstation clusters to shared memory multiprocessors and massively parallel supercomputers, using internally the disparate facilities for parallelism native to each environment, but providing the same interface and BSP semantics to the applications program.

8.1 Routines in the library

The core of the library consists of three pairs of routines, corresponding to the three major components of a BSP abstract machine: processors and memory, communication, and synchronisation. The syntax described below is for FORTRAN applications; the interface to other languages is very similar, and is described in Appendix A.

8.1.1 Process management

The execution of program text P as the SPMD parallel program defined as $BSP(P)$ in our semantic model is achieved by making a call to

```
bsp_start( maxprocs, numprocs, mypid )
```

at the beginning of the program, and

```
bsp_finish( )
```

at its end. The input parameter `maxprocs` represents the maximum number of processors to be used for the run. The number of processors actually allocated is returned in `numprocs`, and may be less than the number requested (but is guaranteed to be at least 1). A special case of `maxprocs=0` acts as a request to use all available processors. The number of processors available is determined in a platform-dependent way, either at execution time (*e.g.* by a user-specified shell environment variable, or a file containing a list of host names), or earlier (as part of the compilation or configuration of the program).

The result of `bsp_start` is that copies of the calling program begin to execute in parallel on `numprocs` different processors, with the value of `mypid` on each one set to the process identifier for that process (an integer between 0 and `numprocs-1`). At the end of the computation, each process must call `bsp_finish`; when all processes have done so, the whole program terminates.

8.1.2 Communication management

The implementation of the remote assignments $x \text{ ?= } pid.y$ and $pid.x \text{ != } e$ is by means of the subroutine calls

```
bsp_fetch( pid, fromdata, todata, size )
```

and

```
bsp_store( pid, fromdata, todata, size )
```

In each case, `fromdata` and `toata` identify the source and destination data objects (variables, array names, or indexed array expressions in FORTRAN; pointers in C). A total of `size` bytes will be transferred.

Note that this simple subroutine interface represents both local and remote data objects by their memory address. This design imposes a limitation on current versions of the library: variables which are accessed remotely must exist at the same (virtual) address in every process. This condition will be satisfied for statically allocated variables in C and FORTRAN; but to enable the use of dynamic or stack-frame variables in general will require a more sophisticated interface.

Both `bsp_fetch` and `bsp_store` are asynchronous. While the routines return con-

trol immediately to the calling program, the data may not actually reach its destination until later. In a shared memory implementation, it is possible that remote assignments actually copy data directly from one process to another; but in general these operations are only guaranteed to be completed after the end of the enclosing superstep.

8.1.3 Synchronisation management

The superstep $\{ P \}$ in a BSP program is identified by bracketing the text of P by a call to

```
bsp_sstep( stepid )
```

at its beginning, and

```
bsp_sstep_end( stepid )
```

at its end. The `stepid` parameter serves to identify different supersteps in the program for consistency checking and error reporting: each separate superstep in the source code should be identified with a different value for `stepid`, and matching pairs of `bsp_sstep` and `bsp_sstep_end` must have the same `stepid`.

Execution on all processors must proceed through all the supersteps in the same order. This implies that if a superstep is written inside a loop or `if` statement, the loop control variable or conditional expression must have the same value in all processors. Within a superstep, each processor may perform different computations, follow different control paths, and even call different subroutines, but the library implementation ensures that every processor has reached the end of the current superstep before any one can proceed to the next: the `bsp_sstep_end` routine contains a global barrier synchronisation.

Note that any code executed between the `bsp_sstep_end` of one superstep and the `bsp_sstep` of the next can itself be considered to be a separate ‘implicit’ superstep. Although such an implicit superstep is communication free (the remote assignment primitives are only allowed to be used from within an explicitly bracketed superstep), semantically there must be another barrier at its end: in case, for example, a local variable modified in an implicit superstep is fetched by another process in the succeeding explicit superstep. In library implementations using direct shared memory, there will be an actual synchronisation within the `bsp_sstep` routine; in some other implementations this synchronisation would be redundant and is not performed.

8.1.4 Noninterference rules

Because arbitrary concurrent accesses to non-local data are a potential source of error, there are important restrictions on the use of the communication primitives. The superstep structure of a BSP program provides the discipline which makes remote assignments safe to use. It is the responsibility of the programmer to ensure that reads and writes referring to the same data object, or multiple writes (of different values) to the same data, are separated by a superstep barrier, by observing the following rules:

- The destination data object of a `bsp_fetch` or `bsp_store` may not be examined or modified by any process during the same superstep — that is, it may not be the source or destination of any other `bsp_fetch` or `bsp_store` or appear in a local expression or assignment.
- A data object which is modified locally by a process (*e.g.* the target of an assignment or input statement) may not be fetched by any other process in the same superstep.

As a particular instance of the first rule, a process which fetches a remote data object may not use the result until the next superstep. If a parallel program is written without observing the consistency rules, the result may be non-deterministic, since there is no guaranteed ordering on communications.

8.1.5 Higher level operations

Besides the six core routines, the library contains an extensible set of higher level operations which capture common patterns of parallel computation. The initial release of the library includes only the `broadcast` and `reduction` operations. Portable versions are implemented in terms of the basic library primitives, and are performed by scalable algorithms taking no more than $\lceil \log p \rceil$ supersteps on p processors. Where appropriate, alternate versions are written to use native facilities of the underlying platform to achieve higher efficiency.

8.2 BSP library implementations

8.2.1 Generic message-passing versions

The first prototype of the BSP library was built on top of the PVM message-passing system [Geist and others, 1993]. Process management in the prototype library uses PVM daemons to initiate processes on multiple hosts. Remote assignments are implemented by means of simple ‘active messages’: `bsp_store` sends the remote process a message containing the destination address and the data to be stored there; `bsp_fetch` sends a message containing both the source and destination address, which the remote process uses to construct in reply a `bsp_store` message containing the requested data. For efficiency, requests are implicitly buffered so that multiple BSP requests can be aggregated into a single PVM message. At the end of a superstep, all processes exchange synchronisation messages, and then each process waits within the `bsp_sstep_end` routine for any outstanding `fetch` replies.

Because PVM is itself a portable software package, the prototype provided a library implementation which could run on a variety of parallel systems, although its performance was less than optimal. The next step was to use the same overall active message strategy as the basis for a family of library versions, using different underlying message-passing systems for better performance in different parallel environments.

The `tcp` version uses the TCP Internet Protocol for communication between networked workstations. By accessing TCP/IP directly through the UNIX socket interface, and using the `rsh` command to initiate remote processes, the overheads of PVM and the complication of multiple daemon processes are avoided. This version of the library is not highly scalable, because of limitations on the the number of open sockets, and because networks based on ethernet or token-ring design behave essentially like a shared bus, with global communication cost g increasing linearly (or worse) with the number of processors.

The `parmacs` version uses the portable PARMACS 6.0 library as an alternative to PVM. Other parallel software platforms, such as the proposed standard MPI, could be used in a similar way. A simple interface between the generic BSP implementation and the underlying platform means that an initial version of the library for a new message-passing system can be produced very quickly. Further efforts in adapting the library to make more effective use of the idiosyncrasies of each particular platform can improve parallel performance, without requiring any system-specific tuning of applications code.

For multiprocessor systems with shared memory, the **shm** version of the library uses a shared buffer allocated by the System V **shmat** system call to pass active messages between processors. This library version is also useful for emulating a parallel environment on a single workstation, so that parallel applications can be developed and debugged on a personal machine before moving to a more expensive parallel system for production use. Barrier synchronisation is achieved with semaphores, using either the System V kernel **semop** facility, or user-process spin locks. The latter choice greatly reduces message latency on a multiprocessor by avoiding unnecessary context switching, but may consume an unfair share of CPU resources in a multi-user environment.

8.2.2 Native versions

More highly optimised versions of the library have been created for specific parallel machines, by using vendors' proprietary interfaces to implement the BSP communication and synchronisation primitives. The BSP library for IBM's SP1 and SP2 distributed-memory machines uses MPL — the native message-passing library which forms part of the IBM Parallel Operating Environment — for efficient access to the high-speed switch. A special virtual memory implementation for Silicon Graphics' shared memory machines takes advantage of the ability of IRIX processes to map directly into one another's address space, performing fetch and store operations as a simple memory-to-memory copy without interrupting the remote process. On Cray's massively-parallel T3D, the highly efficient native **shmem** library for shared memory access has a semantic model which matches the BSP primitives almost perfectly, so that **bsp_fetch** and **bsp_store** translate directly into **shmem_get** and **shmem_put**, and superstep barrier synchronisation is performed by the T3D hardware.

Native BSP libraries for other high-performance parallel machines are planned or in preparation. Using these, application programmers can gain much of the performance advantage of vendor-specific architectures and software interfaces, without sacrificing single-source portability.

8.3 Library performance

8.3.1 Communication granularity

During experiments to measure the performance of implementations of the BSP library in order to estimate values for L and g for different machines, it became clear that a refinement to the g parameter was required if the cost model is to be used to predict the performance of actual programs as well as for theoretical complexity bounds.

According to the original model, the cost of a communication superstep in which each process stores h words of data to other processes should be $g \times h$, regardless of whether it is performed by a single `bsp_store` of h words or by h `bsp_stores` of one word each. In practice the time taken by the two operations may be quite different, because of the overhead involved in each call to `bsp_store`. Note that this overhead does not include the latency or ‘start-up’ cost of the actual communication, because message-passing versions of the library can combine multiple store requests, buffering them implicitly into one message per superstep whose latency cost is already included in L . But the overhead per store does include the CPU time for the subroutine call itself (a significant cost on shared-memory machines where the actual remote data transfer can be performed in a single instruction), and for distributed-memory versions also includes the cost of address information and other active-message headers for each separate store operation.

To quantify the effect of data granularity on communication cost, we adopt the simple linear model proposed by Hockney [Hockney, 1991], in which the bandwidth r depends on message size n according to the formula

$$r = \frac{r_\infty}{1 + N_{\frac{1}{2}}/n}$$

The parameters r_∞ and $N_{\frac{1}{2}}$ represent the asymptotic bandwidth for infinitely large messages, and the ‘half-bandwidth’ message size which allows half the peak bandwidth to be achieved. The start-up time t_0 per message is given by the ratio

$$t_0 = \frac{N_{\frac{1}{2}}}{r_\infty}$$

We can refine the BSP cost model by replacing the fixed parameter g by a function g_n which depends on data granularity n . We denote by g_∞ the asymptotic minimum g (for infinitely large stores), and the worst-case g_1 is equivalent to the g of the original model, giving an upper bound on communication cost for any

granularity. Since the communication cost per word is inversely proportional to bandwidth, for a given size n of objects stored or fetched we estimate

$$g_n = g_\infty \times \left(1 + \frac{N_{\frac{1}{2}}}{n}\right)$$

8.3.2 Performance measurement

A simple benchmark procedure was devised to estimate the BSP parameter values for some actual parallel machines. The benchmark consists of several pure computation and pure communication supersteps, whose elapsed time is measured with the `bsp_time` library routine, which uses the highest-precision hardware clock available on each platform.

The benchmark procedure is as follows:

- Measure the time t to compute the dot product of two n -element double precision vectors, where n is large enough so that the vectors do not fit within the first level memory cache of the machine. Estimate $s = (2n - 1)/t$
- Measure the time t_n for a superstep in which each process performs a single `bsp_store` of n double precision words to one other process, using a random permutation of process-ids. Repeat for different values of n , and use a linear regression to fit the measured t_n values to the equation

$$t_n = t_0 + n/r_\infty$$

where t_0 is the superstep synchronisation time and r_∞ the asymptotic pairwise bandwidth in words per second. Estimate $L = s t_0$, and $g_\infty = s/r_\infty$

- Let N be the number of words per `bsp_store` which gave the highest bandwidth in step 2. Measure the time t_k for a superstep in which each process performs N/k separate `bsp_store` operations of k words each, repeating for different values of k to find the smallest value for which

$$t_k \leq t_0 + 2N/r_\infty$$

Estimate $N_{\frac{1}{2}} = k$

This benchmark procedure has been performed using Version 1.1 of the library on the following systems:

RS6K	a cluster of IBM RS/6000 model 350 workstations, on a token ring, using the <code>tcp</code> library version
SP1	an IBM SP1, using the <code>mp1</code> library version on the high-speed switch
SGI	a Silicon Graphics Power Challenge, using the direct shared memory <code>vm</code> library version
T3D	a Cray T3D

In Table 8.1, the measured processor speed s (in megaflops per second) is rather less than each machine's advertised 'peak' megaflop rate, but should be more representative of actual sustainable speed in applications which have not been tuned for the highest memory cache performance.

Machine	s	p	L	g_{∞}	$N_{\frac{1}{2}}$
RS6K	17	2	67000	150	4
		4	130000	300	
SP1	25	2	7400	58	11
		4	13000	59	
		8	18000	60	
SGI	75	2	900	12	21
		4	1600	12	25
T3D	10	2	240	1.1	37
		4	250	1.1	37
		8	240	1.5	28
		16	300	2.1	20
		32	270	2.5	16
		64	250	3.0	14
		128	240	3.3	12
		256	360	4.0	10

Table 8.1: Estimated BSP Parameters

Chapter 9

Parallel molecular mechanics

9.1 Software acceleration

One of the obstacles to the conversion from sequential to parallel computers for scientific applications is the prevalence of ‘dusty decks’: large, poorly-structured, often poorly-understood programs, which nevertheless have great strategic value as production tools. The most effective way to make use of the new parallel machines is probably to write new programs from the original specifications, using parallel algorithms and modern software engineering techniques. However, many organisations are understandably reluctant to invest the time and effort in completely rewriting software which already works (mostly), when all they want is to make it go faster. Besides, in programs developed incrementally by many hands over many years, there may well be no real specification: the only description of the behaviour of the program is the source code.

A tempting alternative to the complete redesign of programs for parallel execution is a selective ‘software acceleration’ procedure: identifying the computationally-intensive kernel of the program which is responsible for most of the processing time — a common rule of thumb is that a typical program spends 90% of its time executing 10% of its code — and rewriting only that part to execute in parallel, while leaving the rest of the original program unchanged.

There is a limit to the benefits of selective acceleration: *Amdahl’s Law* [Amdahl, 1967] says that the more we accelerate the parallel part of the program, by adding more processors or using more efficient communication mechanisms, the more the total execution time will come to be dominated by the remaining sequential part. Suppose, for example, we rewrite for parallel execution the part of a program which is responsible for 90% of its processing time. Even on a machine with an infinite

number of processors, and infinitesimally small values of L and g , the sequential 10% takes the same time as before, so that we cannot speed up the whole program by more than a factor of 10. To make effective use of a massively parallel computer with hundreds of processors, it is clear that the selective rewriting approach will be practical only for programs whose execution time is very strongly dominated by a small fraction of the program text with a high degree of potential parallelism.

9.2 AMBER molecular mechanics

AMBER 4.0 [Pearlman *et al.*, 1990] is a suite of molecular modelling programs written in FORTRAN widely used by chemists for studying the conformations of large molecules. The **minmd** module of AMBER is used to perform molecular mechanics calculations, combining the functions of energy minimisation and molecular dynamics. In the former, a standard minimisation algorithm such as the nonlinear conjugate gradient method is used to search for the ‘best’ or lowest-energy conformation of a given molecular structure. In the latter, Newtonian laws are used to calculate the motions of atoms in a molecule (*in vacuo* or in solution), in order to simulate the molecule’s dynamic behaviour. In both applications, the forces which atoms exert on each other are calculated according to the semi-empirical ‘AMBER force field’ model [Weiner *et al.*, 1984, Weiner *et al.*, 1986], which forms the computational heart of **minmd**. The work involved in fully evaluating the force field increases with the square of the number of atoms (since electrostatic and van der Waals forces exist between every pair of atoms). For biochemists working with protein molecules containing thousands of atoms, this is an expensive proposition which can involve hours or days on a supercomputer. In practice, the computation time is reduced by imposing a ‘cutoff’ on the distance for which long-range forces are evaluated. Because atoms must be packed into physical space, there is a constant upper bound on the number of neighbours within the cutoff distance for each atom, so that the force field computation for n atoms is reduced from $O(n^2)$ to $O(n)$ time, at the expense of some accuracy in the final result.

9.3 Execution profiling

To identify the computationally-intensive kernel of **minmd**, test runs of the sequential version of the program were performed on several sets of data and analysed with the UNIX run-time profiling tool **prof**, which uses a clock sampling technique

Routine	Sec	%

Pairlist	14.32	.89
Nonbond	1514.19	94.57
Bond	.00	.00
Angle	.51	.03
Dihedral	2.49	.16
Shake	49.28	3.08
Pol. der	.00	.00
Efielde	.00	.00
Indip	.00	.00
Matmul	.00	.00
Other	20.28	1.27

Total	1601.09	.44 Hours

Table 9.1: Timing: Sequential AMBER on RS/6000

to estimate the amount of time spent in each subroutine. A coarser view of the distribution of execution time is given by the AMBER output itself, in a final report summarising the time taken by each major component of the program.

Table 9.1 shows the result of a brief 100-step timing run on an IBM RS/6000 workstation. The test data, supplied by Dr. Chris Edge of Oxford University's Glycobiology Unit [Woods *et al.*, 1994], were for the dynamic simulation of an oligosaccharide in water, comprising a total of 10,704 atoms. A cutoff distance of 10Å was used, limiting the calculation of long-range forces to 2,376,022 pairs of non-bonded atoms.

The timing results show that the non-bonded force calculation accounts as expected for nearly 95% of execution time. Thus a reasonable improvement in speed on small-scale systems should be achievable by parallelising only the single subroutine responsible for the non-bonded force calculation. For scalable performance on larger systems, Amdahl's law requires conversion of other routines as well. In particular, 3% of execution time is expended on the SHAKE algorithm [Ryckaert *et al.*, 1977], which constrains bond lengths to equilibrium values in order to allow a larger simulation timestep by eliminating high-frequency vibrations. Without running SHAKE in parallel, the maximum possible speedup for the test problem would be limited to a factor of 33. Unfortunately SHAKE appears to be inherently sequential, involving an iterative adjustment of bonded pairs of atoms in order of

their position in the bond list, repeating the procedure until all bond lengths have converged within a specified tolerance. This algorithm has been reported [Ellingworth, 1989] to be notoriously difficult to parallelise for distributed-memory machines.

9.4 Force field evaluation on transputers

In an earlier exercise [Miller, 1990], the author created a parallel version of the minimisation module of a previous release of AMBER, in which the force field evaluation routines were rewritten in `occam` for a network of INMOS transputers. In this approach, the transputers are used as a force field calculation engine, communicating via a simple remote-procedure-call interface with the rest of the AMBER code on a Sun workstation host. Although the parallel force field code itself is fairly efficient (one evaluation for a 4,628-atom molecule with 2,312,820 non-bonded pairs takes 0.303 seconds on $80 \times \text{T800}$ transputers, compared with 1.065 seconds on $20 \times \text{T800}$), the master-slave programming model sets a limit on the overall efficiency of the program:

- For each force field evaluation, $3n$ words of coordinate information and $3n$ words of energy gradient must be communicated between the master and slaves. The interface between the master and slaves via a single transputer link adapter on the Sun VMEbus is a bottleneck, with much higher latency and lower bandwidth than the mesh of links joining the slaves.
- Speedup is limited by the sequential code running on the host, which includes SHAKE and the minimisation and dynamics integration algorithms. This limit would become even more serious if the T800's were replaced by more modern, faster processors.

Further, the approach is overly specialised to one particular architecture by the use of transputer-specific channel communication and `occam`: the code would require substantial rewriting for any other parallel machine.

9.5 Bulk-synchronous parallel AMBER

To investigate the applicability of the Oxford BSP library in a large-scale production application program, a new parallel adaptation of AMBER `minmd` was undertaken.

- The program uses an SPMD structure: each processor runs all of **minmd** with a complete replicated copy of almost all data including coordinates of all atoms. The only exception is the non-bonded pairlist (a list of indices of pairs of atoms separated by less than the cutoff distance), which is partitioned each time the pairlist is rebuilt (typically every 25–50 timesteps) to give each processor an approximately equal number of pairs.
- One distinguished processor performs all input and output. Input data is distributed to the other processors with **bsp_broadcast** in $\lceil \log p \rceil$ supersteps.
- All components of the force field are evaluated in parallel in a single BSP superstep. For the non-bonded long-range forces, each processor calculates energy and gradient values for pairs of atoms in its segment of the pairlist. For the other forces, each processor is assigned an equal share of bonds, angles, and dihedrals. At the end of this calculation, the contributions of each processor to the total energy and to the global gradient vector are summed and the result distributed to all processors. This operation is accomplished with a single call to **bsp_reduce**, taking $\lceil \log p \rceil$ supersteps on p processors.
- The SHAKE procedure is also run in parallel, with each processor responsible for adjusting the positions of an equal-sized segment of atoms. The parallel strategy is based on an idea found in [DeBolt and Kollman, 1993], where a much more complicated parallel SHAKE is described. The key idea is that the order in which bonds are adjusted is not critical: any ordering should be equally acceptable provided all bonds converge to their equilibrium values. During each iteration of our version of SHAKE, each processor first adjusts all its ‘interior’ bonds (for which it owns the atoms at both ends). In one communication superstep, pairs of processors exchange the new coordinates for all ‘halo’ atoms (at the ends of bonds which cross processor boundaries). Then the shared bonds are adjusted, with a redundant calculation by pairs of processors which own the atoms at each end. Finally, the values of a logical variable in each process signifying local convergence are combined and distributed with a **bsp_reduce** to determine whether the global iteration has converged.
- The computation of the non-bonded atom pairlist and the integration of the equations of motion are also run in parallel.
- The rest of the code is executed redundantly on all processors.

9.6 Parallel performance

The dynamics problem described in Section 9.3 was used as a benchmark for the parallel **minmd** on machines representing four different classes of parallel system:

- RS6K — IBM RS/6000's with a token-ring (workstation cluster)
- SP1 — IBM SP1 with a high-speed switch (distributed memory)
- SGI — Silicon Graphics Power Challenge (shared memory)
- T3D — Cray T3D (massively parallel)

Table 9.2 gives total elapsed time for the run in seconds (excluding disk I/O) in each case, with the CPU time for the original sequential AMBER 4.0 on a Cray Y/MP for comparison.

While the T3D's low L and g values permit high parallel efficiency (the superlinear speedups for four and eight processors result from improved cache-hit ratio), the benchmark problem is too small for extended scalability on a massively parallel machine. To test the effect of an increase in problem size on the T3D, the nonbonded pair cutoff was increased to 15Å, giving 7,924,551 pairs. Since the increased problem required more memory than was available on a single processor, the speedup figures in Table 9.3 are relative to an estimate of single-processor runtime.

Machine	Processors	Time	Speedup
RS6K	1	1600.67	1.0
	2	925.73	1.7
	4	663.16	2.4
SP1	1	1077.35	1.0
	2	586.55	1.8
	4	326.26	3.3
	8	202.47	5.3
SGI	1	431.86	1.0
	2	221.97	1.9
	4	115.58	3.7
T3D	1	2239.92	1.0
	2	1165.10	1.9
	4	543.69	4.1
	8	276.61	8.1
	16	142.95	15.7
	32	76.53	29.3
	64	44.30	50.6
	128	27.27	82.1
	256	19.92	112.4
Y/MP		223.9	

Table 9.2: BSP AMBER Benchmark Times

Machine	Processors	Time	Speedup
T3D	8	871.33	8.0
	16	449.48	15.5
	32	229.28	30.4
	64	120.16	58.0
	128	65.38	106.6
	256	38.24	182.3
	512	28.62	243.6
	1024	21.40	325.7
Y/MP		602.9	

Table 9.3: BSP AMBER Benchmark Times — Extended Cutoff

Chapter 10

Conclusions and future work

The work reported in this thesis includes a number of original contributions:

- a formalisation of the semantics of multidimensional rectangular arrays as a constructive data type, avoiding the asymmetry caused by the ordering or hierarchy of dimensions in other models;
- a refinement to the BSP cost model to quantify the effect of data granularity on communication cost;
- a simple SPMD programming model for BSP computation, with a formal semantics defined in CSP;
- the specification, design and implementation of a library of primitives for architecture-independent BSP programming;
- a portable and scalable parallel adaptation of the UCSF AMBER 4.0 molecular dynamics and minimisation module, based on the BSP library.

10.1 Further development of the array theory

10.1.1 Partial homomorphisms

Although the formal presentation of arrays defines homomorphisms as total functions, we have sometimes used the homomorphism construction to ‘define’ operations which are in fact only partial. The obvious example is $(+ /)$ — we can only flatten an array if its elements are arrays with shapes that are conformant along each dimension.

A partial homomorphism $h = \mathcal{H}(f, \oplus)$ satisfies a weaker form of the homomorphism equations:

$$\begin{aligned} h \cdot [\cdot] &\sqsubseteq f \\ h \cdot \#_i &\sqsubseteq \oplus_i \cdot (h \times h) \end{aligned}$$

where $\sqsubseteq$ is a *refinement* relation on functions

$$(f \sqsubseteq g) \iff \forall x, y ((f x \simeq y) \Rightarrow (g x \simeq y))$$

Note that the direction of the refinement is critical: we want $h(x)$ to be defined only if $h(u) \oplus_i h(v)$ is defined for every possible splitting of x into $(u \#_i v)$. To maintain uniqueness we define $\mathcal{H}(f, \oplus)$ to be the ‘most defined’ of all the functions h which satisfy the above equations.

It turns out that we can calculate the domain of a partial homomorphism on arrays: $\mathcal{H}(f, \oplus)x$ is defined if and only if

- every element of x is in the domain of f
- $f a \oplus_i f b$ is defined for every pair of elements a, b which are *adjacent* along dimension i

To formalise and prove this, we need a weaker form of the Unique Extension property.

10.1.2 The empty array

Many definitions and calculations would be made simpler if we had an *empty* array $[]$, which would act as a common identity element for all the join constructors. However, such an object would not obey the existing array axioms: for example, if x and y are non-conformant arrays, then $(x \#_i [])$ and $([] \#_i y)$ are defined, but $(x \#_i [] \#_i y)$ is not — a violation of the associativity of the join constructors.

Adding the empty array consistently to the theory would require that

- the associativity and abiding laws must be weakened by excluding the empty array
- the array homomorphism $\mathcal{H}(f, \oplus)$ must have an extra defining equation

$$h [] = e$$

where e is an identity element for all the $\oplus_i$

10.2 Further development of the BSP library

Besides the obvious tasks of building new native versions of the BSP library, and improving the performance of existing versions, a number of extensions to the library's capabilities are worth investigating, which include:

- reducing L by allowing nested supersteps, with barrier synchronisation between inner supersteps performed by a subset of the processes taking part in the enclosing superstep
- reducing $N_{\frac{1}{2}}$ by defining aggregate assignment primitives such as *gather* and *scatter* for constant-stride array slices and vector-indexed lists
- introducing a mechanism for remote assignments to dynamically-allocated objects, perhaps by an executable 'declaration' implemented by an exchange of pointers between processors

There is also plenty of scope for the development of tools for performance monitoring and debugging, using the BSP superstep as a natural unit of granularity for program execution.

10.3 Integrating the two approaches

The most inviting direction of future research lies in the gap between the high-level, abstract, implicit parallelism of constructive arrays and the low-level, concrete, explicit parallelism of the BSP implementation.

Building from the top downwards, the next step is to use the BSP model to provide an architecture-independent parallel operational semantics for the basic operations and combining forms of the array algebra. The BSP semantics will provide a 'cost calculus' as proposed by Skillicorn [Skillicorn, 1993], to be used for guidance in the algebraic transformation of abstract algorithms into a more concrete, parallel form. By incorporating the BSP architecture parameters explicitly in the resulting parallel algorithms, it should be possible to predict their performance across a range of implementations. More ambitiously, the operational semantics could serve as a basis for compilation of a high-level implicitly-parallel language — an applicative analogue of data-parallel languages such as High Performance FORTRAN [High Performance Fortran Forum, 1993].

Building from the bottom upwards, the BSP library needs a much more comprehensive set of higher-level routines to extend the existing *broadcast* and *reduce*,

capturing common patterns of collective communication and combining forms used in parallel applications programs. Results from the array algebra can be used to guide the choice of operations to be added, and to specify their semantics. While the higher-level routines can be implemented in terms of the core BSP library primitives, and adapted automatically to different architectures by explicit parameterisation with p , L , and g , alternative versions (with the same semantics) can also be produced to make the most effective use of specific machines which have native facilities such as hardware for broadcasting and combining of messages.

Appendix A

The Oxford BSP Library Reference Manual

A concise reference to the Oxford BSP library is given by a set of manual pages created for use with the UNIX operating system's **man** command. The following manual entries are reproduced here:

Section 1 (User Commands)

<code>bsp</code>	Instructions for compiling and running programs using the BSP library on different parallel platforms.
<code>bspcc</code>	A shell command script to compile C programs using the BSP library.
<code>bspf77</code>	A shell command script to compile FORTRAN programs using the BSP library.
<code>bspprof</code>	An awk script to display the results of superstep execution time profiling.

Section 3 (Library Routines)

<code>bsp</code>	The six BSP primitives.
<code>bsp_broadcast</code>	Broadcast data to all processes.
<code>bsp_reduce</code>	Combine data from all processes by reduction with a binary operator.
<code>bsp_time</code>	Elapsed time measurement with a high-precision clock.

NAME

bsp – compiling and running parallel programs using **bsp(3)**

INTRODUCTION

The Oxford BSP library provides a concise set of basic operations for architecture-independent parallel programming according to the *bulk-synchronous parallel* model of computation. The primitives defined in **bsp(3)** allow the identical application source code to be used with any parallel hardware and software platform for which a library version exists. However, the logistics of compiling and running a BSP program depend on the parallel environment, as described below.

TCP/IP DESCRIPTION

Generic version using TCP network protocol. Compile with **-tcp** (the default for Sun, RS/6000, Silicon Graphics and other workstations). To run a program:

1. Set shell variable **BSP_HOSTFILE** to the pathname of a file (default **host.list**) containing the names of hosts to run on (one per line). You can use a PVM hostfile for this purpose (comments and PVM options are ignored). BSP process 0 always runs on the local host, whether or not it is named in the hostfile. Host machines must all have the same architecture: heterogeneous clusters are not supported by the present implementation. If hosts support multiple IP interfaces (for example both ethernet and token-ring on RS/6000), don't forget to use the hostname alias corresponding to the fastest interface.
2. Because **rsh(1)** is used to spawn remote copies of the program, the local machine must be named in the **.rhosts** file for each remote machine. Note that this is a security risk: anyone who can illicitly use your account on the local machine will have access to your account on the remote machines as well.
3. Set shell variable **BSP_PROCS** to the number of processes to run (if unset, it defaults to the number of hosts in the hostfile).
4. Optionally, set shell variable **BSP_CHDIR** to the pathname which remote processes are to use as current directory. If unset or null, it defaults to your login directory on the remote machine. If set to an absolute pathname (beginning with **/** that pathname will be used. If set to a relative pathname, it will be prefixed by the name of the current working directory on the *local* machine (as determined by the **pwd(1)** command). For example, if **BSP_CHDIR=.** remote processes will start in the same current directory in which process 0 is running.
5. Start the program. Remote processes inherit arguments but not environment variables.

PVM DESCRIPTION

Generic versions using PVM. Compile with **-pvm**, **-pvm2**, or **-pvme**. To run a program:

1. Follow the usual procedure for creating a PVM hostfile and starting daemons.
2. Set shell variable **BSP_PROCS** to the number of processes to run (if unset, it defaults to the number of active hosts with the same architecture as the local machine).

3. Start the program. Remote processes do not inherit arguments or environment variables. Remote processes run in the directory specified by PVM (normally your login directory on the remote machine). In PVM3, standard output from the remote processes is redirected to the file `/tmp/pvml.userid` on each remote machine. Currently it is not possible to start PVM3 BSP programs from the PVM console.

PARMACS DESCRIPTION

Generic version using Parmacs V6. Compile with `-mpl`. To run a program, follow the procedure for starting a Parmacs program on your parallel machine, naming the same BSP executable file as both host and node program. On the SP1, Parmacs uses the MPL interface, so a program can be started as described under the MPL below.

MPL DESCRIPTION

Native version for IBM Scalar POWERParallel machines. Compile with `-mpl` (the default for SP1). To run a program:

1. Set shell variable `MP_PROCS` to the number of processes to run (if unset, it defaults to 1). Currently the maximum allowed is one process per node.
2. Start the program. Remote processes inherit arguments and environment, and run in the current directory.

SHM DESCRIPTION

Generic version using `shmat(2)` to create a shared-memory buffer between processes, on a uniprocessor or a shared-memory multiprocessor. Compile with `-shm` (the default for SGI Challenge). To run a program:

1. Set shell variable `BSP_PROCS` to the number of processes to run. If unset, it defaults to the number of shared-memory processors available if that can be determined; otherwise it defaults to 1.
2. Start the program. Remote processes inherit arguments and environment, and run in the current directory.

ENVIRONMENT

BSP_PROCS

number of parallel processes to run.

MP_PROCS

(MPL version only) number of parallel processes to run.

BSP_PROFILE

pathname of file to collect raw data for superstep run-time profiling (see `bspprof(1)`). If unset or empty, no profiling is done.

BSP_CHDIR

(TCP version only) pathname of working directory for remote processes. If unset or null, defaults to the login directory on the remote machine.

BSP_HOSTFILE

(TCP version only) pathname of file containing names of local and remote hosts.

SEE ALSO

`bsp(3)`, `bscc(1)`, `bspf77(1)`

NAME

bspcc – compile a C program using the Oxford BSP library

SYNOPSIS

bspcc [**-v**] [*-medium*] [*cc options ...*] *file ...*

DESCRIPTION

bspcc invokes the C compiler, adding to the list of options and files a reference to the library containing the **bsp(3)** primitives for the local architecture.

OPTIONS

- v** print the generated command before executing it
- medium* specifies which underlying communication medium to use. Invoking **bspcc** with no arguments produces a list of available versions. The default is to use the most efficient.

SEE ALSO

bsp(1), **bsp(3)**, **bspf77(1)**

NAME

bspf77 – compile a FORTRAN program using the Oxford BSP library

SYNOPSIS

bspf77 [**-v**] [*-medium*] [*f77 options* ...] *file* ...

DESCRIPTION

bspf77 invokes the FORTRAN compiler, adding to the list of options and files a reference to the library containing the **bsp(3)** primitives for the local architecture, and any special compiler options necessary for BSP programs (for example, **-static** for Silicon Graphics compilers).

OPTIONS

-v print the generated command before executing it

-medium specifies which underlying communication medium to use. Invoking **bspf77** with no arguments produces a list of available versions. The default is to use the most efficient.

SEE ALSO

bsp(1), **bsp(3)**, **bspcc(1)**

NAME

bspprof – display superstep runtime profile

SYNOPSIS

bspprof [*file*]

DESCRIPTION

If the environment variable **BSP_PROFILE** contains a non-null pathname when a program using the **bsp(3)** primitives is started, elapsed time information will be gathered for each superstep in the program, and written to the named file when **bsp_finish** is called. The **bspprof** command reads this information and displays it in columns under the following headings:

sstep	superstep label given in the bsp_sstep call. If the label is preceded by '<', this entry refers to the time between the end of the previous (dynamic) superstep and the beginning of this one.
%time	percentage of the program's total elapsed time taken by this superstep. Supersteps accounting for less than 0.1% of elapsed time are not shown.
#calls	the number of times this superstep was executed.
pid	process identifier for this line.
elapsed	the average elapsed real time in milliseconds per execution of this superstep, between the call to bsp_sstep and the call to bsp_sstep_end . This represents the computation part of the superstep, and may include some of the communication cost as well (for example buffering, sending 'pre-fetch' requests, and dealing with incoming requests).
sync	the average elapsed real time in milliseconds during the execution of bsp_sstep_end at the end of this superstep. This represents the rest of the communication cost, and the delay while waiting for barrier synchronisation.

OPTIONS

file file containing the raw profiling information (default **\$BSP_PROFILE**).

LIMITATIONS

Note that, unlike the profilers **prof(1)** and **gprof(1)**, all times shown are elapsed ('wall-clock') time rather than CPU time. This means that profiling information will only be valid if the program has been run on a dedicated machine, with no other processes active; and there may be considerable variation from one run to another because of operating system activity behind the scenes.

The profiling mechanism is implemented using three calls to the **bsp_time(3)** routine per superstep. If a high-precision, low-overhead hardware clock is not available, the overhead of these calls will distort the measurement of very short supersteps.

SEE ALSO

bsp(1), **bsp(3)**

NAME

bsp_start, bsp_finish, bsp_sstep, bsp_sstep_end, bsp_fetch, bsp_store — Oxford BSP library primitives

FORTRAN SYNOPSIS

subroutine bsp_start (maxprocs, nprocs, mypid)

integer maxprocs, nprocs, mypid

subroutine bsp_finish

subroutine bsp_sstep (stepid)

integer stepid

subroutine bsp_sstep_end (stepid)

integer stepid

subroutine bsp_fetch (pid, src, dst, nbytes)

integer pid, nbytes

anytype src, dst

subroutine bsp_store (pid, src, dst, nbytes)

integer pid, nbytes

anytype src, dst

C SYNOPSIS

bspstart (argc, argv, maxprocs, nprocs, mypid)

char **argv;

int *nprocs, *mypid;

bspfinish ()

bpsstep (stepid)

int stepid;

bpsstep_end (stepid)

int stepid;

bspfetch (pid, src, dst, nbytes)

int pid, nbytes;

void *src, *dst;

bspstore (pid, src, dst, nbytes)

int pid, nbytes;

void *src, *dst;

DESCRIPTION

The Oxford BSP library provides a concise set of basic operations for architecture-independent parallel programming according to the **bulk-synchronous parallel** model of computation. There are implementations of the library for distributed-memory multicomputers, shared-memory multiprocessors, and networked workstation clusters, using a variety of parallel software platforms to provide the underlying communication and synchronisation. All versions present the same set of primitives to the application program, with identical syntax and semantics.

Process management

bsp_start (*maxprocs*, *nprocs*, *mypid*)

initiates the parallel execution of multiple copies of the calling program, on separate processors if possible. The call to bsp_start must

be the first executable statement of the program.

bsp_finish ()

must be called at the end of the program. When every process has called **bsp_finish**, parallel resources are released and the entire program exits.

maxprocs the maximum number of parallel processes to run. *maxprocs* = 0 requests all available processors.

nprocs returns the number of parallel processes actually started, including the original caller. The degree of parallelism may be specified by the user when starting the program, or in a prior ‘configuration’ step; see **bsp(1)** for details on each parallel architecture. Programs should be designed to work correctly for any value of *nprocs* between 1 and *maxprocs*.

mypid returns in each process the **process identifier** of that process, which is an integer between 0 and *nprocs*–1. Process 0 is the only process which can read from standard input, and may be the only one which has access to program arguments and environment variables.

argc (C interface only) the number of arguments passed to **main()**

argv (C interface only) the list of arguments passed to **main()**

An Oxford BSP program is in SPMD (single program, multiple data) form: all processes execute the same program text, but each has its own independent copy of all variables. The master-slave programming style can also be accommodated, by performing ‘master’ operations only in process 0; generally process 0 can act as a slave as well, unless it has useful work to do while the slave computation is proceeding.

The memory model is ‘direct’ BSP, requiring the data to be explicitly partitioned and distributed. There is no global shared data, but a process can make explicit requests to read and write the local data of another process.

Bulk synchronisation

bsp_sstep (*stepid*)

marks the beginning of a **superstep** — a phase of the parallel program during which each process can perform a series of computation steps on local data, and request the asynchronous reading or writing of non-local data to be used in a later superstep.

bsp_sstep_end (*stepid*)

marks the end of a superstep. There is an implicit global barrier synchronisation, in which processes are delayed until every process has reached the end of the superstep and all asynchronous data accesses have completed.

stepid an integer label which serves to identify supersteps in the program for profiling, consistency checking and error reporting. The **bsp_sstep** and **bsp_sstep_end** delimiting one superstep must have the same label, which should be different from any other superstep in the program text, but the choice of step numbering is otherwise arbitrary.

Execution on all processors must proceed through all the supersteps in the same order. This implies that if a superstep is written inside a loop or if statement, the loop control variable or conditional expression must have the same value in all processors. Within a superstep, each processor may perform different computations, follow different control paths, and even call different subroutines, but every processor must have reached the end of the current superstep before any one can proceed to the next.

The present version of the library does not allow supersteps to be nested, or to synchronise only a subset of processes.

Remote data access

bsp_fetch (*pid*, *src*, *dst*, *nbytes*)

requests an asynchronous transfer of data from another process. The caller may not examine the data until after the next **bsp_sstep_end**, because it is not certain to arrive before the barrier synchronisation.

bsp_store (*pid*, *src*, *dst*, *nbytes*)

requests an asynchronous transfer of data to another process. The remote process may not examine the data until after the next **bsp_sstep_end**. But the calling process is permitted to modify its local copy as soon as the **bsp_store** returns, so that it is possible to reuse one buffer variable as the source for multiple stores in the same superstep.

pid process identifier of the remote process

src data source location in the remote process (fetch) or the calling process (store)

dst data destination location in the calling process (fetch) or the remote process (store)

nbytes length in bytes of the data to be transferred

In FORTRAN, *src* and *dst* may be variables, array names, or indexed array expressions; and *src* may also be an expression yielding a value of size *nbytes*. In C, *src* and *dst* are pointers to any type.

Note that **bsp_fetch** and **bsp_store** are not message-passing primitives, although on some machines they will be implemented by means of messages. Semantically they are remote assignment operations, which may be performed on some shared-memory machines as a direct transfer of data from address space to another, with no need for extra buffering or per-message synchronisation.

The present version of the library permits non-local data access only to objects which are statically allocated. That is, the source of a fetch or the destination of a store must be a **static** or **extern** variable in C, a local or **common** variable in FORTRAN.

Consistency rules

To avoid the nondeterminism caused by concurrent access to shared data, there are logical restrictions on the use of the remote assignment primitives.

- The destination data object of a **bsp_fetch** or **bsp_store** may not be examined or modified by any process during the same superstep — that is, it may not be the source or destination of any other **bsp_fetch** or **bsp_store** or appear in a local expression or assignment.

- A data object which is modified locally by a process (by an assignment or input statement) may not be fetched by any other process in the same super-step.

FILES

`$BSP/lib/$ARCH/libbsp_$MEDIUM.a`

where

`$BSP` directory where the Oxford BSP library is installed (possibly `/usr/local/bsp`)

`$ARCH` architecture of parallel machine

`$MEDIUM` underlying communications medium

AUTHOR

Richard Miller <miller@comlab.ox.ac.uk>
Oxford University Computing Laboratory

SEE ALSO

`bsp(1)`, `bspf77(1)`, `bspcc(1)`, `bsp_time(3)`, `bsp_broadcast(3)`, `bsp_reduce(3)`

A bridging model for parallel computation, L.G. Valiant, CACM 33(8): 103–111, 1990.

The Oxford BSP Library Users' Guide, R. Miller and J. Reed, Oxford University Computing Laboratory

COPYRIGHT

The Oxford BSP library Version 1.1 is Copyright © 1994 by Richard Miller

NAME

bsp_broadcast – broadcast data to all BSP processes

FORTTRAN SYNOPSIS

subroutine bsp_broadcast (pid, src, dst, nbytes)

integer pid, nbytes

anytype **src, dst**

C SYNOPSIS

bspbroadcast (pid, src, dst, nbytes)

int pid, nbytes;

void *src, *dst;

DESCRIPTION

bsp_broadcast copies *nbytes* bytes of data, src location *src* in the process with identifier *pid*, to location *dst* in all processes (including process *pid*). Because **bsp_broadcast** contains one or more BSP supersteps internally, it need not (in fact, must not) be inside an explicit superstep in the calling program. It must be called by all processes, with the same value of *pid* and *nbytes* , and with *dst* representing the same static location.

SEE ALSO

bsp(3), bsp_reduce(3)

COPYRIGHT

The Oxford BSP library Version 1.1 is Copyright © 1994 by Richard Miller

NAME

`bsp_reduce` – combine data from all BSP processes

FORTRAN SYNOPSIS

```
subroutine bsp_reduce (op, src, dst, nbytes)
external op
integer nbytes
anytype src, dst
subroutine op (result, x, y, nbytes)
integer nbytes
anytype result, x, y
```

C SYNOPSIS

```
bspreduce (op, src, dst, nbytes)
void (*op)();
int nbytes;
void *src, *dst;
...
op (result, x, y, nbytes)
int nbytes;
void *result, *x, *y;
```

DESCRIPTION

`bsp_reduce` uses the binary operation defined by subroutine *op* to combine the local copies of the data at location *src* in all processes into a single result, which is broadcast to location *dst* in all processes. The objects *src* and *dst* must be at the same static location in all processes, and all have a size of *nbytes* bytes. The user-defined subroutine *op* combines a pair of operands *x* and *y* of length *nbytes*, placing the result in *result*. Note that `bsp_reduce` may call the *op* routine with either *x* or *y* referring to the same location as *result*.

The order in which the individual combining operations are performed is implementation-dependent. If the operation performed by *op* is not associative, the result of `bsp_reduce` may be different from the result of a sequential reduction; however, at the end of any given call to `bsp_reduce`, the result held in all processors will be identical. The operation need not be commutative for example, reducing with matrix-multiply produces a deterministic result independent of evaluation order.

Because `bsp_reduce` contains one or more BSP supersteps internally, it need not (in fact, must not) be inside an explicit superstep in the calling program.

SEE ALSO

`bsp(3)`, `bsp_broadcast(3)`

COPYRIGHT

The Oxford BSP library Version 1.1 is Copyright © 1994 by Richard Miller

NAME

bsp_time, **bsp_dtime** – high-precision real-time clock

FORTRAN SYNOPSIS

double precision function bsp_time ()
double precision function bsp_dtime ()

C SYNOPSIS

double bsp_time ()
double bsp_dtime ()

DESCRIPTION

bsp_time returns the elapsed real time in seconds since the start of the program (or since the first call to **bsp_time** if not called by an Oxford BSP program).

bsp_dtime returns the elapsed real time in seconds since the previous call to **bsp_dtime**.

On some machines (such as the Silicon Graphics, and IBM RS/6000 and SP1), these routines use a high-precision hardware clock directly available to the user process, giving sub-microsecond resolution and only a few microseconds overhead. In other environments an operating-system routine such as **gettimer(2)** or **gettimeofday(2)** is used, resulting in higher overhead and coarser resolution.

COPYRIGHT

The Oxford BSP library Version 1.1 is Copyright © 1994 by Richard Miller

Bibliography

- [Amdahl, 1967] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings AFIPS Spring Joint Computer Conference 30*, pages 483–485, 1967.
- [Banger, 1993] C.R. Banger. *Categorical Construction of Arrays*. PhD thesis, Queen’s University, 1993.
- [Bird, 1987] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
- [Bird, 1988] R.S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*, volume 55 of *NATO ASI Series F*, pages 151–218. Springer-Verlag, 1988. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.
- [Bird, 1990] R.S. Bird. A calculus of functions for program derivation. In David Turner, editor, *Research Topics in Functional Programming*, pages 287–308. Addison-Wesley, 1990. University of Texas at Austin Year of Programming Series.
- [Bisseling and McColl, 1993] R.H. Bisseling and W.F. McColl. Scientific computing on bulk synchronous parallel architectures. Technical Report 836, Department of Mathematics, University of Utrecht, December 1993.
- [Cheatham *et al.*, 1994] T. Cheatham, A. Fahmy, and D. Stefanescu. H-BSP — a general purpose parallel computing environment. In *Proceedings of IFIP World Congress, Hamburg, August 1994*, volume 1, pages 515–520, 1994.
- [DeBolt and Kollman, 1993] S.E. DeBolt and P.A. Kollman. AMBERCUBE MD, parallelization of AMBER’s molecular dynamics module for distributed-memory hypercube computers. *Journal of Computational Chemistry*, 14:312–329, 1993.

- [Ellingworth, 1989] H.R.P. Ellingworth. *Parallel Algorithms for the Force-field Method in Molecular Modelling*. PhD thesis, University of Oxford, 1989.
- [Flynn, 1972] M.J. Flynn. Some computer organizations and their effectiveness. In *IEEE Trans. on Comp.*, volume C-21, pages 948–960, 1972.
- [Geist and others, 1993] A. Geist et al. *PVM 3 User's Guide and Reference Manual*, 1993.
- [Gerbessiotis and Valiant, 1992] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, Springer Lecture Notes in Computer Science, July 1992.
- [G.Hains and Mullin, 1991] G.Hains and L.M.R. Mullin. An algebra of multidimensional arrays. Technical Report 782, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, Canada., Aug 1991.
- [Gibbons, 1990] J. Gibbons. *Algebras for Tree Algorithms*. D.Phil. thesis, Programming Research Group, University of Oxford, 1990.
- [Hey, 1991] A.J.G. Hey. The genesis distributed-memory benchmarks. *Parallel Computing*, 17:1275–1283, 1991.
- [High Performance Fortran Forum, 1993] High Performance Fortran Forum. High performance fortran language specification. *Scientific Programming*, 2, June 1993.
- [Hoare, 1985] C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computing science. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1985.
- [Hockney, 1991] R.W. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17:1111–1130, 1991.
- [Iverson, 1962] K.E. Iverson. *A Programming Language*. John Wiley & Sons, 1962.
- [Jeuring, 1992] J. Jeuring. The derivation of a hierarchy of algorithms for pattern matching on arrays. In *Proceedings of the ATABLE-92 Workshop*, 1992.
- [Malcolm, 1988] G. Malcolm. Homomorphisms and promotability. Computing Science Notes CS8815, Department of Computing Science, University of Groningen, 1988.

- [McColl, 1993a] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 337–391. Cambridge University Press, Cambridge, 1993.
- [McColl, 1993b] W.F. McColl. GL: An architecture independent programming language for scalable parallel computing. Technical Report 93-072-3-9025-1, NEC Research Institute, Princeton, 1993.
- [Meertens, 1986] L. Meertens. Algorithmics: Towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986. CWI Monograph 1.
- [Miller and Reed, 1993] R. Miller and J.L. Reed. *The Oxford BSP Library Users' Guide, Version 1.0*, 1993.
- [Miller, 1990] R. Miller. Parallel implementation of molecular force field calculations. Project report, Programming Research Group, University of Oxford, 1990. SERC-sponsored project: Parallel Algorithms and Architectures.
- [Miller, 1993] R. Miller. A library for bulk synchronous parallel programming. In *Proceedings of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing, 22 Dec 1993*, pages 100–108, 1993.
- [Pearlman *et al.*, 1990] David A Pearlman, David A Case, James C Caldwell, George L Seibel, U Chandra Singh, Paul Weiner, and Peter A Kollman. *AMBER 4.0*, 1990.
- [Ryckaert *et al.*, 1977] J. Ryckaert, G. Ciccotti, and H.J.C. Berendsen. Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of *n*-alkanes. *Journal of Computational Physics*, 23:327–341, 1977.
- [Skillicorn, 1990] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.
- [Skillicorn, 1993] D.B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 20(3):205–221, June 1993.
- [Valiant, 1982] L.G. Valiant. A scheme for fast parallel computation. *SIAM J. Comput.*, 11:350–361, 1982.
- [Valiant, 1989] L.G. Valiant. Bulk synchronous parallel computers. Technical Report TR-08-89, Computer Science, Harvard University, 1989.

- [Valiant, 1990a] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
- [Valiant, 1990b] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*. Elsevier Science Publishers and MIT Press, 1990.
- [Wadler and Miller, 1990] P.L. Wadler and Q. Miller. *An introduction to Orwell 6.00*, 1990.
- [Weiner *et al.*, 1984] S J Weiner, P A Kollman, D A Case, U C Singh, C Ghio, G Alagona, S Profeta Jr, and P Weiner. A new force field for molecular mechanical simulation of nucleic acids and proteins. *JACS*, 106:765–784, 1984.
- [Weiner *et al.*, 1986] S J Weiner, P A Kollman, D T Nguyen, and D A Case. An all atom force field for simulations of proteins and nucleic acids. *Journal of Computational Chemistry*, 7:230–252, 1986.
- [Woods *et al.*, 1994] R.J. Woods, C.J. Edge, M.R. Wormald, and R.A. Dwek. GLYCAM.93: a generalised parameter set for molecular dynamics simulations of glycoproteins and oligosaccharides. application to the structure and dynamics of a disaccharide related to oligomannose. In Klaus Bock and Henrik Clausen, editors, *Complex Carbohydrates in Drug Research, Alfred Benzon Symposium*, volume 36, pages 15–26. Munksgaard Copenhagen, 1994.
- [Wright, 1988] C.J. Wright. A theory of arrays for program derivation, 1988.