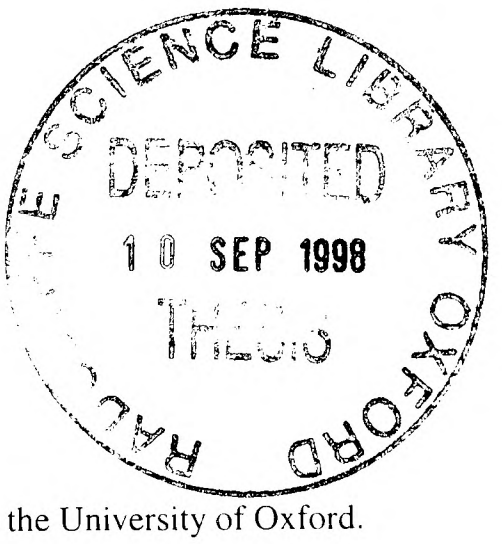
Opal: Modular Programming using the BSP Model

Simon Knee
Keble College

Programming Research Group
Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD
United Kingdom



MICHAELMAS TERM 1997

Submitted in support of an application for the degree of D. Phil in the University of Oxford.

Abstract

Parallel processing can provide the huge computational resources that are required to solve todays grand challenges, at a fraction of the cost of developing sequential machines of equal power. However, even with such attractive benefits the parallel software industry is still very small compared to its sequential counterpart. This has been attributed to the lack of an accepted parallel model of computation, therefore leading to software which is architecture dependent with unpredictable performance. The Bulk Synchronous Parallel (BSP) model provides a solution to these problems and can be compared to the Von Neumann model of sequential computation.

In this thesis we investigate the issues involved in providing a modular programming environment based on the BSP model. Using our results we present Opal, a BSP programming language that has been designed for parallel programming-in-the-large. While other BSP languages and libraries have been developed, none of them provide support for libraries of parallel algorithms.

A library mechanism must be introduced into BSP without destroying the existing cost model. We examine such issues and show that the active library mechanism of Opal leads to algorithms which still have predictable performance.

If algorithms are to retain acceptable levels of performance across a range of machines then they must be able to adapt to the architecture that they are executing on. Such adaptive algorithms require support from the programming language, an issue that has been addressed in Opal.

To demonstrate the Opal language and its modular features we present a number of example algorithms. Using an Opal compiler that has been developed we show that we can accurately predict the performance of these algorithms.

The thesis concludes that by using Opal it is possible to program the BSP model in a modular fashion that follows good software engineering principles. This enables large scale parallel software to be developed that is architecture independent, has predictable performance and is adaptive to the target architecture.

Acknowledgements

I would like to thank my supervisor, Bill McColl, for his encouragement and advice during my time at the Computer Laboratory. I am also grateful to Friedrich Kruse-Fautsch, Constantinos Siniolakis and Ronald Sujithan for their informative discussions on BSP.

I would also like to thank the members of Keble College for their support, help and conference meals. Steve Kersley, Ken Lovett, Anna Millard, Michael Murphy, Deborah Rogers, Diamond Versi and Penny White deserve a special mention. For providing a friendly and apaptable environment while finishing this thesis I would like to thank the staff of XLNT. Particular thanks are due to Mike Anello and Glenn Kiyono.

I am grateful for the help and motivation of Paul Barham, Jason Brown, Philip Gregory, Matthew Jennings and all my family. Without them the completion of this thesis would have been an extremely difficult task.

Finally I would like to thank my wife, Deborah Knee, for her continued support, encouragement and endurance throughout the duration of my D.Phil, and in particular during its final stages.

This work was supported by a grant from the EPSRC (formerly SERC) and a CASE studentship funded by Marconi Radar Systems Ltd.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel processing has been an active topic of research since the initial days of computing. Even though considerable effort has been put into this area there is still no consensus on parallel architectures, languages or models. Although parallel architectures are beginning to converge this has been largely due to financial and technological factors. It is still the case that the majority of computing machines presently in use are sequential.

There are a number of reasons why parallel processing is not the normal mode of computation, including:

- Parallel architectures vary dramatically from machine to machine. An efficient way to perform a computation on one machine is often impractical on another.
- Parallel programs can be more difficult to design and debug than sequential ones. This is compounded by the fact that almost every programmer is first taught to program in a sequential language.
- Methods have been developed to analyse sequential programs in terms of performance. A problem in many industrial applications is predicting how long an algorithm will take to run, or how much hardware will be required to execute it in a given time. The performance prediction of parallel algorithms is still a difficult task.
- Sequential programs can be made portable with negligible loss in performance. This is not the case for parallel algorithms.

Why are these problems less relevant in the sequential case? Sequential machines do not vary dramatically from one architecture to another since they all conform to a model - the Von Neumann model. Each sequential machine may have its own features in order to improve performance but it still complies to this model. Many models exist for parallel architectures but no single one has yet proved to be superior. Once a model is decided upon[1] the problem of dramatically different architectures will be resolved.

The programming of parallel algorithms can only be made easier with improved parallel languages and tools. Of course, if the model of computation being used is difficult for the programmer to comprehend then their problems are only increased.

The ability to analyse the performance of algorithms is a property of the model of computation being used. The Von-Neumann model allows you to predict performance by stating that instructions and data are taken from memory and executed in sequence by the processor. Hence execution time is simply the total number of instructions to be executed multiplied by the time per instruction. Although this is a rather simplified view due to effects such as cache performance, it is often a useful figure for estimating performance.

The portability of programs is related to the number of properties that are assumed about the target machine. An example in the sequential case is an algorithm that assumes that a portion of code is executed in a specified time e.g. the delay loop of a real time algorithm. If such an algorithm is ported to a processor whose speed is different then the delay loop will execute slower or faster, perhaps resulting in data being lost or not serviced quickly enough. The speed of the processor is a property of the sequential model. This highlights the fact that any model of computation cannot assume too many properties of the machine, and yet it must capture the essential details in order to be useful.

A problem that is apparent with parallel processing, more than sequential, is that of the portability versus performance trade off. This means that the majority of sequential algorithms will still run efficiently when ported to another sequential architecture. In contrast, many parallel algorithms rely on specific details regarding the target architecture, e.g. its communications network topology. If the parallel model has to be general enough to cover all architectures then it must capture the performance range of these machines. For this reason the parallel model should be parameterised to allow the algorithm to adapt to the target architecture.

These facts suggest that a fixed model of parallel computation should be defined in an analogous way to the Von-Neumann model of sequential computation. Once defined, programming languages can be designed to fit this model and to allow cost analysis to be performed. If algorithms

[1] Or industry dictates one by the convergence of parallel architectures.

are written in a way that uses the parameters of the model then they should be able to adapt to different architectures with minimum performance loss. If the model abstracts away from the actual architecture then the algorithms should also be portable. Portable algorithms may not achieve the same level of performance as algorithms developed for specific architectures, but this may easily be counteracted by savings in the development and maintenance of such software.

1.1 Models of Parallel Computation

Many models of parallel computation have been devised. This section does not attempt to describe each model of computation completely, but rather gives an overview of the essential features. For a recent survey of parallel computation models see [Maggs *et al.*, 1995].

1.1.1 PRAMs

A parallel random access machine (PRAM) can be described as a set of processors which access a global memory [Fortune and Wyllie, 1978]. Accessing any location in this global memory is assumed to take the same amount of time, this being the time to execute one instruction.

Since all processors can access memory in a single instruction it is often the case that two or more processors can try and access the same location. This is resolved by splitting the PRAM model into a number of different classes, with each class having a separate method of resolving memory conflicts. Examples of such classes are the Exclusive-Read Exclusive-Write (EREW) PRAM where no more than one processor is allowed to access each memory location, or the Concurrent-Read Exclusive-Write (CREW) PRAM where processors can perform a shared read but not a shared write. A good introduction to the PRAM model can be found in [Gibbons and Spirakis, 1993].

The PRAM model is excellent for designing parallel algorithms and performing cost analysis, and much work has been done in this area [Jájá, 1992]. The success of this model is largely due to the inherent portability of algorithms that are written for it, and the fact that the performance of these algorithms can be easily compared.

The disadvantage of the PRAM model is that these machines are very difficult to construct in reality. Designing a machine that has a global shared memory with access to any element within one instruction cycle is an extremely hard problem. A possible implementation of the PRAM model has been proposed with the Fluent Abstract Machine [Ranade, 1989]. This uses combin-

ing networks on a butterfly topology with a hashed address space to try and hide the network latency. Ranade's approach has been analysed in a quantitative way by giving cost models for implementing various parts of the PRAM machine [Abolhassan *et al.*, 1991], more recently called the SB-PRAM [Abolhassan *et al.*, 1993, Bach *et al.*, 1997]. This is then used to demonstrate an improvement on Ranade's Fluent machine using multiple butterflies and parallel slackness. It is shown that the proposed improved Fluent machine would have a similar price / performance ratio to conventional distributed memory architectures.

Other attempts at realising the PRAM model involve its simulation on conventional distributed memory architectures. This method usually involves hashing the address space of the PRAM across the distributed memory of the machine with replication of variables [Mehlhorn and Vishkin, 1984], or using multiple hash functions [Abolhassan *et al.*, 1991].

Some variants of the PRAM model have attempted to address the issue of locality, relaxing the constraint that access to *any* location has a cost of one instruction. Such models include the HPRAM [Heywood and Ranka, 1992a, Heywood and Ranka, 1992b], YPRAM [de la Torre and Kruskal, 1991], LPRAM [Aggarwal *et al.*, 1990] and BPRAM [Aggarwal *et al.*, 1989]. Comparisons and critical analyses of these various models can be found in [Heywood and Leopold, 1995].

1.1.2 BSP

A Bulk Synchronous Parallel machine consists of a number of processor-memory pairs connected by a communications network [Valiant, 1990, Valiant, 1989, Skillicorn *et al.*, 1996]. This network is assumed to be able to deliver messages from point to point with uniform cost - this means the cost of accessing a remote processor's memory is independent of the locality of the processor.

Execution of a program in the BSP model progresses as a series of supersteps. Inside a superstep a process may update its local and global variables, but other processes will not see the updates to global memory until the end of the current superstep. At this point all processes synchronise, global memory is updated, and execution continues to the next superstep.

The model is parameterised by four variables:

- p. Number of processors.
- s. Speed of each processor.
- l. Bulk synchronisation period of the network.

- g. Global communications performance.

The units used for s, l and g are arbitrary as long as they are consistent. Possible units include processor cycles, instruction cycles, real time or floating point operations. Typically s is measured in floating point operations with l and g normalised with respect to this, i.e. l is measured in floating point operations and g in floating point operations per word.

The parameters p and s are obvious and require no further explanation. l and g are used to measure the communications performance. l is the number of steps required for a bulk synchronisation of processors. g is defined in terms of an h-relation.

Definition 1.0: *An h-relation is a routing problem where each processor sends and receives at most h words.*

g can now be defined as:

Definition 1.1: *g is the value such that an h-relation is implemented in $g * h$ steps.*

These parameters can be combined in order to calculate the cost of a superstep. The key question to be considered before a superstep cost can be calculated is whether communication and computation can be overlapped. If a superstep performs an amount of computation consisting of c steps and realises an h-relation then assuming no overlap the cost, T, of the superstep is:

$$T = l + c + gh \tag{1.1}$$

If communication and computation can be perfectly overlapped then the equation 1.1 becomes:

$$T = l + \max(c, gh) \tag{1.2}$$

The BSP model is designed to relax the constraints of the PRAM model and thus make it a more realistic option to implement.

1.1.3 LogP

The LogP model [Culler *et al.*, 1993b] attempts to improve upon the BSP model. As with BSP, the LogP model views a parallel machine as a number of processor-memory pairs attached to a communications network. It also assumes that access to global memory incurs a uniform cost.

Execution of an algorithm on the LogP model proceeds as a number of separately executing

threads. The threads are asynchronous with respect to each other, in contrast to the synchronous nature of processes in the execution of a BSP algorithm.

The model is parameterised by four variables. It should be noted that although some of the parameters in the LogP model have the same names as those in the BSP model they are quite different.

- L. This is an upper bound on the latency incurred in communicating a message.
- o. The overhead of performing a communication, i.e. the number of cycles that **have** to be spent by the processor to initiate a communication.
- g. The *gap*, which is the minimum time interval between performing communications. $\frac{1}{g}$ is the bandwidth per processor.
- P. The number of processors.

To illustrate the use of these parameters we consider the various equations for costing statements in the LogP model. To send a message to another processor will cost an overhead of o to initiate the communication, a delay of L while it is transported, and an overhead of o to receive the message. This is reflected in equation 1.3, which gives the cost for sending a signal to another processor. Equation 1.4 is the cost of reading or writing a remote location, in which case not only do we need to signal the target processor but we also need to wait for an acknowledgement.

$$cost\_signal = 2o + L \tag{1.3}$$

$$cost\_rw = 4o + 2L \tag{1.4}$$

The model assumes that there is a delay of exactly g in issuing a message to the network. With high network traffic this may not be true so the model limits the number of packets that can be injected into the network to one in every $\frac{L}{g}$ steps. If an algorithm obeys this rule then it is said to be well behaved. If it is not well behaved then it will be subject to stalling in order to preserve the $\frac{L}{g}$ limit.

1.1.4 Other Latency-Bandwidth Models

BSP and LogP are both examples of latency-bandwidth based models. In such models communication is assumed to consist of an initial overhead or latency, after which the bandwidth dictates

the speed of the transfer. These are modelled independently in LogP using the o and L parameters, while in BSP they are both included in the g parameter.

Variations of both the BSP and LogP models have been developed that allow for large messages. The BSP* model [Bäumker *et al.*, 1996, Bäumker *et al.*, 1995] changes the cost of an h-relation of size s to be $g*h*\lceil\frac{s}{B}\rceil$, where B is the minimum size the packets must have in order to fully utilise the communications network. This model therefore charges a fixed cost for messages smaller than B, making it attractive to send large messages where possible. The LogGP model [Alexandrov *et al.*, 1995] also allows for large messages using LogP.

The Coarse Grained Multicomputer (CGM) [Dehne *et al.*, 1993] considers a parallel machine as a number of processor-memory pairs connected by some network. Using this scheme all communication is of the form of a global sort of $O(n)$ items on $O(p)$ processors. In [Dehne *et al.*, 1993] it is demonstrated how global sorts can support operations such as broadcasts and prefix sums.

The WPRAM [Nash *et al.*, 1995] is a model that is similar to BSP but allows much greater asynchrony. It also uses combining networks and has a slightly more detailed cost model than that of BSP.

1.1.5 Comparison

The PRAM model, although excellent for analysing algorithms, is unrealistic for real parallel machines. Using the techniques of [Abolhassan *et al.*, 1991, Ranade, 1989] could lead to an implementation of a PRAM machine but it is still unclear whether or not this can be done efficiently.

The LogP model seems excellent for predicting the performance of algorithms and even captures the concept of prefetching. This cost of prefetching, or computation and communication overlap, cannot be estimated directly in the BSP model. This extra expressiveness of the LogP model is at the cost of simplicity. Analysing a LogP algorithm is not just a case of looking at the superstep structure, since all processes are running asynchronously, but instead the whole algorithm must be considered. It must be checked that the algorithm does not issue requests to the communications network more frequently than $\frac{L}{g}$, or delays will be incurred that cannot be predicted. It could even be the case that a well behaved algorithm on one architecture becomes a stalling algorithm on another.

In [Culler *et al.*, 1993b] it is noticed that the performance prediction of an FFT algorithm becomes unstable for larger problems on a LogP machine. The solution suggested is to insert a barrier synchronisation into the algorithm, i.e. change the algorithm into a superstep structure. In Split-C,

the programming language for the LogP model [Culler *et al.*, 1993c], statements such as `sync()` and `all_sync_now()` are given to signal that updates to global memory can be performed. This mechanism is similar to the memory updates in the BSP model.

A comparison of the LogP and BSP models can be found in [Bilardi *et al.*, 1996]. This indicates that while the LogP model may be a more accurate representation of a real parallel machine this is at the expense of simplicity and programming abstraction. Comparisons of other parallel models of computation can be found in [Maggs *et al.*, 1995, Heywood and Leopold, 1995].

1.2 Parallel Programming Languages

In the following sections we shall describe some of the parallel programming languages that are related to the above models of computation. A good overview of recent parallel programming languages and tools can be found in [Perrott, 1992, Cheng, 1993].

1.2.1 FORK

FORK [Hagerup *et al.*, 1991] is a programming language developed for the PRAM model. It is a Pascal like language with constructs added for starting new threads of control. FORK offers a global address space to the programmer, but also allows variables to be declared local to a process. As with the PRAM model, updates to shared memory take effect immediately.

All statements in FORK execute synchronously with respect to each other due to the properties of the PRAM model. To allow threads of control to run asynchronously, FORK has the concept of a group. A group is a set of threads that share variables and run synchronously with respect to each other. Threads that are in different groups execute asynchronously. Groups can be hierarchical so that inter-group communication can be performed using shared variables. This hierarchical structure is similar to the group structure of GPL [McColl and Miller, 1995, McColl, 1994a].

New processes are started in FORK using the **start [1..N]** statement. This will create N processes who are all members of the group that the **start** statement was executed in. These processes can access the global memory of the parent process but cannot access its private local memory. At the end of the **start** statement all processes synchronise and terminate.

New groups are created using the **fork [1..N]** statement. This again creates N separately executing processes, but they are now members of a new group which is a sub group of the parent.

An interesting feature of FORK is the **if** statement. If the condition of an **if** statement is based

upon a local variable then different processes in the same group could take different branches. This could lead to the different processes becoming unsynchronised since there may be more statements in the true branch than that of the false or vice-versa. To rectify this the **if** statement is defined such that if the condition depends upon the value of a local variable then the two branches are executed as different groups. Since groups run asynchronously, and only synchronise at the end, this solves the problem.

Recent work on FORK has produced a new programming language called Fork95 [Kessler and Seidl, 1995]. It contains additional features above the original FORK language, making it more practical for real use. The syntax has also been changed so that it is a superset of C.

1.2.2 Jade

Jade [Lam and Rinard, 1991, Scales *et al.*, 1991] uses a substantially different approach than the other languages in this section. Jade attempts to add compiler directions to sequential code to allow it to be executed in parallel. It makes the assumption that the sequential program can be split up into a number of coarse grain tasks by the compiler. They take the view that explicit parallelism is too complicated to program with, and that a programmer annotating a sequential program with information about how variables are used will be sufficient. This is effectively an automatic translation of algorithms written for the Von-Neumann model into a parallel computation model.

Given a sequential program written in C the programmer annotates it with commands such as **withth**, **withonly** and **without**. These commands state that the segments of code they enclose will need exclusive read (**rd**), exclusive write (**wr**), or both exclusive read and write (**rw**) to the specified shared variables. **withth** claims variables, **without** releases them and **withonly** warns the system that variables are about to be claimed. For example, consider the following Jade code:

```
double x, y;
x = f(1);
{   double s;
    s = g (2); x = h (x, s);
}
y = f(2);
```

A programmer may annotate this as follows:

```
SharedDouble x, y;
withth { x.wr(); } () {
    x = f(1);
}
withth { x.rw(); } () {
    double s;
    s = g (2); x = h (x, s);
}
withth { y.wr(); } () {
    y = f (2);
}
```

The compiler and runtime system can then work out from the side effect specifications of the **withth** statements that the assignment $x = f(1)$ and $y = f(2)$ can occur in parallel[2], whereas the middle block of code must execute after the first block.

1.2.3 Split-C

Split-C [Culler *et al.*, 1993c, Culler *et al.*, 1993a] is a programming language developed for the LogP model [Culler *et al.*, 1993b]. The syntax is that of the C programming language, with extra keywords added to allow for shared variables and synchronisation.

The language offers a global address space to the programmer, but can compile to a distributed memory architecture. Each processor claims a certain amount of this global address space as its own and controls the use of it. The method of updating remote memory depends on the type of assignment that is performed, which in turn affects the cost of the assignment statement.

Split-phase Assignment

The split-phase assignment operator (:=) allows computation and communication to be overlapped during the reading or writing of a remote variable. For example, if G is a global pointer and x a local pointer then the assignment $*G := *x$ will complete as soon the write request has been issued[3]. This costs $cost\_rw$ steps as defined in equation 1.4.

The data transfer involved in a split-phase assignment is only guaranteed to have occurred after

[2]The function $f()$ would also need a side effect specification .

[3]The processor is stalled for o units of overhead while issuing the request.

the processor has executed a `sync()` statement. The intention of the operator is that useful work could be performed by the processor before it has to execute a synchronisation. An important point to note is that only the processor performing the split-phase assignment has to issue the `sync()` statement.

Signalling Store

The signalling store operator (`:-`) is a relaxed form of the split-phase assignment, where the requesting processor does not need to wait for an acknowledgement message. For example, the assignment $*G$:- $*x$ completes as soon as a message has been sent to update the remote store.

Signalling store uses the `store_sync(n)` and `all_store_sync()` functions to implement synchronisation between processors. If a processor executes `store_sync(n)` then it will wait until it has been signalled that n bytes of data have been written to it. The `all_store_sync()` function would be used by all processors as a method of synchronising all outstanding signalling store operations.

Since signalling store operations only send messages in one direction their cost is half of that of a split-phase assignment, as shown in equation 1.3.

Bulk Transfer

Split-C provides library functions that can transfer whole arrays in one message. This not only avoids multiple overhead costs in sending small messages, but also gives a greater opportunity for overlapping communication and computation. The cost of this type of statement depends upon the amount of data being transferred.

1.2.4 High Performance FORTRAN

High performance FORTRAN (HPF) [Forum, 1993] uses the data parallel model of computation. It is an attempt to extend existing versions of FORTRAN to allow the computational power of parallel machines to be exploited. Due to the large number of FORTRAN applications in the scientific and engineering communities this seems an attractive option.

The statement **FORALL** is used to describe that a loop can be executed in parallel. The semantics are defined such that conceptually the right hand side of the loop is evaluated for all index values and then this is assigned to the left hand side. For example, consider the following piece of code:

FORALL (i = 2:999) a(i) = a(i-1) + a(i) + a(i+1)

This would be executed correctly in parallel since the right hand side is evaluated first for all values of *i*, after which it is assigned to the left hand side.

Arrays can be distributed across processors using the **DISTRIBUTE** keyword. Such distributions can be performed in a block or cyclic fashion. An interesting feature of HPF is the ability to align two distributed arrays in order to guarantee that certain elements are on the same processor. This is done using the keyword **ALIGN**, an example of which is show below.

ALIGN a(i) **WITH** b(i+1)
FORALL (i = 2:999) a(i) = a(i) + b(i+1)

The **ALIGN** statement guarantees that *a(i)* and *b(i+1)* are on the same processor therefore allowing *a(i) + b(i+1)* to be efficiently evaluated. The **ALIGN** statement can only be performed once on an array at the beginning of execution. Arrays can be declared as **DYNAMIC** in which case **REALIGN** statements can be performed on them at any point in the execution. While such alignment operators are attractive they are often difficult to implement efficiently in real compilers, and perhaps hide the real costs from the programmer.

1.2.5 OCCAM-3

OCCAM-3 [Barrett, 1992] is a message passing language based on the CSP model of computation [Hoare, 1985]. OCCAM-3 adds a number of extensions to the OCCAM-2 language [Inmos, 1988] including user defined data types, records and unions, shared channels, libraries and modules.

The most interesting of these is the shared channel. Communication in OCCAM-2 is performed in a strictly pairwise fashion. In order to implement a library mechanism, channels needed to be shared to allow different processes to use the same library functions, requiring a many to one message passing system. This was done using the shared channel, an example of which follows.

```
CHAN TYPE REC -- Declare a record of two channels
    RECORD
        CHAN OF REAL32 param, result:
:
PAR
    CLAIM cos -- Claim the channel cos for use
        cos[param] ! x
        cos[result] ? y

    GRANT cos -- Grant a process use of this channel
        cos[param] ? c
        cos[result] ! COS (c)
```

In this example the channel *cos* is shared using the **CLAIM** and **GRANT** keywords.

The developers of OCCAM-3 realised the importance of the library mechanism for code re-use. For this reason libraries were implemented in OCCAM-3 in a fashion similar to most other modular languages - an interface with a body that performs the described functions.

1.2.6 BSP FORTRAN and BSP C

Extensions have been added to the FORTRAN and C programming languages to allow them to be programmed using the BSP model [Miller, 1993]. These extensions are in the form of a library of procedure calls to perform tasks such as barrier synchronisation and the reading and writing of remote memory locations.

The superstep execution of a BSP algorithm is described using the procedures `bsp_sstep()` and `bsp_sstep_end()`. At the end of the `bsp_sstep_end()` statement any updates to a processor's memory are made visible to the other processors.

Remote fetch and store operations are performed using the `bsp_fetch()` and `bsp_store()` functions. For example, if a process wishes to fetch a variable *flag* from a process *master* into a local variable then it would perform:

```
bsp_fetch (master, &flag, &local);
```

Similar extensions have also been added to the C++ programming language [Lecomber, 1995], which attempts to capture the BSP model in the form of C++ classes.

1.2.7 BSPlib

BSPlib [Hill *et al.*, 1997c] is the successor to the BSP FORTRAN and BSP C libraries [Miller, 1993]. The aim was to provide the algorithm designer with a small set of BSP functions that can be efficiently implemented on existing parallel architectures. The library focuses on the issues of scalability, portability and predictability. Included with the library is a tool for the automatic prediction of BSP algorithms on different target architectures using the profiling data taken from a single architecture [Hill *et al.*, 1996], and a call graph profiling tool [Hill *et al.*, 1997b].

The library provides support for both direct remote memory access (DRMA) and bulk synchronous message passing (BSMP). Both these methods of communication have variants depending on the level of buffering required for the sending and receiving processes.

DRMA access relies on processes registering sections of memory that are to be shared. This is done using the `bsp_push_reg()` function. Once this area of memory has been registered it can be used as shared variable space using `bsp_put()` and `bsp_get()`.

BSMP allows for a more dynamic communications structure. Using this method processes may send messages to other processes using the `bsp_send()` routine. This function is non-blocking and only guarantees that the data will be present in the target process's buffer at the beginning of the next superstep. The target processor may then move this data from the message buffer using `bsp_move()`.

1.2.8 The Green BSP Library

The Green BSP (GBSP) library [Goudreau *et al.*, 1995] is another set of functions that extends the C programming language. As with BSPlib a minimalist approach was taken - there are only a total of seven functions in the library.

The interesting feature of this BSP library is that it relies solely on message passing, with no shared memory available. The function `bspSendPkt()` is used to send a message to another process, but this message can only be received in the next superstep by calling `bspGetPkt()`. These messages are of a constant size, specifically picked to be both optimal for the target architecture and large enough to be useful. In current implementations this fixed message size is 16 bytes.

The library is purposely designed to be very close to the BSP model, offering only functions that are absolutely necessary. There is no mechanism for accessing the BSP parameters of the machine, making it difficult for an algorithm to automatically adapt to a specific architecture.

1.2.9 GPL

GPL [McColl and Miller, 1995, McColl, 1994a] is a high level programming language developed specifically for the BSP model. GPL allows variables to be declared local to a process while remaining accessible to other processes. Variables local to a process are updated immediately, while global variables are not updated until data synchronisation points using **sync**.

Threads are introduced using the parallel operator **par**. A **par** construct starts off a number of independently executing threads that communicate via shared memory using variables that are higher in the scope. Execution of these threads follows the BSP superstep structure. This means that although separate threads may update the value of a global variable, this update is not visible to other threads until a global synchronisation is performed using **sync**.

Nested **par** statements allow a hierarchy of threads to be created. This hierarchy can be visualised as a tree of depth n. The **sync**(n) statement is used to synchronise all processes a depth n back from the current position in the tree. The command **gsync** can be used to synchronise all processes in the tree and is effectively **sync**(∞).

The following GPL code illustrates how processes are started and the use of the memory update model.

```
var a, x: int;
par
    a := 1; sync; a := 2;
!!
    begin local x;
        sync; x := a;
    end
rap
```

This code will result in the final value of x being 1. This is because the value of variable a is not updated until the **sync** statement after which x is set to the value of a. In this example x was initially global to both threads of control, but then claimed local by the second thread. This

causes all updates to x to take effect immediately in the second thread, while they follow the BSP memory model for the first thread.

GPL also contains a number of novel built in high level functions that operate upon sequences and tuples. Examples of this are the **fold** and **zip** functions performing folding and zip merging of sequences respectively.

1.2.10 Skeletons

Although not truly a programming language, skeletons are a useful methodology for the programming of parallel machines. Skeletons focus on the idea that a parallel algorithm can be constructed from the parallel coordination of sequential components [Cole, 1989, Darlington *et al.*, 1995], where the skeletons are the *glue* which bring these components together.

If a development environment had a number of predefined skeletons available then they can be used to compose sequential algorithms into parallel solutions. Such composition allows for the re-use of both the sequential code, and the skeletons themselves. If skeletons are tuned to the target architecture then this approach to parallel programming can offer portable solutions without significant loss in efficiency. However, it does not provide a cost analysis framework, making it difficult to compare algorithms that are written using skeletons.

1.3 Thesis

From the preceding discussion it can be seen that sequential computation has succeeded due to the portable nature of its algorithms. This success has come from the uniform acceptance of the Von Neumann model of computation, a feature that allows efficient and portable algorithms to be developed.

It is also true that large scale software projects rely on the decomposition of problems into a number of libraries, with each library having a well defined interface. Since parallel software projects are likely to be solving "grand challenges" then parallel languages must include the same software engineering mechanisms as sequential languages. Except for OCCAM-3 this issue has been largely ignored, and is perhaps not well understood for the parallel case.

Portable algorithms are of no use unless they retain some element of efficiency across multiple architectures. This is relatively easy in the sequential case, but very hard for parallel code due to the large number of widely differing architectures. This automatic adaption of algorithms to

the architecture they are running on requires the parallel model being used to be parameterised in some way. The lack of access to the BSP parameters is perhaps a major drawback in the Green BSP library.

A large area for the application of parallel processing is in that of real time systems. In such systems there can be a large amount of computation to be performed within tight time constraints. At the same time these systems must react to many, parallel, inputs from the real world. This requires us to be able to put accurate constraints on the run time of an algorithm, or to be able to determine the level of processor and communication performance required in order to execute in the alloted time. Computation models such as BSP, LogP and their variants allow such analysis.

While adapting existing languages with parallel libraries can lead to useful tools it does not enforce good software engineering practice. For example, library functions can be either intentionally or unintentionally misused, perhaps leading to bizarre problems. Such a problem is the communication of badly typed messages. This type of error is very difficult for a library to detect since it has no access to the workings of the underlying compiler. Developing a new language with a new compiler not only allows these types of problems to be detected, but also gives scope to the compiler to perform optimisations directly related to the parallel model of execution.

This thesis presents a programming language that has been designed specifically for the BSP model. Issues of portability, performance prediction and software engineering principles have been considered from the start. The goal can be simply described as a parallel language that allows efficient portability between architectures, with highly predictable performance, and with constructs that allow libraries of parallel code to be developed by teams of programmers.

The BSP model was chosen since this offered the most accurate cost modelling when compared to that of the PRAM and LogP. It is also a model that incorporates all existing parallel architectures, unlike that of the PRAM model. The LogP model could be said to offer more realistic cost analysis, but this is at the expense of simplicity. It is also the case that some LogP algorithms that are well behaved on one architecture become stalling when ported to a different parallel machine.

In the following chapters we shall discuss the BSP model further and then introduce the rationale behind the design of the Opal programming language. The Opal language definition is described, and an insight given into cost analysis of Opal programs. We shall support these features by giving concrete examples of algorithms that have been implemented and whose performance has been closely analysed. These example algorithms also serve as an illustration of the parallel library mechanism that has been developed. A brief description of the Opal compiler and its use is presented. Finally we shall discuss features that could be added to the Opal language definition in order to increase its usefulness.

Chapter 2

The Bulk Synchronous Parallel Model

2.1 Introduction

In this chapter we will describe the Bulk Synchronous Parallel (BSP) model and its method of execution. The BSP parameters will be elaborated upon, including a more in-depth discussion of h-relations and the global communications performance g. Example algorithms will be given, and simple cost analysis performed. We will then show how BSP algorithms can adapt to the actual parallel architecture that they are being executed on, without requiring changes to the source code of the algorithm.

2.2 The Model

The bulk synchronous parallel model is defined as consisting of a number of processor-memory pairs connected by a communications network [Valiant, 1990, Valiant, 1989]. This network is assumed to be able to deliver messages from point to point with uniform cost. This means that regardless of the locality of a processor the cost of sending a message to it is the same as sending a message to any of the other processors.

Figure 2.1 shows how the memory of each processor can be split into two parts; local and global. Global memory is accessible to all other processors, although it does not have to be real shared memory and may be implemented by some message passing scheme on a distributed memory architecture.

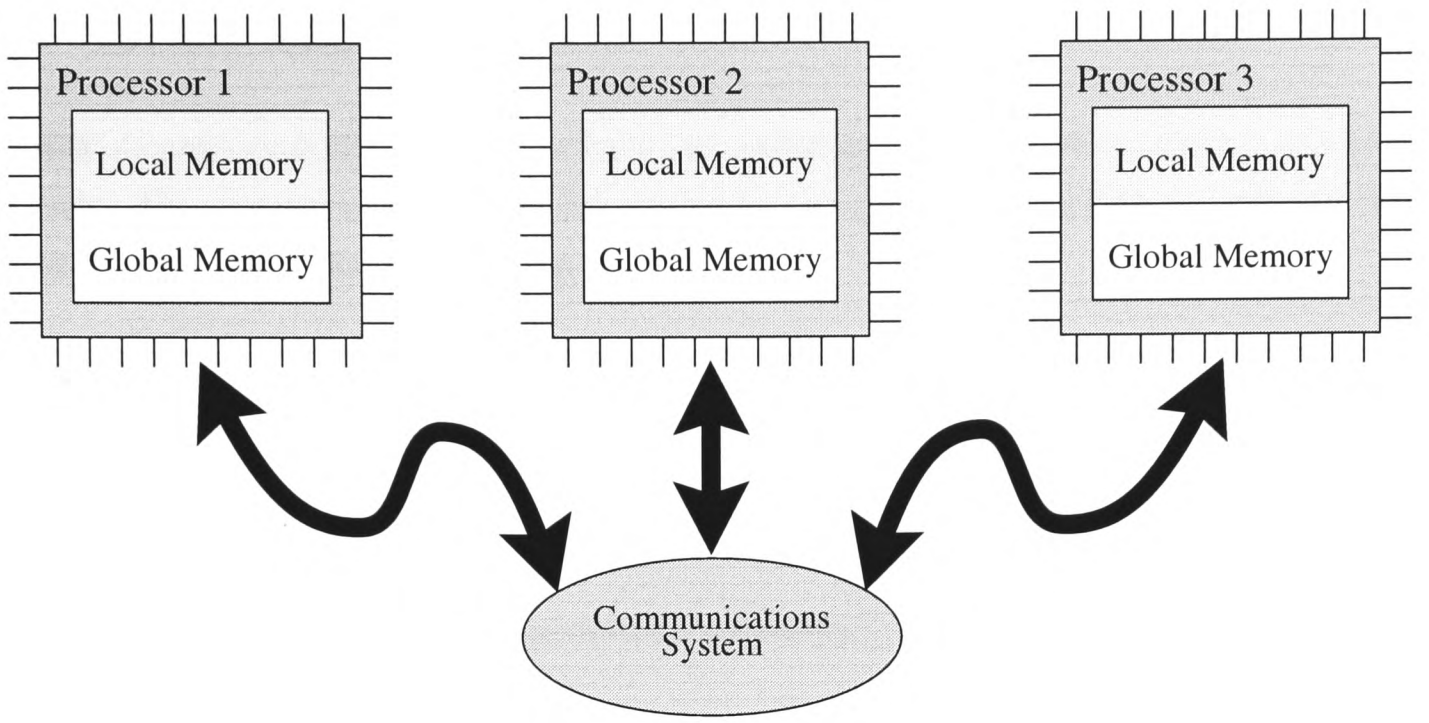


Figure 2.1: The BSP Model

2.2.1 Supersteps

A BSP algorithm consists of a number of processes that execute supersteps. A superstep is simply a collection of statements that perform both local computation and remote communication. A BSP process will consist of many of these supersteps, where each superstep in the same process is executed in turn but the supersteps of different processes are executed in parallel. At the end of a superstep all of the processes bulk synchronise and continue with the execution of the next superstep.

The BSP model allows either all, or any subset of, the processes to synchronise. These sets of synchronising threads can be viewed as groups of processes, with the supersteps for processes within the same group being synchronised, and those within different groups running asynchronously.

During a superstep a processor can access its own local memory, perform computation, and access the global memory of other processors. Access to local memory follows the same protocol as with sequential machines, i.e. the result of a local memory access is available immediately. Global memory accesses follow the BSP memory protocol.

2.2.2 The BSP Memory Protocol

The BSP memory protocol defines how local and global memory may be accessed inside a single superstep. The memory model can be interpreted in one of two ways, which we will call "single superstep mode" and "double superstep mode".

Single Superstep Mode

This can be defined as follows:

1. Accesses to local memory are performed immediately and need not wait until the end of the current superstep.
2. Read accesses to global memory locations are performed immediately. The process performing the read is blocked until the results are ready.
3. Write accesses to a global memory location are only guaranteed to be complete by the start of the next superstep.

We call this "single superstep mode" since in a single superstep a remote memory location can be read and its value examined.

Double Superstep Mode

Double superstep mode is essentially the same as single superstep, except that read accesses are not performed immediately:

1. Accesses to local memory are performed immediately.
2. Read and write accesses to a global memory location are only guaranteed to be complete by the start of the next superstep.

To read and examine a remote memory location takes two supersteps in "double superstep mode". In the first superstep the data is requested, then a global synchronisation is performed, and in the second superstep the remote variables value can be examined.

While the definition of the BSP model in [Valiant, 1990, Valiant, 1989] implies double superstep mode a more liberal interpretation could include single superstep mode. Single superstep mode

has some advantages in terms of algorithm readability and can lead to a quicker execution if only one variable is to be read[1]. It may also be the case that a compiler could perform a read optimisation and issue the request ahead of the actual point where the value is needed. These issues will be discussed further in subsequent chapters.

2.2.3 Global Communication, g

The global communication parameter, g, is used to judge the number of steps that will be required to implement an h-relation, giving the communication complexity of a superstep. The definition of g and h that we will use follows that of [Hill *et al.*, 1996], and can be described as follows:

Definition 2.0: *g is the value such that an h-relation is implemented in $g * h$ steps.*

For example, if each process received h words during a superstep then the communication complexity is $g * h$. Similarly if each process sent a message of size $\frac{h}{p}$ to processor 1 then the cost is still $g * h$ since processor 1 will receive a total of $p * \frac{h}{p}$ words.

The value of g for real architectures can be measured by timing the routing of arbitrary h-relations for large enough h. It is obviously the case that small messages will incur a significant penalty in terms of the overhead of accessing the communications network, while in large messages the limiting factor is the bandwidth of the machine. For this reason we presume that h is large enough so that we will be measuring the asymptotic performance of the communications system.

The assumption that an h-relation will cost $g * h$ is reasonable for large values of h. In such situations the time of an h-relation will be relative to the amount of data each processor has to transfer, or if all communications are directed at one processor then it is this processor that will dictate the total time of the h-relation.

While this method of using the asymptotic value of g is valid for judging the cost of an algorithm it does not allow the accurate prediction of performance for small h. In [Miller, 1994] the effect on g of using messages of size x is modelled using the following equation.

$$g(x) = \left(\frac{N_{\frac{1}{2}}}{x} + 1 \right) g_\infty \tag{2.1}$$

Here $N_{\frac{1}{2}}$ is the value of x such that g is half that of the asymptotic value g_∞. i.e. $g(N_{\frac{1}{2}}) = 2g_\infty$.

Using this equation we can now see that the cost of an h-relation is $h * g(h)$. For suitably large h

[1] Reading a single remote variable may be quicker than a global synchronisation.

this is $h * g_\infty$, which is $h * g$ if we define the parameter g of the BSP model to be g_∞.

When analysing the cost of a BSP algorithm we may choose to use g_∞ or $g(x)$ depending on the accuracy required. For the analysis of algorithms g_∞ would be sufficient, but for the accurate prediction of performance on real architectures using $g(x)$ gives improved results.

2.2.4 Bulk Synchronisation, l

There must be a cost associated with the bulk synchronisation that occurs at the end of every superstep. This synchronisation cost, l, can be defined as the number of steps that it takes to execute an empty superstep, i.e. a superstep that routes a 0-relation and performs no local computation.

2.2.5 Performance Prediction

Using these definitions for g and l we are now in a position to define the cost of a complete superstep. Suppose that a BSP program consists of p processes. Consider a single superstep and let c_i be the cost of the local computation performed by process i. If in this superstep an h-relation is implemented then the cost of the superstep is at most:

$$\max_{i=1}^{p}(c_i) + g * h + l \tag{2.2}$$

This assumes that there is no overlap of computation and communication. If overlap does occur then the cost can only be reduced by at most a factor two, even if we ignore synchronisation[2].

The cost of the BSP program as a whole is simply the sum of each of the supersteps. If there are n supersteps and h_k is the h-relation implemented in superstep k and c_ik the computation performed by process i in superstep k then the cost of the complete algorithm is:

$$\sum_{k=1}^{n}(\max_{i=1}^{p}(c_{ik}) + g * h_k + l) \tag{2.3}$$

In these equations we have assumed the asymptotic value of g. They can be easily modified for a more accurate prediction by replacing g with $g(h)$ as defined in equation 2.1.

[2]The cost if communication and computation could be overlapped is $max(\max_{i=1}^{p}(c_i), g * h) + l$

2.3 BSP Algorithms

The development of BSP algorithms is an active area of research [Goudreau *et al.*, 1994, Goudreau *et al.*, 1996, McColl, 1994a, McColl, 1994b]. In the following sections we shall illustrate the BSP model using two algorithms taken from [McColl, 1993b, McColl, 1993a]. Many other BSP algorithms have been developed covering the areas of scientific computing [Bisseling and McColl, 1993], computational geometry [Siniolakis, 1996a], sorting [Siniolakis, 1996a, Gerbessiotis and Siniolakis, 1996a] and other primitive operations [Gerbessiotis and Siniolakis, 1996b, Gerbessiotis and Valiant, 1992].

2.3.1 String Edit Distance

Consider two strings $X = x_1x_2...x_m$ and $Y = y_1y_2...y_n$ where x_i and y_j are the characters of X and Y at positions i and j respectively. The edit distance is defined as the number of insertions and deletions of characters that are required to change X into Y, and is a measure of the similarity of the two strings.

This problem can be easily solved using sequential dynamic programming. Let us define X_i as the substring $x_1..x_i$, and Y_j as $y_1..y_j$. We now define $DIST(X_i, Y_j)$ as the edit distance of the strings X_i and Y_j. The edit distance of X and Y is now described by $DIST(X_m, Y_n)$.

It is easy to see that $DIST(X_i, Y_0) = i$ and $DIST(X_0, Y_j) = j$. We can now define the following recurrence for $i > 0, j > 0$:

$$if\ x_i = y_j\ then\ DIST(X_i, Y_j) = DIST(X_{i-1}, Y_{j-1})$$

$$if\ x_i \neq y_j\ then\ DIST(X_i, Y_j) = 1 + min(DIST(X_i, Y_{j-1}), DIST(X_{i-1}, Y_j))$$

A sequential algorithm for the string edit distance problem could use the above recurrence to calculate the two dimensional array $DIST$ in a diagonal fashion as shown in figure 2.2. This requires $O(m + n)$ diagonals to be computed, giving a total cost, T_s, of O(mn).

A parallel algorithm for string edit distance is performed by assigning a vertical slice of the $DIST$ array to m processors. This will require a total of $O(m + n)$ supersteps to compute $DIST(X_m, Y_n)$. Note that not all processors will be computing values for the first $n - 1$ supersteps.

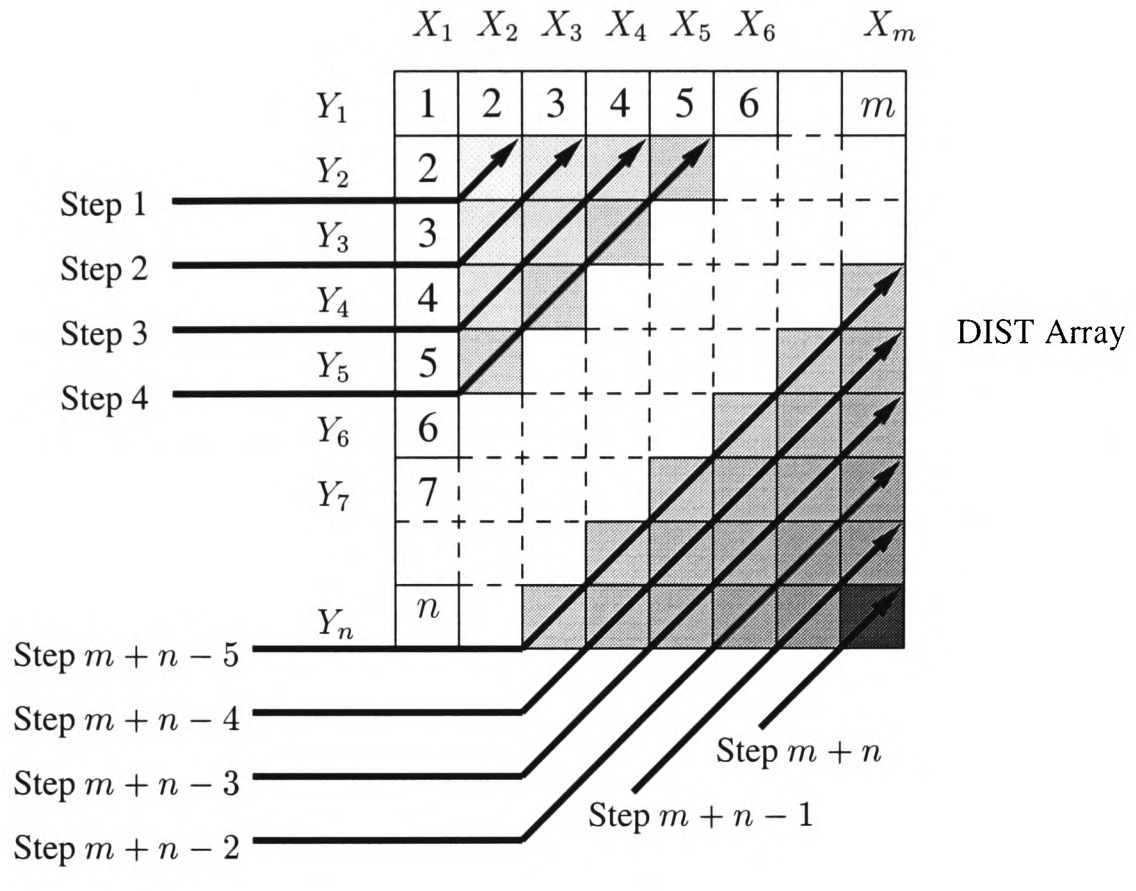


Figure 2.2: Sequential Edit Distance

Assuming the arrays $DIST$, X and Y are held in the global memory then each processor, i, would execute the following BSP pseudo code:

---

begin "initial" **superstep**
  – Initialise DIST(i,0), DIST(0,j), read myX and myY
end superstep
for j **in** $\{2 \,..\, m+n\}$ **loop**
  begin "loop" **superstep**
    y = j - i
    if $y > 0$ **and** $y <= n$ **then**
      if $myX = myY_y$ **then**
        $DIST(i,y) = DIST(i-1, y-1)$
      else
        $DIST(i,y) = 1 + min(DIST(i, y-1), DIST(i-1, y))$
      end if
    end if
  end superstep
end loop

In this algorithm the main loop is required to be a superstep to guarantee that the value of $DIST(i, y)$ is updated before it is used by another process. For example, in the first superstep process 1 calculates $DIST(1, 1)$, after which we must synchronise in order to ensure that process 2 reads the correct value of $DIST(1, 1)$ when it calculates $DIST(1, 2)$.

Performance Prediction

In order to predict the performance of the BSP edit distance algorithm we must first expand on the locations of the arrays $DIST$, X and Y in the global memory. For this example we shall assume the following:

1. $DIST(i, j)$ is held on processor i, $\forall j.0 \leq j \leq n$.
2. X and Y are initially held in the global memory of a single processor. During initialisation each processor receives its own local copy of the Y array in myY and the X_i element in myX.

We can now see that the "initial" superstep implements an $(n + 1)$-relation and initialises two arrays of size m and n respectively. If we assume that an assignment costs c then the cost of this superstep is $O((m + n)c + (n + 1)g + l)$.

In the "loop" superstep each processor will access two locations in $DIST$ that are not on the local processor, therefore implementing a 2-relation. In the worst case there are two comparisons, an addition, two assignments and a minimum. If we assume that high level operations such as addition, comparison and minimum cost the same as an assignment then this superstep has total cost $O(6c + 2g + l)$.

Since the "loop" superstep is executed $O(m + n)$ times the total cost, T_p, of the algorithm is:

$$T_p = O((m + n)c + (n + 1)g + l + (m + n)(6c + 2g + l)) \tag{2.4}$$

This is a rather simplistic view of the performance of the string edit algorithm but it still gives a reasonable estimate of its performance.

For a PRAM machine with $c = g = O(1)$ and $l = 0$ we can see that $T_p = O((m + n)c)$. If m and n are approximately equal then for a PRAM $T_p = O(\frac{T_s}{m})$. This is an optimal speed up for m processors.

For a realistic parallel machine it would probably be the case that c is much less than g and l, causing the main loop to be communication bound and therefore providing a very poor parallel algorithm. There is also the issue that we must have as many processors as there are characters in the X string i.e. m. This can be resolved by using a coarse grained variant of the algorithm where each processor takes a number of vertical strips of the $DIST$ array. Now we only require communication at the boundaries of the strips of $DIST$, and much more computation can be done in the inner loop.

An implementation of the coarse grain string edit distance algorithm is given in later chapters. The runtime of such an algorithm is profiled on a real machine, and the performance analysis is shown to be very close to reality.

A more scalable implementation of the string edit problem uses the diamond DAG technique [McColl, 1996a, McColl, 1996b]. If we assume that $n = m$ then using this method we split the $DIST$ array into p^2 blocks of size $\frac{n}{p}$. To compute the string edit for each block costs $\frac{n^2}{p^2}$ computation steps, with each processor sending and receiving $\frac{n}{p}$ elements. Using the diamond DAG scheduling requires p supersteps in order to compute the complete string edit, therefore giving an overall cost of $O(\frac{n^2}{p} + ng + pl)$.

Using the BSP cost model to compare algorithms in such a way is an important aspect of BSP. In the above simple analysis we have concluded that the naive solution to the string edit problem is inherently non-scalable, and that a much better solution is to use the diamond DAG scheduling technique.

2.3.2 Matrix Multiplication

Here we shall consider the problem of finding the matrix C formed by the multiplication of two n x n matrices A and B using p processors [McColl, 1994a, McColl, 1993a]. Each processor will compute a block of size $\frac{n}{\sqrt{p}}$ x $\frac{n}{\sqrt{p}}$ of the matrix C using the normal sequential algorithm. Figure 2.3 shows the three matrices laid over each other, and the values each processor will require.

This BSP algorithm can be performed in two supersteps. In the first superstep each processor fetches the required values from A and B. The second superstep then uses these values to compute the matrix multiplication of its portion of C.

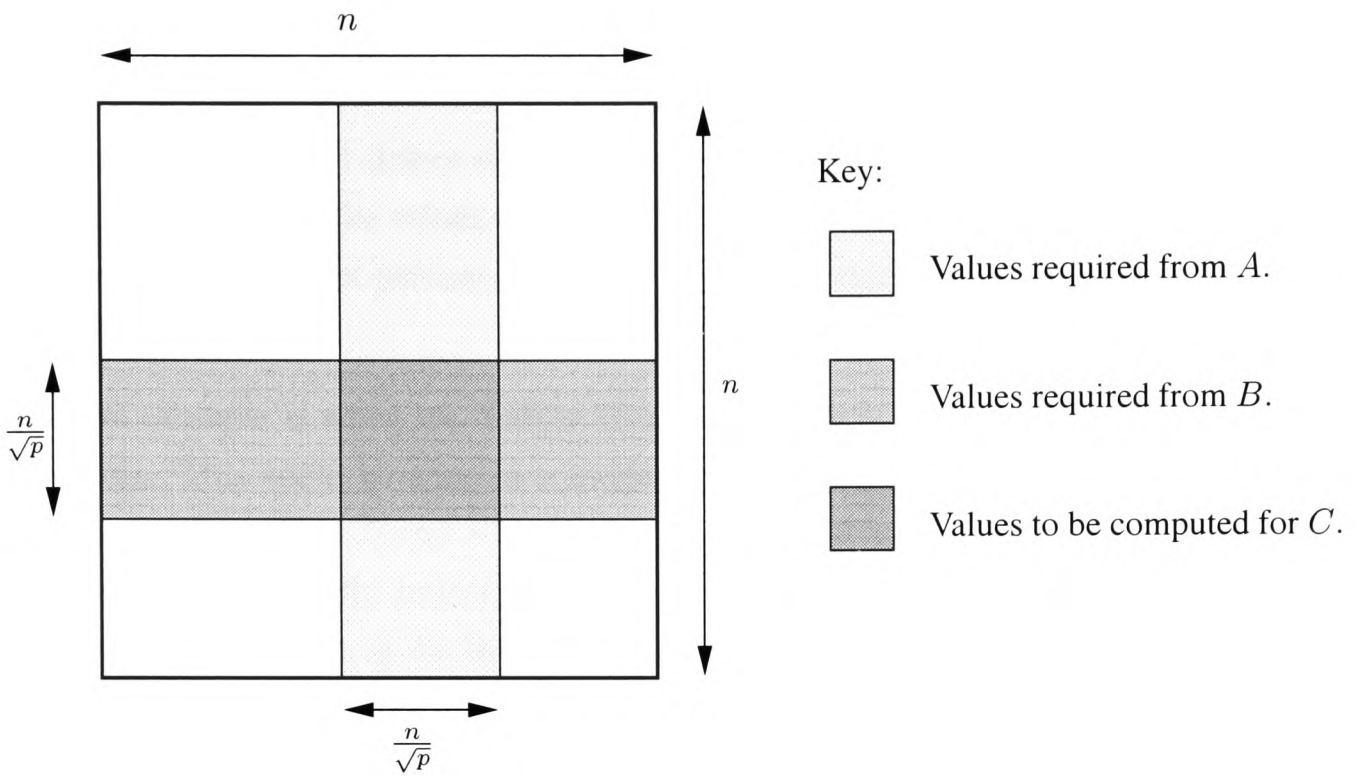


Figure 2.3: Matrix Multiplication

Performance Prediction

Let us assume that the matrices A and B are distributed evenly across all processors such that each processor holds $\frac{n^2}{p}$ elements.

In the first superstep each processor will read $\frac{n^2}{p}$ values from $\sqrt{p}$ processors for the vertical strip and the same for the horizontal strip. Each processor will also be asked to supply the values of the block that it has been allocated. The block is of size $\frac{n^2}{p}$ and will be required by $O(2\sqrt{p})$ processors. Therefore the total number of words sent and received by each processor will be $O(\frac{2n^2}{\sqrt{p}})$.

In the second superstep, the sequential cost for computing the multiplication of two $\frac{n}{\sqrt{p}}$ x $\frac{n}{\sqrt{p}}$ matrices will be $O(\frac{n^3}{p})$.

Summing the cost of these two supersteps we see that the cost of the whole algorithm, T_p, is:

$$T_p = O\left(\frac{4gn^2}{\sqrt{p}} + \frac{n^3}{p} + 2l\right) \tag{2.5}$$

For a PRAM machine with $g = O(1)$ and $l = 0$ we can see that $T_p = O(\frac{n^3}{p})$. This is an optimal speed up when compared to the sequential cost of multiplying two n x n matrices ($O(n^3)$).

2.4 Adaptive Algorithms

In this section we shall demonstrate how it is possible to make BSP algorithms adapt their behaviour depending on the values of the BSP parameters. This mechanism allows the same source code to achieve the best performance possible for a range of architectures, eliminating the need for substantial tuning[3].

A simple example is when we wish to broadcast n words from processor one to all of the other processors. This could be done in a single superstep where each processor reads n words from processor one, costing $O(npg + l)$. Another approach is to use a two stage broadcast [Hill *et al.*, 1997a, Hill *et al.*, 1997b, Juurlink and Wijshoff, 1996]. This method evenly distributes the data across the p processors in the first superstep, with each processor receiving $\frac{n}{p}$ elements. In the second superstep each processor then broadcasts their $\frac{n}{p}$ elements to every other processor using a naive single step copy. Summing these two supersteps gives us a total cost of $O(2ng + 2l)$.

For parallel architectures that have a very poor barrier synchronisation, and when n and p are small, the first approach may be better than the two stage algorithm. However, for large n or p the two stage approach will give improved performance. An adaptive BSP algorithm would use these costs[4] to determine which method is best to use for a given n, p, g and l. While this is a simple example it illustrates how BSP algorithms can make runtime decisions based on the architecture they are running on.

A more interesting example would be the determination of how may processors to use for a matrix multiplication using the algorithm described previously. Using equation 2.5 and differentiating with respect to the number of processors we obtain the following.

$$\begin{aligned} \frac{dt}{dp} &= -\frac{2gn^2}{p^{\frac{3}{2}}} - \frac{n^3}{p^2} \\ \frac{p^2}{n^2} \cdot \frac{dt}{dp} &= -2g\sqrt{p} - n \end{aligned}$$

We can see that $\frac{dt}{dp} = 0$ when $p = O(\frac{n^2}{4g^2})$. This is intuitively correct, since as the value of g improves more processors can be assigned to the task. For a PRAM, whose value of g is O(1), the number of processors almost equals the number of elements.

[3] It would still be possible to fine tune an algorithm for a specific architecture. The purpose of adaptive algorithms is to keep these adjustments to a minimum, allowing a stable base of software to be developed.

[4] With all constants accounted for.

Using this analysis we could say that an adaptive BSP algorithm should use $O(\frac{n^2}{4g^2})$ processors when performing a matrix multiplication of size n x n. However, there is a problem with this method. In our analysis we have assumed that g and l are constant and independent of p. For current parallel architectures this is often not the case, and g and l are increasing functions of p. We could replace g and l with functions of p but then our analysis will become dependent on the target architecture. In [McColl, 1993a] it is suggested that for a 2D array g may increase at the rate of $O(\sqrt{p})$, but for a hypercube it may increase at $O(1)$.

Although static analysis on the optimum number of processor may not be possible we may be able to perform such functions at runtime. At this point the runtime system knows exactly how many processors are available, and knows the values of g and l for different values of p. It may be possible to request that the runtime system minimise a function by evaluating it for all possible values of p, returning the value which led to a minimum. Using this method we would request that the runtime system issue us with the optimum number of processors for the algorithm we wish to run. This is discussed further in later chapters.

Of course, the converse analysis can also be performed i.e. if we fix the number of processors that are available then we can determine the optimum problem size, n, to run on a particular machine. In this case we can use static analysis methods since the values of g and l will be fixed.

2.5 Summary

We have described how the BSP model consists of a number of processes executing supersteps. Current interpretations of the model assume a "double superstep mode" where each read request is not blocked, but is guaranteed to be ready by the beginning of the next superstep. In some situations it may be more convenient to use "single superstep mode" where the requesting processor is blocked on a read until the results are available. This could lead to a more efficient algorithm, and perhaps simplifies algorithm design. In the majority of cases however double superstep mode is the preferred method of operation.

The BSP model provides a convenient framework for the analysis of algorithms where simplicity can be balanced with accuracy. For the comparison of algorithms it is reasonable to use the asymptotic value of the global communications performance, g_∞. However, if performance prediction is required then using equation 2.1 will lead to much improved results. Such comparison of estimated performance against actual results has already been show to be successful on conventional parallel architectures [Knee, 1994b, Hill *et al.*, 1996] and embedded transputer processors [Knee, 1994a]. In later chapters we shall show the results of comparing the estimated performance of an

Opal algorithm against its actual performance.

The BSP string edit distance and matrix multiplication algorithms have been presented and analysed. While these are simple examples they serve as an illustration of the usefulness of the BSP model. The literature on BSP algorithms is rapidly increasing, and the reader is directed towards [Gerbessiotis and Siniolakis, 1996b, Gerbessiotis and Siniolakis, 1996a, Siniolakis, 1996a, Gerbessiotis and Valiant, 1992, Goudreau *et al.*, 1994, Goudreau *et al.*, 1996, McColl, 1994a, McColl, 1994b, McColl, 1993b, McColl, 1993a, Bisseling and McColl, 1993] for further information.

Finally it was shown how BSP algorithms can automatically adapt to the architecture they are running on. This automatic adaption of algorithms has been previously mentioned in [Knee, 1994b, Cheatham *et al.*, 1995] but real analysis of this method is yet to be performed.

Chapter 3

Opal Design Rationale

3.1 Introduction

This chapter describes the issues that were considered during the design of Opal - an architecture independent parallel language designed for the BSP model. From the previous chapters we can see that the design of such a language should include the following goals:

- A portable language capable of running on a variety of parallel architectures with the same ease of portability as sequential languages.
- The language constructs should aid in the cost analysis of algorithms and should not affect the performance prediction properties of the BSP model. The algorithm designer should be able to easily recognise when remote communication is being performed so that its cost can be accounted for.
- A programming team should be able to produce libraries of parallel algorithms. It must be possible to use these libraries in new algorithms with the minimum of effort, i.e. code re-use.
- The language constructs should allow the compiler to make efficient decisions in the code generation phase, e.g. re-ordering of communications or synchronisation.
- The language should allow the compiler to detect as many errors as possible at compile time. This must include the correct checking of inter-process communication.

In subsequent sections we shall discuss these issues and describe how they have been incorporated into Opal. We shall also discuss the relative merits and disadvantages of existing parallel languages and libraries.

3.2 Programming the BSP Model

The Opal language was designed around the BSP model, giving the inherent features of predictable performance and portability. There are however a number of variants on how the BSP model can be programmed, how inter-process communication is performed and how the BSP parameters are made accessible to the programmer. These issues are discussed in the following sections.

3.2.1 Automatic vs Direct Algorithms

In the original definition of the BSP model two modes of programming were suggested [Valiant, 1990]; automatic and direct mode. In the automatic mode of operation the compiler is left to make the decisions regarding the placement of shared variables, allowing the programmer to use a PRAM style of algorithm design.

It may be the case that an automatic compiler would use parallel slackness to ensure that the most efficient use is made of the parallel machine. Parallel slackness is when there are more processes than processors on a machine, and is a frequently used technique for the hiding latency [Valiant, 1990].

Automatic mode may also make use of randomised routing [Valiant, 1982]. This method of routing first sends a packet to a random destination, which then routes it to its real destination. This method offers very good worst case performance, and can alleviate bottlenecks that would otherwise exist if direct routing was employed. A similar effect is achieved if instead of using randomised routing the automatic compiler randomly places variables in the shared memory.

The opposite to automatic mode is direct mode. Here the programmer is left to make all of the decisions regarding shared variable placement, and may or may not decide to use parallel slackness.

BSP C [Miller, 1993], BSPlib [Hill *et al.*, 1997c] and the Green BSP library [Goudreau *et al.*, 1995] all use the direct mode of programming. GPL [McColl and Miller, 1995] is a hybrid of automatic and direct mode - the programmer may claim a variable as being local, but variables

may be left unclaimed and therefore placed by the compiler.

While automatic mode leaves the programmer with an easier task it tends to hide the cost details of an algorithm. For example, different compilers may use different methods of distributing the data, or if randomised methods are used then the same compiler may produce different results on every compilation. This is not suitable for real time systems where the timing constraints are critical and the algorithm must *never* take longer than the predicted worst case. Such unpredictable behaviour is also present with languages such as Jade [Lam and Rinard, 1991, Scales *et al.*, 1991] and HPF [Forum, 1993]. In both these languages the actual cost associated with performing a given communication may be hidden from the algorithm designer, causing them to incorrectly assume their algorithm is efficient.

To ensure that Opal programs are as predictable as possible, and to allow then to be used for real time embedded systems, it was decided that the direct mode of programming was most appropriate. It has also been suggested that automatic mode is more suited to parallel architectures that have a very good value of g [Gerbessiotis and Valiant, 1992]. While parallel architectures are improving it is still the case that the majority of them do not have small values of g, making the direct mode of programming more suitable for today's machines.

While direct mode leaves the majority of the work to the programmer it ensures that the compiler does not make bad decisions regarding the placement of variables. Since the programmer must allocate each shared variable, and define its placement, they have complete knowledge of the distribution of data allowing them to predict the performance of the algorithm more accurately. It may also be the case that the programmer can use the properties of the algorithm to determine a better data placement than an automatic method could do.

3.2.2 Inter-Process Communication

As an Opal process executes its supersteps it must be able to communicate data with other processes. Such communication could use shared memory [Koelbel *et al.*, 1990] or a message passing scheme similar to that available in BSPlib[1] and the Green BSP library (GBSP).

The concern with message passing is that it introduces the notion of a sender and receiver, a concept that often causes programming errors in the form of mismatched send and receive pairs These mismatches are fatal in languages such as OCCAM-3 as they cause deadlock. In BSPlib and GBSP this per message synchronisation has been relaxed, resulting in a message passing mechanism which can never cause deadlock. It could be argued that while this has removed the

[1] BSPlib allows both shared memory and message passing.

possibility of deadlock, it could lead to algorithms that are even more difficult to debug, e.g. messages that are sent but never consumed could be difficult to detect.

An obvious candidate instead of message passing is shared memory, as used with BSPlib, GPL, FORK and HPF. Shared memory, like bulk synchronous message passing, can never lead to deadlock. It can, however, be quite difficult to debug and must be incorporated into a language with care.

The main advantage that shared memory has is that the communication is one-sided, i.e. a process can read the value of a remote variable without the cooperation of the target process[2]. For example, to change a shared memory algorithm so that a process updates a single element of a remote array is very easy, but to do the same with message passing is much more error prone.

For these reasons Opal uses shared memory as its inter-process communication medium. While shared memory has its disadvantages, it offers algorithm designers a convenient form of programming with improved software maintenance issues.

It would be possible to allow Opal processes complete access to the address space of other processes, as implemented in BSPlib. By using the `bsp_push_register()` function any variable in BSPlib can be a candidate for shared memory communication. In GPL any variable that is in scope when a process is created can be used as a shared variable. While this leads to a flexible shared memory mechanism it makes it difficult for the programmer to determine if a variable is involved in shared memory or not.

On some parallel architectures it may also be the case that there is a limited amount of real shared memory available. If the compiler could detect exactly which variables were going to be shared then it may be able to use specific features of the architecture to improve performance. Conversely, if a variable is known never to be shared then it can be implemented in the most efficient way, e.g. possibly held in a processor register.

In order to provide the Opal compiler with the maximum opportunity for optimisations, and to ensure that shared memory accesses are easily visible to the algorithm designer, Opal variables must be declared either shared or local.

Opal shared variables can be accessed by any other process. They can be considered as a service that a process is willing to offer to other processes. However, local variables cannot be accessed by other processes and any attempt to do so would result in a compile time error.

To implement this feature Opal uses a notion of process specifications and bodies. This can be

[2] It may require cooperation in the implementation of the shared memory, but at the high level language level no cooperation is required.

likened to the specifications and bodies of Ada [Barnes, 1989] and Modula-3 [Harbison, 1992] libraries, although at a finer granularity.

An Opal process specification lists all of the shared variables of a process and only these variables can be accessed by other processes. Other processes would access them using methods similar to that of the module interface of Ada and Modula-3 i.e. a process would import the specification of another process in order to use its shared variables. This also provides a simple method of data placement - any shared variable declared in the specification of a process is guaranteed to be held in the local memory of the processor that it executes on.

A process body contains the actual code to be executed by a process, just as the module body of Ada contains implementations of the procedures it has exported in its specification. Any variables declared inside the process body are local to that process and cannot be accessed by other processes. This is exactly the same mechanism that is used to declare local procedures in modular languages such as Ada and Modula-3.

The Opal process specification and body constructs let the compiler check that only shared variables are accessed by other processes and uniquely identifies those variables that are shared. It allows local variables to be implemented in the most efficient way for specific architectures, while also allowing the compiler to reorder communication since it knows exactly which shared variables are in use.

3.2.3 Access to the BSP Parameters

In order for an algorithm to be portable and efficient it must be able to adapt to the parameters of a given parallel architecture. In the case of BSP this requires that the programmer has access to the g, p and l parameters of the architecture. The programmer may then use these parameters to determine the size of internal data structures, how many processes to use and other decisions described in section 2.4 on adaptive algorithms.

Since Opal uses the BSP machine in the direct mode it would be the responsibility of the programmer to incorporate use of the parameters into their algorithms. For example, instead of using p processes in a library the programmer may code it to use $\frac{p}{g*n}$, where g is the usual BSP parameter, and n is an indication of the problem size. This equation would have been calculated by analysing the algorithm and determining the optimum number of processors.

Opal provides access to the BSP parameters in the form of variables, e.g. $BSPp$ is a variable whose value is the number of available processors, with $BSPg$ and $BSPl$ defined in a similar

way. To allow the programmer to be as accurate as possible the $N_{\frac{1}{2}}$ parameter is also available as the $BSPN12$ variable[3].

These values will change depending on the number of processors in use and so the above variables are defined with respect to using $BSPp$ processors. To allow the programmer to create more adaptive algorithms access is also given to functions parameterised by p. For example, $fBSPN12(p)$ is the value of $N_{\frac{1}{2}}$ when p processors are used and $fBSPg(p)$ the value of g for p processors.

The fact that algorithms can dynamically re-configure themselves at runtime necessitates that we have data structures that can be dynamic in size. Many languages such as C restrict data structures so that their size must be known at compile time. C algorithms can bypass this restriction by using the heap, but since Opal does not make use of pointers we must ensure that data structures can be declared whose size is not known at compile time. For example, we should be able to declare an array whose size is $BSPp$.

3.3 Integrating Libraries into BSP

A library is a collection of source code that performs a specific service for the library user. Such a library may sort an array, multiply two matrices or perform a prefix sums computation[4].

Libraries should be considered as black boxes. This is to say that the user of a library may not know how the services are implemented and it may be beneficial to hide these implementation details. This can be performed in a variety of ways, but the common point is that a library offers a set of services, or interface, that the user incorporates into their algorithm.

In order to combine libraries with the BSP model a number of issues must be addressed, including the following:

- Since the flow of control is passing from the user's algorithm to the library we must consider the effects of nested supersteps, and whether this feature should be allowed.
- Should the user's flow of control pass through the library, or should we make the library mechanism active, i.e. do we have passive libraries where the user's flow of control performs the work, or active ones where we request work to be performed.

[3]The value of g_∞ in Opal is $BSPg$.

[4]The reader should not confuse this with BSP libraries such as BSPlib and the Green BSP Library. In this section we are discussing how to incorporate libraries, or modules, into a BSP language.

- Sequential libraries have one point of access since there is only one flow of control. Parallel libraries consist of one or more processes with many flows of control and yet we would like to present a single uniform interface to the user.
- Should the shared variables of the library exist in the same space as the user's shared variables.

In the following sections we shall discuss these issues in more depth, and arrive at a library mechanism for Opal programs.

For the library mechanism to be most effective, and to follow good software engineering principles [Sommerville, 1989], the implementation of the library should be transparent to the user. It may even be the case that the implementation of the library changes over time, and yet its interface remains the same. Such changes in the library's implementation should not require the user to change the source code. For this reason Opal libraries present a single uniform interface such as in Ada and Modula-3. The implementation of a library may then change, but as long as the interface remains constant there is no need to change the user's source code.

3.3.1 Nested Supersteps

Nested supersteps may occur when a procedure or library function is called from within an existing superstep. If the procedure that is called uses its own supersteps then we must consider the semantics of having supersteps executing within a parent superstep. Such nested supersteps are allowed in GPL, but not in BSPlib or the Green BSP library.

It should be noted that nested supersteps can only be created in BSP languages that use different mechanisms to mark the beginning *and* the end of a superstep. For example, BSPlib uses the function `bsp_sync()` to mark the end of one superstep and the beginning of another. In this library it is not possible to create nested supersteps since inserting an extra `bsp_sync()` will simply split the current superstep into two separate supersteps[5].

Nested supersteps will require a change in the memory model of BSP. For example, when we enter a nested superstep do we allow the sibling supersteps to access the variables of the parent, or do we require them to declare their own shared variables? If the sibling supersteps can access the shared variables of the parent superstep then we must define when these variables are updated, i.e. are they updated at the end of a sibling superstep, or at the end of the parent superstep? GPL

[5]However, in BSPlib it is possible to inadvertently end the current superstep by calling a function that uses `bsp_sync()`.

addresses this issue by using the `sync(n)` function. This synchronises the nested supersteps up to n levels backward from the current point.

For these reasons nested supersteps are not allowed in Opal groups[6], and any attempt to use them will create a compile time error. Opal procedures can use supersteps, but the compiler ensures that these are never called from within an existing superstep.

3.3.2 Passive vs Active Libraries

A passive library is one in which the user's flow of control passes through the library, whereas in an active library the user requests a service be performed but this work is done by another independent flow of control. These two methods can be likened to the passive library structure of C, versus the active mechanism of remote procedure call.

In the parallel environment an active library may be a collection of processes that are waiting for a signal to perform a service. These library processes would then perform the required work and signal back to the calling process to say that the task has been completed.

To aid in the discussion of active and passive libraries figure 3.1 shows the different types of calling mechanisms that may be employed by a programmer. The arrows represent user processes calling a library routine, *where each user process is in the same superstep.* In an all-to-all call each user process p_i requests work to be performed by a corresponding library unit l_i. In contrast is a one-to-all type call, where a single user process requests a library function, which may be implemented using many library processes. Finally, in an any-to-any library call different user processes may require the functions of different libraries. Note that with sequential libraries the only possible call mechanism is all-to-all, since there is only one user flow of control.

An example of an all-to-all calling mechanism is when the user requests a parallel sort to be performed, but where the data is already distributed across the user's processes. If the data was not distributed across the user's processes, but instead the library performed this function, then a one-to-all mechanism would be employed. In an any-to-any mechanism the different user's processes may be using two libraries - one to perform a sort and another to perform a matrix multiplication.

It can be immediately seen that passive libraries only support all-to-all and any-to-any type calls, since in a one-to-many call there is only one thread of control being passed to a library which contains many threads. Another problem that a passive library mechanism would have is that

[6]Nested supersteps do appear in group communication, and are considered in later chapters.

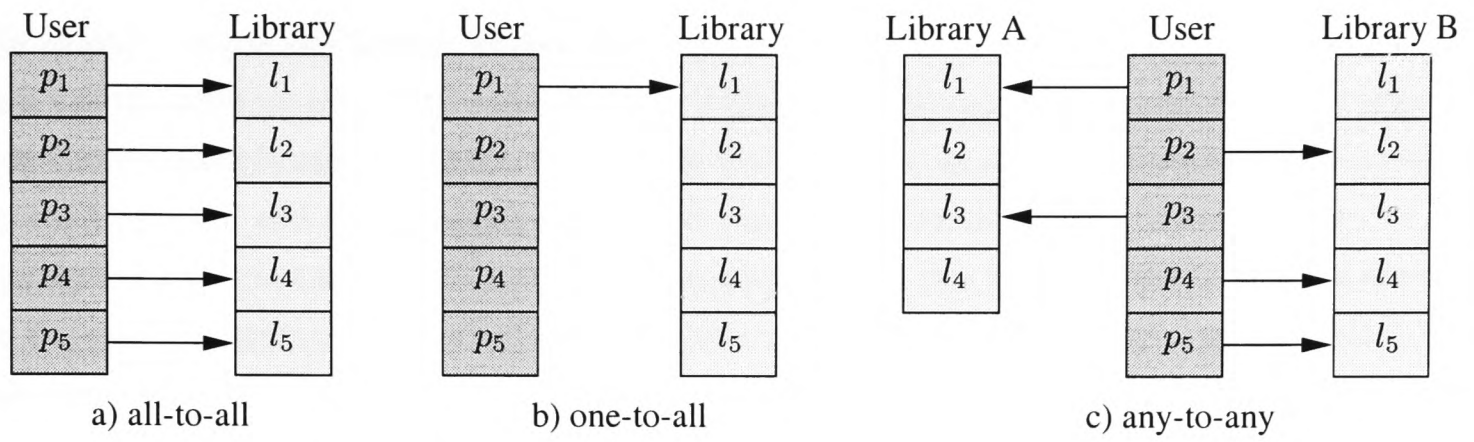


Figure 3.1: Library Calling Mechanisms

each user process would have to execute the same superstep structure once inside the library. This would be reasonable for a all-to-all call, but for any-to-any it requires that library A and library B both execute the same number of supersteps. Of course, if library A executed less supersteps than library B, then the processes that called library A could be padded out with "dummy" supersteps. This requires knowledge of the implementation of both libraries, and hence breaks the black box principle.

With active libraries all forms of calling mechanisms can be supported. However, there is still the problem of ensuring that the supersteps of the libraries do not interfere with the supersteps of the user processes. This can be solved by classing an active library as a separate subset of synchronising processes. In this case it does not matter if in an any-to-any call different libraries perform different numbers of supersteps, since in this case the supersteps of the library are separate from those of the user process[7].

Since Opal libraries must be black boxes the calling processes know no details about the operation of the library processes. However, the library processes must be able to communicate with each other in order to accomplish the desired task, requiring some form of shared memory. In the case of passive libraries these shared variables must either be integrated into the shared variable space of the calling processes, or created on demand when the library is called.

With active libraries the shared variables would be self contained, i.e. the processes of a library would be declared just like any other process, including the definition of shared variables. These active library processes would perform some sort of housekeeping duty at start up and then remain dormant until requested to perform a service.

We must also remember that the library structure is likely to be hierarchical in large applications.

[7]The supersteps of the two different libraries would also be independent of each other.

For example, a parallel sorting library may use the services of a bucketing library, which may in turn use the services of a prefix sums library. With passive libraries this hierarchical decomposition further complicates the issues of ensuring that the same number of supersteps are executed by each process, and incorporating the shared variables of the libraries. Active libraries would simply use a set of processes for each library module. While this leads to more processes running on the parallel machine they would only be active when required to perform a service and would otherwise be dormant.

A further benefit of active libraries is that they are separately executing algorithms, i.e. they have their own shared variable address space and perform superstep synchronisation only with the processes in their own library. We can consider active libraries as a black box of processes, and need not make any changes to the user's source code if the library implementation changed.

Active libraries also fit neatly into the the definition of the BSP model. In [Valiant, 1990] a BSP machine is described as a set of processes in which any subset can bulk synchronise. If we consider each active library to be a separate subset then we require no changes to the BSP model in order to incorporate parallel libraries.

For these reasons an active library mechanism was consider to be more appropriate for Opal.

3.3.3 Library Interfaces

Opal libraries offer services that will be implemented using parallel algorithms, requiring that each library has a number of independently executing processes. The user of such a library need not, and should not, know exactly how many processes are used or how they should communicate with each process. Instead, a uniform interface to all of the library processes should be offered, requiring that a single file describes all of the services of that library.

While in real terms the library calling mechanism may be all-to-all or one-to-all, the user of the library sees a uniform interface. In other words each service in the library may have a single entry point (one-to-all), or multiple entry points (all-to-all), but the user does not know how these entry points are mapped to actual library processes.

This not only improves the black box features of Opal libraries, but also simplifies the task of determining what services a library is actually offering.

3.4 The BSP Memory Model

In systems such as BSPlib the user is given a range of options regarding the amount of buffering that is required for a remote variable access. For example, the user may perform a remote write where the remote variable is updated immediately, or they may choose to have the remote variable updated *only* at the end of the superstep. The following sections discuss how these memory model issues apply to Opal.

3.4.1 Single vs Double Superstep Reads

In section 2.2.2 the notion of single and double superstep reads was introduced. This section attempts to clarify the relative pros and cons of these two methods.

To review what was described in section 2.2.2, a single superstep read is when a process issues a read request and then blocks until the data is ready. In double superstep mode the process is not blocked, and it is guaranteed that the read request will have been answered by the start of the next superstep. In both modes the writing of remote variables is the same, i.e. remote writes do not block the calling process and are only guaranteed to have occurred by the beginning of the next superstep.

While double superstep mode is the most common interpretation of the BSP model it does have a number of problems, including:

- Should the read be performed exactly at the end of the superstep, i.e. is the local variable we are reading the remote value into available for other use in the same superstep, or will the read be performed at any point up until the beginning of the next superstep. If the result of the remote read can be returned at any point then we cannot use the local variable after we issue the remote read. BSPlib solves this by requiring the user to specify the buffering scheme.
- Delayed reading requires that an end of superstep is encountered before the result of the read is guaranteed to be available. It effectively requires the programmer to insert a synchronisation.
- Will algorithms coded in this delayed read style be easy to understand and maintain?

In single superstep mode there is no issue regarding the buffering of data on a remote read[8]. While

[8] Although we must still consider the buffering of data on remote writes.

single superstep reads may not be strictly adhering to the BSP model, they should be considered due to the benefits of decreased global synchronisations and algorithm maintainability.

There are also disadvantages associated with single superstep reads. The first is that the prediction of algorithms employing this method is complicated and would probably require two different values of g depending on whether we are reading or writing. It would also be the case that if a process was using a large number of reads in the same superstep then the communications network will not be efficiently utilised since each read would cause the process to block. Using double superstep mode would allow the process to flood the network with read requests since it does not wait for the results of the request.

Single superstep reads also prevent us from merging messages destined for the same processor, and only issuing them at the end of a superstep. This method of merging, which is described in the next section, offers improved performance on many types of networks, including those with high overhead and high bandwidth, e.g. fat pipes.

For these reasons it was decided that Opal should use double superstep reads. This leads to a more predictable language which will perform better in situations of large communications traffic. The Opal compiler that has been produced does allow, via a compiler flag, the use of single superstep reads, allowing this issue to be investigated further.

3.4.2 Merging h-relations

Consider a process that reads two shared variables from another process. These two read requests can be sent as two separate messages or merged into a single message. It is noted in [Skillicorn *et al.*, 1996, Hill and Skillicorn, 1997] that existing parallel architectures often have a large overhead associated with transferring a message, but once this transfer has been initiated the high bandwidth of the links can be utilised. For this reason the buffering of h-relations into a single message at the end of the superstep is likely to be more efficient than sending multiple smaller messages.

While merging the messages is more efficient, it means that we have to wait until the end of a superstep before any communication can be performed, or at least wait until some specified number of messages have accrued in a buffer. Sending the messages immediately allows for more overlap of communication and computation, at the expense of the overhead due to sending two separate messages. This computation and communication overlap complicates the accurate prediction of algorithms since the amount of overlap must be accounted for.

We may also not have the opportunity to merge h-relations, as the semantics of the reads and

writes may not allow it. For example, if we were to employ single superstep reads then we must send the read request immediately, giving no opportunity for merging.

Since the use of merging increases the efficiency of communications, and improves performance prediction, it was decided that the Opal compiler should merge all communications. If the user has significant reasons why this merging should not be performed then it can be disabled by the use of a compiler flag.

3.4.3 Shared Variable Buffering

Since we have decided that all communication in Opal will use double superstep mode we must now discuss the issues of buffering. This is the problem of where to put data that arrives in response to a write or read request.

Writing Shared Variables

A process may perform a write to a shared variable at any point during a superstep. This write request may or may not be sent to the target process straight away. If the request is sent immediately then we must consider what the target process should do when it receives the data - it could buffer the data until the end of the superstep, or perform the write straight away.

This obviously affects the way the programmer writes an algorithm. If the semantics of the shared variable write guarantees that the remote variable is not updated until all processes have synchronised then the programmer may use that variable during the superstep in which it is written. If the write could occur at any point then the programmer must note that variable as volatile and not use it during the superstep.

Reading Shared Variables

A similar situation occurs during the reading of remote variables. In this case if the read request is issued immediately and a response from the target process arrives back before the end of the superstep, then we must decide where to place the data. Again, we could place it directly into the local variable that the programmer designated during the read, or buffer the data until the end of the superstep.

If we perform the read immediately then the programmer must be aware of this, and must ensure that the local variable is never used after it participates in a shared variable read.

3.4.4 Opal Shared Variables

It has already been decided that Opal should use the double superstep mode of communication, perhaps employing the merging of h-relations. It has also been stated that all shared variables in Opal will be easily recognisable and that if a variable is declared as local then it can never be accessed by another process.

We could now offer the programmer different language constructs depending on the exact buffering required, as done with BSPlib. While this method is obviously the most flexible it leads to a complicated language. Indeed, it could be the case that offering such flexible buffering increases the chances of a programmer mistakenly using an incorrect scheme.

It was decided that shared variables in Opal can be updated at any point during the superstep, and that the programmer should not rely on when they are updated, i.e. a remote write can update the target location at any point during a superstep, and the results of a remote read can be placed into a local variable at any point. This seems reasonable since shared variables, and their access, is easily identified in Opal source code.

3.5 Dynamic vs Static Processes

A language that allows processes to be created and destroyed during the execution of an algorithm has a dynamic process structure. In contrast is a language where processes are created during an initialisation phase after which the number of processes is static.

Dynamic processes complicate the run time system of any language, often leading to a less efficient implementation. In [Skillicorn *et al.*, 1996] it is noted that most distributed memory architectures do not allow dynamic process creation[9]. Such process creation and destruction can also incur a significant penalty on some parallel architectures and this cost is not captured by the BSP cost model. However, the use of dynamic processes can make algorithms more readable and easier to describe.

Opal uses a compromise of these two methods where the static process structure is kept, but the number of processes started during the initialisation phase may be dependent upon a runtime variable[10]. For example, it would be possible to specify that $BSPp$ copies of a process are to be started up during the initialisation phase.

[9] Such architectures include the IBM SP2, Cray T3D, Meiko CS-2, Parsytec GC and Hitachi SR2001.

[10] Or an environment variable.

3.6 Barrier Synchronisation

The BSP model defines a barrier synchronisation as a point at which all processes join together and global memory is updated. In [Miller, 1993] it is noted that such a synchronisation may require two synchronisations at the implementation level; failure to do so could result in one process beginning its next superstep and issuing a read request to a process which is still updating its global memory, resulting in the wrong value being fetched. While Opal does not insist on the implementation details of the barrier synchronisation, it does guarantee that such race conditions cannot exist.

There has also been some discussion in the BSP community as to including a higher level synchronisation primitive. A possible use of such a primitive would be to ensure that all processes executed the same number of supersteps during the execution of a loop. Without this primitive processes that finish the loop early would be required to implement "dummy" supersteps, waiting for the other processes to catch up. Such a mechanism is used in the **if** statement of FORK. While such synchronisation primitives could be useful we believe them to be at a higher level than the Opal language is targeted at.

3.7 Static Source Checking

The number of compile time checks that can be performed on source code significantly reduces the development time of large software projects. To aid in such compile time checks the Opal language must have the following properties:

- Strong type checking, including the type checking of remote variable assignments and library calls.
- Array bound and shape checking, enabled by default but with the ability to disable.
- Ensuring that the local variables of one process cannot be accessed by another.

The quality of such checks will depend on the quality of the compiler, but we must ensure that language constructs that require runtime checks are only introduced if absolutely necessary.

3.8 Summary

In this chapter we have discussed the issues that have influenced the design of the Opal language. We can summarise the main decisions as follows:

- Opal uses the direct mode of the BSP machine. While this places more of the decisions with the programmer it provides a more accurate framework for performance prediction. This method is also better suited for applications in real time systems.
- Remote communication must be easily identifiable at the source level and must not hide any costs from the programmer. An example of a hidden cost is the **REALIGN** statement in High Performance FORTRAN.
- Opal processes consist of a specification and a body. Any variable declared in the specification is classed as being in global memory and may be accessed by other processes. Variables declared in the body of a process are guaranteed to be local.
- An active library mechanism was chosen as this offers the most flexibility and resolves issues regarding the superstep synchronisation of the user and library processes. Libraries must offer a single interface, even though there may be multiple entry points to the functions of the library.
- Opal provides a range of variables and functions for access to the BSP parameters of the target machine, allowing algorithms to be as adaptive as possible. A side effect of this is that we must allow data structures to be declared whose size is not known at compile time, e.g. an array of size $BSPp$.
- A static process structure has been adopted. In this framework the number of processes that are created during the initialisation phase may be variable, but once this phase has completed no new processes can be created.

Following these goals will lead to a language which is convenient for the programmer, follows good software engineering principles, offers predictable performance and implements a parallel library mechanism for maximum code re-use.

Chapter 4

The Opal Language

4.1 Introduction

In this chapter we provide an introduction to the Opal language and demonstrate how we have achieved the goals set out in the design rationale. It assumes the reader is familiar with basic sequential constructs found in imperative languages such as C [Kernighan and Ritchie, 1988], and has an understanding of the principles of modular languages such as Ada [Barnes, 1989] and Modula-3 [Harbison, 1992].

We first provide an overview of the Opal language, introducing the key features of Opal processes, process communication and groups. The remaining sections describe each of the language constructs in turn, illustrated with the use of actual source code.

This chapter concentrates on the parallel issues of the Opal programming language. For a complete description of the sequential features of Opal the reader is directed to chapter A.

4.2 Overview

4.2.1 Processes

An Opal process describes an independently executing task that may be placed on any of the available processors. Every process in an Opal program must be a member of one, and only one, group. Processes that are in the same group execute their supersteps in bulk synchronisation and

may communicate with each other via shared memory[1]. Processes that are in different groups do not bulk synchronise and cannot share memory.

As described in section 3.2.2, a process must be able to declare variables that are exclusive to itself and cannot be accessed by other processes. These variables can be read and written immediately, and do not follow the BSP memory protocol. Processes must also have a set of variables that are declared as shared, and can be accessed by any member of the group. These shared variables follow the BSP memory protocol and are not guaranteed to be updated immediately.

Since Opal programs use the BSP machine in direct mode, the programmer must specify where shared variables are to reside. This is performed by splitting the definition of an Opal process into a specification and body.

In Opal, a process specification may only contain the declarations of variables. Any variable that is declared in a specification is classed as being in shared memory, making it accessible by all other processes. It is also guaranteed that the variable will reside in the address space of the process that declared it, allowing the programmer to sensibly place data rather than letting the compiler automatically assign locations.

A process body contains the supersteps that are to be executed by the process. Any variables that are declared inside the process body are exclusive to that process, and cannot be accessed by other processes.

The use of process specifications and bodies can be likened to the way packages are declared in Ada. In this language, variables declared in the specification are "public" and accessible by users of the package, while variables declared in the package body are "private".

4.2.2 Process Communication

We must now consider how the shared variables of a process are referenced by other processes. This is done using the same mechanisms that Ada and Modula-3 use for incorporating existing modules into new programs. We effectively class a shared variable as a service that a process is willing to offer.

If a process requires the use of the shared variables of another process then it must *import* that process's specification. Importing a process's specification does not cause a second set of memory locations to be created for the shared variables, but instead gives the process access to another

[1]Through out this thesis we use the term "communicate via shared memory" to refer to one-sided communication into remote variables, as opposed to real shared data structures.

process's shared address space. It is comparable to using an extern declaration in C, i.e. it indicates that a variable is to be used but space for it has been allocated elsewhere.

Opal uses dotted notation to precisely specify the shared variable that is being referenced, a mechanism that is used in Ada and Modula-3. This method uses the notation *process.var* to reference a variable called *var* that is declared in the specification of *process*. This prevents conflicts that may occur when two specifications contain a variable of the same name, and indicates to both the programmer and compiler exactly which process holds the shared variable.

4.2.3 Groups

In section 3.3 it was described how the library mechanism in Opal must be active. This defines a library as a set of self contained processes that offer specific services, e.g. matrix multiplication. This is incorporated into the Opal language using the concept of a group.

As stated previously, every Opal process must be a member of a single group. If this group is to perform a library service then the user of the library should not need to know the details of the processes that implement it. For this reason, and in accordance with the design rationale, we require that a group has a single interface. If a user wishes to use the services of a group then they need only consult the group interface.

In Opal this mechanism is provided by using a level of abstraction above that of processes. A group is split into a specification and body in a similar way to that of processes. The body, or implementation, of a group is the collection of processes that are a member of it. The specification of a group is a separate file that describes the services that this collection of processes is willing to offer.

An Opal group specification is a contract between the group user and the group implementor. An added complexity above that of sequential libraries is that the implementor of an Opal group wishes their library to be a parallel algorithm, requiring that multiple processes are to be used. For this reason a group encapsulates a number of processes, and offers a single unique interface to them. As with Ada and Modula-3, the actual implementation of the group is hidden from the user and may change over the life cycle of the software. However, such changes in the implementation of a group will not require any changes to the user's source code, as long as the group interface remains the same.

We must remember that processes in separate groups do not bulk synchronise with each other, and cannot share memory. This is in accordance with the design rationale, and prevents problems

caused by nested supersteps and any-to-any library calls. However, if an active library is to be of any use then we must be able to communicate with it. Such group communication is discussed in the following section.

4.2.4 Group Communication

Group communication is at a much higher granularity than that of process communication. For example, two processes may be required to exchange a single integer with each other, and may perform this exchange many times during the same superstep. In contrast to this is a group that will be requested to perform library tasks such as sorting, matrix multiplication or I/O. In this case, when a process requests another group to sort some data then there will be many process communications compared to a single group communication[2], i.e. the result of one group communication is likely to lead to the execution of many supersteps.

A group offers a number of services that it is willing to perform, for example, a matrix group may offer the services of matrix multiplication, inversion and transformation. We must know which task is to be performed, and what data it is to be performed upon. In Ada and Modula-3 this library mechanism is accomplished by declaring procedures that accept data as parameters. When the procedure returns from its call, the parameters to it have been updated to reflect the algorithm that has been applied. It should be noted that when a procedure is called in these languages it is the user's thread of control that actually runs through the procedure, i.e. a passive mechanism.

An Opal group could also offer a number of procedures that can be called, but this could be easily confused as a passive mechanism, rather than active. In Opal only the data is transferred to the library group, not the thread of control.

Group communication could be performed with shared memory in a similar fashion to process communication. The problem with this is that we would need to describe exactly which shared variables are required to be set for each library routine the group offers. Procedures neatly package this data together in the list of formal parameters, i.e. it is the list of parameters that specifies the data that is required for the library routine. We would also require some higher level synchronisation primitive in order to guarantee that group shared memory is updated at the correct points.

For these reasons Opal groups use *entry* points that are based on the Ada rendezvous mechanism [Barnes, 1989]. An entry point looks similar to a procedure specification, but the key difference is that only the data is transferred to an entry, not the thread of control. When one process calls an

[2]The process communications are from the processes that implement the sort group.

entry point, and a group accepts it, a rendezvous is said to have occurred.

A key difference between the Opal and the Ada rendezvous is that in Ada the calling process is blocked while the rendezvous is executed. At the end of the rendezvous the results, in the form of updated parameters, are available immediately. Opal rendezvous follow the same delayed mechanism as shared memory updates in the BSP model. This is to say that an Opal process is not blocked during the rendezvous, and is allowed to continue with its next statement in the superstep. However, the results of the rendezvous are not guaranteed to be available until the beginning of the next superstep.

4.2.5 Program Execution

An Opal program can consist of many groups, with each group containing a number of processes. We must define how the processes of these groups are started, and in what sequence. In order to describe the execution of an Opal program we must first define what it means for a group to require the services of another group:

Definition 4.0: *Group P requires group Q if any process that is a member of P uses a service of group Q, and Q is not shared.*

The definition of a shared group is given in section 4.5.9, but at the moment we shall assume that all groups are exclusive and not shared.

An Opal program must have a single group which is designated as the *main group*. The processes in this main group are the first to be started at execution time. This concept is the same as the `Main` module in Modula-3.

The first stage in the execution of an Opal program is that the main group enters an *initialise* phase. During this phase the processes of the main group are started, as are any groups that the main group requires. These subgroups will then enter the initialise phase and will start their own processes and any groups that they require. This initialise phase continues in a recursive fashion until all required groups are started[3]. At this point all processes enter the "execute" phase and start executing the code defined in their bodies.

An important aspect of the initialise phase is that before an Opal program starts executing, all of the processes in all groups are active. After the initialise phase no new processes or groups can be spawned, although they can terminate. This structure follows the static process mechanism described in section 3.5.

[3]It is guaranteed that the *initialise* phase will reach a fixed point since group specifications cannot be recursive.

It should be noted that there may be multiple copies of the same group running. For example, if group P requires groups Q and R, and group R also requires group Q, then two copies of Q will be started during the initialise phase. This guarantees that a group has the exclusive use of its subgroups, and that contention for the services of a group is limited to processes that are members of the same group[4].

The purpose of a shared group is to avoid the spawning of multiple copies of the same group. If a group is declared as a shared group then only one copy of it will be started during the initialise phase, and all groups will share it. Shared groups are useful for libraries that provide access to a single resource, e.g. the system console.

4.2.6 Process and Groups Example

Consider an Opal program that is to calculate the convex hull of a number of points. The programmer creates a group called *chull* and defines a set of processes that are members of this group, and that will perform the convex hull algorithm. It is possible that as part of the convex hull algorithm we need to perform a parallel sort of a number of coordinates in a plane.

The programmer may have access to a pre-existing sorting group called *sort*, and may decide to use this for part of the convex hull algorithm. Unknown to the programmer is the fact that the *sort* group actually uses a parallel prefix group to help implement the sort algorithm.

The execution of this program would proceed as follows. First the main group, *chull*, would be started along with all of the processes who are a member of this group. *chull* requires the use of *sort*, so this group and all of its processes would be started. As *sort* enters its initialisation phase it will start all of its processes and the *parprefix* group. After *parprefix* has started its own processes the programs initialisation phase would be complete as all required groups have been started. The Opal program would then start executing the supersteps in the body of each process.

It is important to note that a group performs a service but only when requested. For this reason you would expect that the processes that are members of the *sort* and *prefix* groups will be dormant until they are required to perform work[5].

Figure 4.1 shows how these processes could be allocated on a parallel architecture consisting of three processors.

---

[4] Group contention is therefore limited to processes that are bulk synchronising.

[5] You may expect a group to perform a few housekeeping tasks first and then block waiting for a rendezvous. Only one process need block on a rendezvous in order for the whole group to stop as that process will not reach the end of its superstep.

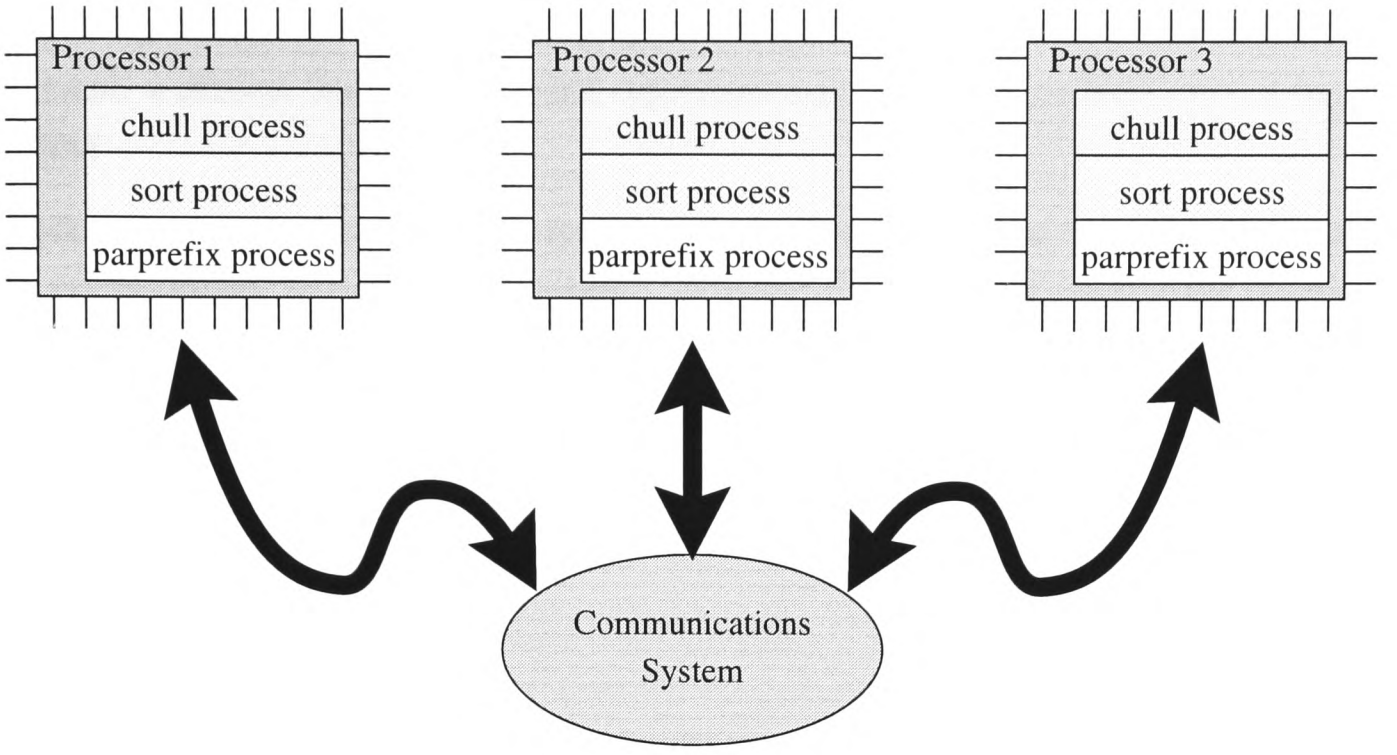


Figure 4.1: Opal Processor Allocation

Note how each processor runs a process from each of the groups, and that it is likely that at any point in time only one of these processes will be executing.

With this overview in mind we can now progress to the definition of the Opal language.

4.3 Processes

In the following sections we shall describe how to declare process specifications and bodies, a concept that was introduced in section 3.2.2. The language constructs for supersteps and inter-process communication are then introduced, and a discussion on process termination is given.

4.3.1 Process Specifications

A process specification contains two pieces of information:

1. Which group the process is a member of.
2. The variables that this process is willing to share with other processes.

A process specification serves no other purpose except to define the above information. On the other hand, a process body defines what supersteps a process is to perform once it has begun executing.

The following code is an example of a specification for a process named $master$, that is a member of the $prefix$ group. It declares two variables, $data$ and $flag$, that will reside in the address space of the $master$ process, but which may be accessed from any other member of the $prefix$ group.

```
prefix process master is
    data : array (1..200) of integer;
    flag : boolean;
end master;
```

In parallel algorithms it is often the case that a number of identical processes need to be started in an SPMD fashion. This can be performed in Opal using a replicated process such as:

```
prefix replicated process slave (i: 1..10) is
    dataSlice: array (1..20) of integer;
end slave;
```

This specification is for 10 processes named $slave$ that are all members of the $prefix$ group. Each $slave$ is given a unique value of the replicated variable i in the range 1 to 10. An array variable named $dataSlice$ is also declared, and each of the 10 executing versions of the $slave$ process will contain a $dataSlice$ variable in its address space.

The replicated variable, i, can be referenced inside the process body so that the different $slave$ processes can perform different functions. A process may also be replicated across multiple variables such as (i: 1..3, j: 1..4). The range of the replicated variables need not be constant, and can use the BSP parameters described in section 4.7, as well as expressions and built in functions. For example, $1..\mathbf{floor}(\mathbf{log2}(BSPp)/BSPg)$ is a valid expression for a replicated range. This method of replicated variables allow SPMD processes to be created in a structure that matches the problem, while also allowing algorithms to be adaptive in the number of processes they use.

A group may have any number of replicated and non replicated processes, allowing a MIMD style of execution. Of course, a group may also contain a single set of replicated processes thus giving

an SPMD style of execution.

It should be noted that the variables declared in the process specification follow the BSP memory protocol, and are only guaranteed to be updated at the end of a superstep.

4.3.2 Process Bodies

A process specification declares information about the process and the shared variables it contains. A process must also have supersteps to execute, and these are defined in the process body.

Variables declared in a process body are not visible to any other processes. These variables are accessed in the normal fashion, and do not follow the BSP memory protocol. This method of distinguishing shared and local variables follows the principles of inter-process communication described in section 3.2.2.

An example of a replicated process body is show below:

---

replicated prefix **process body** slave (i: 1..10) **is**
    flag : **boolean**;
begin
    supersteps
end slave;

A non replicated process body would be declared in the same fashion except the keyword **replicated** is not used and a replication variable is not given. For each process specification there must also be a matching body that obeys the following rules.

1. A replicated specification must have a replicated body and the replication variables must match exactly.
2. A non replicated specification may only have a non replicated body.
3. The specification and body must be members of the same group.

It is a compile time error if these rules are violated.

Typically the process specification and body would be held in separate files. For example, the current compiler implementation expects to find a specification of the *slave* process in the file slave.spl, and its body in slave.opl.

4.3.3 Supersteps

A process body consists of a sequence of supersteps, where processes in the same group synchronise at the end of each superstep. A superstep is defined in Opal using the following syntax:

---

declare
    declarations -- Declare variables whose scope is the superstep. Can be omitted
begin superstep
    statements
end superstep;

Any statement that accesses remote variables must be contained inside a superstep. Statements are allowed outside of supersteps but they must only operate on local variables. For the reasons given in section 3.3.1, nested supersteps are not allowed. If a program attempts to reference remote variables outside of a superstep, or attempts to nest supersteps, then a compile time error is given. The following segment of code illustrates how statements can be placed outside a superstep:

---

for i **in** 1..10 **loop**
    begin "loop " & **string** (i) **superstep** --A named superstep
        statements
    end superstep;
end loop;

In this case the statement that is outside of the superstep is the **for** loop, and since this only references the loop variable it is perfectly valid. The same mechanism can be used to put supersteps inside other control statements such as **if**, **case**, **while** and **loop**.

This example also demonstrates how supersteps can be named by placing a string expression between the keywords **begin** and **superstep**. This string expression could be a constant such as *"initialise"* or an expression such as in the above example, with the expression being evaluated each time the superstep is executed. String expressions, including the concatenation operator &, are described in chapter A.

The purpose of a named superstep is to increase the effectiveness of runtime error messages, and to improve code readability. For example, if the above code referenced an array outside of its

bounds then the error message could be:

Process "slave": Error in superstep "loop 2": array overflow of "matrix" at (3,4).

Any expression that evaluates to a string can be used to name a superstep. It should be noted that supersteps that synchronise do not have to have the same name.

4.3.4 Shared Variable Access

Processes must have a method of accessing the shared variables of other processes. This is performed by *importing* the specification of a process, and then referencing its shared variables using *dot notation*.

For example, consider the body of a replicated process named $slave$ that wishes to access the shared variables of a process named $master$, and the shared variables of other $slave$ processes. Such an example is shown below:

```
with master;                    -- Import the specification of the master process.
with slave;                     -- Import the specification of the (replicated) slave process.
replicated prefix process body slave (i: 1..10) is
    local declarations
begin
    supersteps
    begin "compute" superstep
        x := master.flag;       -- Access the variable flag held by the master process.
        y := slave(4).data(3);  -- Access the variable data(3) held by slave(4).
        z := #slave.data(3);    -- Access the variable data(3) held by the current process.
    end superstep;
    supersteps
end slave;
```

The first point to note is that if a variable of another process is to be referenced, then that process's specification must be imported using the **with** keyword at the beginning of the process body. The same rule applies to replicated processes that wish to access the shared variables of other processes that are in the same replication.

Once a process's specification has been imported, its shared variables can be referenced using dot notation, e.g. $master.flag$ references the $flag$ variable held by $master$. In the case of

a replicated process the replicated index must be given in order to uniquely identify both the process and the shared variable, e.g. $slave(4).data(3)$ refers to the value of the $data$ array at the third index, in the $slave$ process with replication index value 4.

This method of accessing the shared variables of other processes is very easy to spot at the source level, i.e. a programmer can scan an Opal program and accurately determine where shared variable access is being performed. This allows the programmer to easily analyse the communication complexity of an algorithm, which was a requirement that was listed in section 3.1.

The final assignment in the *"compute"* superstep of the *slave* process demonstrates a local dotted component. This is used when a process wishes to access its own shared variables, but wants to do so in the most efficient way possible. Local dotted component accesses can be easily identified as they start with a #, e.g. $\#slave.data(3)$ refers to the value of the $data$ array at the third index in the current process. In the case of local dotted component access there is no need to include the replication index since the process being referred to is already uniquely identified.

Local dotted component access does not follow the BSP memory protocol, even though the variable being referenced is a shared variable. Instead local dotted component accesses are performed immediately, just as if they were an access to local variables.

A process can still access its own shared variables using the normal dot notation but this could lead to less efficient code, possibly forcing a process to send a message to itself. For this reason it is good practice to use local dotted component access where ever possible. Indeed, the ability to place shared variables on specific processes can only be fully utilised by using local dotted component access.

4.3.5 Shared Variable Buffering

The buffering scheme that is used for remote assignments is shown in table 4.1, with the terminology following that of BSPlib [Hill *et al.*, 1997c].

Assignment Type	Terminology	Buffering Mechanism
$x := p.y$	Unbuffered Get	$p.y$ is read at any point during the superstep, x is updated at any point during the superstep.
$p.y := x$	Unbuffered Put	$p.y$ is updated at any point in the superstep. The value used is the value of x at the point of assignment.
$p.y := p.x$	Invalid	Invalid assignment that is detected by the compiler.

Table 4.1: Opal Variable Buffering

Other languages offer buffering schemes such as buffered get and buffered put, e.g. BSPlib. For reasons described in sections 3.4.3 and 3.4.4, these buffering mechanisms have not been included in the initial specification of Opal, although section 7.4 discusses how they could be incorporated.

4.3.6 Process Termination

When an Opal program is executed it first enters an *initialise* phase during which all of the required processes of the program are started. We must now define what happens when a process decides to terminate.

A process will finish execution, and remove itself from memory, when both of the following conditions are met:

1. The process is ready to terminate, i.e. it has reached the end of its final superstep or has selected a **terminate** alternative in a **select** statement[6].

2. All other processes in the same group are ready to terminate.

If a process is ready to terminate, but there are still other processes in the same group which are executing, then the process will not terminate. Instead it will execute null supersteps until all other processes in the same group are ready to terminate.

This method of termination prevents shared variables from being deleted when a process reaches the end of its final superstep. If a process were to terminate immediately then all of the shared variables in its address space would cease to be accessible to other processes. Using the above method of termination guarantees that all shared variables are accessible until the last process in a group terminates.

4.4 Subroutines

In the following sections we describe how to declare subroutines in an Opal program. Subroutines can be recursive in nature and can be defined at any point in an Opal program where declarations are valid.

It should be noted that shared variables cannot be used as the actual parameters to a subroutine call. The reason for this is that the value of a shared variable is not available until the end of the

[6] **select** statements are described later in this document.

current superstep, and yet the subroutine needs to be executed immediately. Attempting to use shared variables as actual parameters is a compile time error.

4.4.1 Formal Parameters

Opal uses the parameter passing mechanism of Ada, where the formal parameters of a function or procedure can be passed in one of the three modes listed in table 4.2.

Mode	Method of reference
in	The actual parameters value can be examined in the subroutine, but not changed.
out	The actual parameters value cannot be examined, but it can be changed.
in out	The actual parameters value can be both examined and changed.

Table 4.2: Opal Formal Parameter Modes

The compiler ensures that **in** mode parameters are never updated, allowing them to be passed using the most efficient form possible, e.g. for scalar values this may be pass by value, but for large arrays it could use pass by reference. The **out** mode ensures that code cannot be written that depends on the input value of the actual parameter, while **in out** mode is standard pass by reference.

Any of the Opal data types, including records and arrays, may be passed as parameters. Procedures and functions may also be passed as parameters but only using the **in** mode.

4.4.2 Functions

Opal functions can return any of the scalar data types that were defined in section A.2. They may also return arrays or records, but they cannot be used to return other functions or procedures. An example of a simple function declaration is shown below:

```
function sum3 (x, y, z: in real) return real is
begin
    return x + y + z;
end sum3;
```

Functions can contain any number of **return** statements, but it is a runtime error if a function does

not return a value. The name of the function must be repeated in the final **end** statement, aiding the reading of large programs that contain many function definitions.

It should be noted that function bodies cannot contain supersteps.

4.4.3 Procedures

Opal procedures may be declared using one of two methods; *local procedures* and *superstep procedures*. A simple example of a local procedure declaration is shown below:

```
procedure swap (x, y: in out sort.T) is
    t: sort.T;
begin
    t := x; x := y; y := t;
end swap;
```

In this example the function *swap* exchanges the value of two variables. Such local procedures can be called anywhere in an Opal program without restriction.

As with function declarations, local procedures cannot contain supersteps and any attempt to do so will result in a compile time error. However, procedures may be declared that do contain supersteps:

```
superstep procedure scale (c: in real; m: out array (1..BSPp) of real) is
begin
    begin superstep
        statements
    end superstep;
    more supersteps
end scale;
```

The procedure *scale* is an example of a superstep procedure, which is the only type of subroutine that is allowed to contain supersteps.

In order to guarantee that supersteps are never nested, procedures must obey the following rules.

1. Superstep procedures can only be called from outside a superstep.
2. Local procedures can only call other local procedures. This rule is recursively applied all of the way down the call tree.
3. Superstep procedures cannot be used as actual parameters.

These rules can be checked at compile time, and guarantee that supersteps are never nested, whilst allowing the procedure abstraction to be used throughout an Opal program.

4.4.4 Unconstrained Arrays

An unconstrained array is a formal parameter in which the number of dimensions is fixed, but the size of each dimension is adjusted to fit the shape of the actual parameter in use at the time. The following is an example of a subprogram declaration that uses an unconstrained array:

```
procedure multiply (vector: in out array (<>,<>) of real) is
begin
    for i in first (vector) .. last (vector) loop
        for j in first(first(vector)) .. last(last(vector)) loop
            perform summation
        end loop;
    end loop;
end multiply;
```

The built in functions **first** and **last** are used to obtain the first and last index of the array *vector*. They may be used with either constrained or unconstrained arrays, allowing algorithms to be independent of the shape of an array.

Unconstrained arrays are a feature of many sequential programming languages, providing a mechanism for passing variable amounts of data to a subroutine. This feature is especially useful for adaptive algorithms, where the size of data structures may vary depending on the runtime value of the BSP parameters.

It should be noted that if any index of an array is unconstrained then all indices must be unconstrained. If the user attempts to define such a mixed array then a compile time error will be issued.

4.5 Groups

As defined in section 3.3, groups are the method by which modular Opal programs are built. They *glue* together many processes into a single entity, forming an interface that describes the combined services of the processes. In the following sections we describe how groups are declared and how they describe the services that they offer. To begin with we must first describe entry points, which is the method by which groups offer services.

4.5.1 Entry Statements

An entry statement describes a service that a group is willing to perform, and may only appear in a group interface.

```
entry sort (a: in array (<>) of real; c: out array (<>) of real);
```

The above example declares an entry point called $sort$ that accepts an array a as input and writes the results into an output array c. The modes of an entry statement are the same as those of a subroutine as defined in section 4.4.1. As with subroutine parameters, entry points may also use unconstrained arrays, providing a method of transferring variable amounts of data to the group.

4.5.2 Group Interfaces

A group interface combines a number of entry statements into a single interface. Group interfaces may also be used to declare group wide constants such as the size of the input data or the number of processors to be used. However, these are the only type of declarations that are allowed in a group interface, i.e. variables cannot be declared. The following is an example of a group named $arrayUtils$:

```
group arrayUtils is
    n : constant := 200;
    entry sort (a: in array (<>) of real; c: out array (<>) of real);
    entry mergesort (a, b: in array (1..n) of real; c: out array (1..n) of real);
end arrayUtils;
```

Groups do not execute code and are simply a mechanism for providing a single uniform interface to the services of a set of processes, a requirement of section 3.3. The group body can be thought of as the collection of processes that are members of this group.

4.5.3 Accept Statements

In section 4.5.1 entry statements were described as the mechanism by which groups offer their services. In order for a group to perform this service there must be a process ready to accept the entry point and do the required work. This is done with an **accept** statement, an example of which is shown below:

---

accept sort (a: **in array** (<>) **of real**; c: **out array** (<>) **of real**) **do**
    supersteps
end accept;

The above code executes an **accept** statement for the *sort* entry. Note that the process performing this accept must be a member of the group where the entry statement is defined, and the accept parameters must match exactly with that definition. The operation of the accept statement is defined in the following way:

1. The process executing the **accept** statement waits until a call, or rendezvous, is received on the entry point. If an entry point is called before a process is ready to accept it then the message is queued.
2. The parameters of the entry are communicated.
3. The body of the accept statement is executed.
4. At the end of the accept statement, any **out** or **in out** mode parameters are passed back to the calling process.
5. Execution then continues with the code after the **end accept**.

An important point to stress is that although the entry point is defined in the group interface, it is implemented, or accepted, by a process which is a member of that group. The purpose of defining it in the group interface is to provide a single unified interface to the group that does not require knowledge of who accepts an entry point, or how it is implemented.

It can be seen that rendezvous are a two-sided mechanism. To aid in further discussion let us define the process that is performing the accept as the *accepting process*, and the process performing the entry call as the *entry process*.

It should be noted how the accepting process is blocked if there is no entry process, while the entry process does not block even if there is no accept process available. The only guarantee that is offered is that an entry process will not complete its end of superstep synchronisation until an accept process becomes ready and performs the rendezvous.

Since the accept process is performing supersteps, it effectively blocks all processes that are a member of the same group, only continuing when an entry process is available. This is the mechanism by which groups become dormant, waiting to perform a service.

4.5.4 Select Statements

A **select** statement allows a process to wait for a rendezvous on any number of entry points. This is useful when a group implements many services, and so would contain more than one entry point in its interface. For example, a group that performs prefix sums may offer entry points for $allsums$ and $minimum$, utilising a **select** statement as follows:

```
loop
    select
        accept allsums (parameters) do body end accept;
    or
        accept minimum (parameters) do body end accept;
    or
        terminate;
    end select;
end loop;
```

Each option in a **select** statement must be either an **accept** or **terminate** statement. The **terminate** option specifies that if the parent group is terminating then the process is to select the **terminate** alternative and terminate itself. If such a selection is chosen then no other statements are executed after the **terminate**.

The above Opal code gives an example of how the **terminate** statement can be used. This example uses an infinite loop in order to repeatedly service a number of entry points. If the parent of this

group terminates then no process can rendezvous on the entry points, and so the loop should be exited[7]. Without this special **terminate** alternative an extra rendezvous point would have to be inserted in order to signal a request for termination.

The **accept** option in the **select** statement is used to specify that if a rendezvous is ready on this entry point then it is to be accepted, otherwise select another alternative. If no alternatives are available then the process will sleep until an alternative is ready, or in the case of this example, until the parent group terminates.

4.5.5 Restrictions on the use of Accept Statements

In order to simplify the performance prediction of rendezvous, a number of restrictions have been placed on the use of the **accept** statement. The reasons for these restrictions are described in chapter 5, but for now we shall just list them:

1. The body of an **accept** statement must contain supersteps.
2. Supersteps cannot be used before or after an **accept** statement.
3. The body of a process which is not a member of the main group must contain an **accept** statement.

From these rules we can deduce that **accept** statements cannot be nested, and that the only control structures that can surround them are those that rely on local variables, e.g. **select** and **while**.

4.5.6 Calling Entry Points

In the previous sections we have shown how to declare an entry point for a group and how a process that is a member of that group may accept a rendezvous. We must now describe how the entry points of a group may be used by processes in other groups.

Processes use the entry points of other groups in a similar manner to the way they access the shared variables of other processes. The following code is an example of how a process uses the services of a $arrayUtils$ group:

[7]Since groups are not shared, we know that if the parent group terminates then there are no processes that could perform a rendezvous.

```
with arrayUtils, bucket;
bucket process body slave is
    a, b: array (1..bucket.N) of integer;
begin
    begin superstep
        arrayUtils.sort (a, b);        – Rendezvous with another group.
    end superstep;
end slave;
```

The *sort* entry point of the $arrayUtils$ group is referenced using the same *dot notation* used for shared variable access. Again, this makes it easy for the programmer to identify where remote communication is being performed.

In this example the process called $slave$ is using the services of another group to sort the contents of an array a into another array b. At the point where the rendezvous is called the parameters are passed to the accepting process, but the calling process is not blocked. The accepting process then performs the work that is involved in sorting the array, and communicates the results back to the entry process. The entry process is then allowed to complete its end of superstep synchronisation.

The programmer would visualise such as rendezvous as the act of a process synchronising with the $arrayUtils$ group, where in reality it is two processes synchronising with each other. This illustrates how the group interface provides a single uniform interface to the services of a number of processes. There is no need for the programmer to know the details of the $arrayUtils$ processes, and they can simply consult the group interface to determine how the entry points should be used. This allows the implementation of the $arrayUtils$ group to change, and accomplishes a design goal that was described in section 3.3.

4.5.7 Arrays of Entry Points

The rendezvous mechanisms we have described so far provide a method of performing one-to-any library calls as defined in section 3.3.2. Using this mechanism for performing all-to-all or any-to-any library calls would be cumbersome, and so the concept of an array of entry points was created.

As an example let us consider a $arraysort$ group that is implemented in an SPMD fashion using $BSPp$ replicated processes. The user of such a group may already have the data distributed across a number of processes, and so a one-to-one library call would not be suitable. In this case

the *arraysort* group interface may be declared as follows:

```
group arraysort is
    entry sort (1..BSPp) (a: in array (<>) of real; c: out array (<>) of real );
end arraysort;
```

This example creates *BSPp* entry points called *sort*, all of which take an array *a* as input and an array *c* as output. Such arrays of entry points are used in exactly the same manner as normal entry points, except that an entry index must be supplied in the entry declaration, entry call and accept statement. Each replicated process in the *arraysort* group may now accept a different rendezvous depending upon the value of its replication index.

If the user of the *arraysort* group was also using *BSPp* processes then each process would call the entry associated with its replicated index, allowing an all-to-all calling mechanism[8]. Of course, using such mechanisms would also allow us to create any-to-any type library calls.

Arrays of entry points are an important feature of Opal, allowing the programmer to use the normal SPMD programming mode of BSP. Further examples of such entry points can be found in chapter 6.

4.5.8 Generic Groups

The goal of Opal groups was to provide a modular environment where programs can be *glued* together using algorithms that have already been developed[9]. An example given was that of a parallel sorting algorithm that could be re-used in a number of different applications, of which a sample group interface was described in section 4.5.7.

There are a number of reasons why this group would not be suitable as a library for performing sorting. The most obvious problem is that the group will only perform sorting on **real** arrays, while the algorithm for performing a sort on **integer** arrays would be very similar. This is solved in languages such as Modula-3 and Ada by the use of generic groups, a concept that has been adapted in Opal for use with parallel libraries.

The second problem with the *arraysort* group is that the number of processes has been fixed at

[8]Having the same number of processes in the user and library groups is a restriction that is addressed in later sections by the use of generic groups.

[9]In a manner which does not disturb the performance prediction properties of BSP algorithms.

$BSPp$. While some users may require that all processes are involved in the sort, there will be cases when only a subset of the available processors should be used, e.g. any-to-any type library calls.

A more subtle problem is that of the size of the input arrays. While unconstrained arrays have been used, allowing variable sized arrays to be passed as parameters, it is likely that each process in the $arraysort$ group will contain a shared array for holding such data. While shared variables can have a dynamic size[10], their declarations can only access constants that are in scope at the process specification level. This would require the designer of the $arraysort$ group to define some maximum size of array that they are willing to sort.

Generic groups can be used to address these problems. A generic group is an interface parameterised by generic variables and types, allowing the group designer to perform a higher level of abstraction. For example, the $arraysort$ group interface could be transformed to the following generic group:

```
generic group garraysort (maxn   : in integer;
                          nprocs : in integer;
                          T      : in type;
                          cmp    : in function (a, b: in T) return integer) is
    entry sort (1..nprocs) (a: in array (<>) of T; c: out array (<>) of T);
end garraysort;
```

This is an example of a generic sorting group which is parameterised by the maximum size of array to be sorted, the number of processors to use, the type of the array, and the comparison function used in the sort. Processes can now be written that are members of this generic group and that use the generic parameters $maxn$, $nprocs$, T and cmp. These generic parameters can be referenced by the group's processes using the normal dot notation.

Before a generic group can be executed, actual values for the generic parameters must be supplied. The act of supplying the actual values for the generic parameters is known as instantiation, an example of which is shown below:

[10]Dynamic in the sense that it does not have to be fixed at compile time, but is fixed once the algorithm starts executing.

```
with integers, garraysort;
group smallsort is new garraysort (20000, BSPp / 2, integers.T, integers.cmp);
end smallsort;
```

This example creates a group called *smallsort* which is an instantiation of the generic group *garraysort*. The *smallsort* group will use $BSPp/2$ processes to sort integer arrays of maximum size 20000. The *smallsort* group can now be used just like a normal group, and entry calls can be made to it. An Opal program may use many instantiations of the same generic group, all of which will be separately executing groups.

Generic groups increase the amount of code re-use that can be performed, while not affecting the efficiency of the implementation. They allow us to overcome some of the restrictions regarding Opal's static process structure whilst keeping the benefits of having a fixed number of processes at runtime.

4.5.9 Shared Groups

In section 4.2.5 it was stated that if a group P requires group Q, then group P will have its own exclusive copy of Q to work with. However, consider a parallel architecture where only a designated node could perform I/O with the console. It would be advantageous to define a group that could perform this I/O, and then have this group shared by all other groups[11]. Using the present Opal group constructs this would not be possible, as each group that used the I/O group would require its own separately executing version. Shared groups provide a mechanism for circumventing this default startup behaviour.

A shared group is similar to a normal, exclusive, group except that only one copy of it is started when an Opal program that uses it is executed. A shared group is declared by putting the keyword **shared** at the start of the group specification, such as in the following example:

```
shared group SystemIO is
    entry points
end SystemIO;
```

[11] This console I/O group would execute on the special node.

Shared groups complicate the performance prediction of Opal programs, as many groups of processes will be competing for the same resource. However, with exclusive groups only processes in the same group will be competing, making it much easier to predict. Therefore shared groups should only be used in circumstances where an exclusive group could not be used due to implementation restrictions. In later chapters there is a more detailed discussion of performance prediction in Opal, including the use of shared groups.

It should be noted that since more than one group will be using a shared group, the **terminate** alternative of a **select** statement would not work as expected. For this reason shared groups cannot use **terminate** alternatives, and programmers must devise their own group termination sequence.

4.5.10 Group Termination

A group terminates once all processes that are its members have terminated, where process termination is defined in section 4.3.6. Once a group has terminated, its entry points are no longer available and any process that attempts to use them will deadlock.

The initialisation phase of an Opal program, described in section 4.2.5, creates a tree of groups with the main group at the root. When the main group terminates, due to its processes terminating, a message is sent to all of its children informing them that their parent is terminating. Such a message can be considered as a *terminate rendezvous*, and any of the processes in the subgroups that are waiting on a **terminate** alternative will terminate. These subgroups then send terminate rendezvous signals to any groups that they started, continuing in a recursive fashion until the leaves of the tree are reached.

Such a termination mechanism will cause any processes that are waiting on a **terminate** alternative to finish execution. However, any processes that are not waiting on a **select** statement will not terminate, perhaps causing some groups to be left executing while their parent has terminated. While this may not affect future executions of the Opal program it does use up the resources of the machine and should be avoided. For this reason select statements with terminate alternatives should be used where possible or some other method of process termination must be employed[12].

[12]Such as "one shot" groups that only ever perform one rendezvous, or employing special rendezvous to signal termination.

4.6 Packages

Packages are employed where the library paradigm is required but where it is not necessary to have a parallel algorithm to perform the work. Packages in Opal are effectively a simplification of Ada packages, providing a useful mechanism for creating libraries of sequential code. Opal provides standard packages for the sequential sorting and merging of arrays, algorithms that are often used as the basis for a parallel solution.

As an example, consider a library that implements a number of mathematical functions such as the tangent of an angle and random number generation. While it would be good practice to have these services defined in a library it would be wasteful of resources to have a parallel algorithm compute them. Instead, the work should be done by the process using the package, where the thread of control of the calling process passes through the procedure defined in the package, i.e. a passive mechanism. An example of such a *math* package is shown below:

```
package Math is
    function tan (x: in real) return real;
    procedure seedRandom (seed: in integer);
end Math;
```

The code to execute when these procedures and functions are called is defined in the package body. A package body for the above specification would be defined in the following fashion:

```
package body Math is
    function tan (x: in real) return real is
        statements
    end tan;

    procedure seedRandom (seed: in integer) is
        statements
    end seedRandom;
end Math;
```

Every procedure and function defined in a package specification must have a matching declaration in the package body. It should also be noted that such bodies cannot contain supersteps, which in

turn prevents access to shared variables.

4.7 BSP Parameters

As described in section 3.2.3, in order to allow Opal algorithms to adapt to the architecture on which they are being executed on, the BSP parameters g, l and p must be available to the programmer. Table 4.3 shows the identifiers that are used to access these parameters. These identifiers may be used anywhere in an Opal program including group interfaces, replicated variable ranges and array definitions.

BSP Parameter	Opal Identifier	Type	Unit
p	**BSPp**	**integer**	-
g	**BSPg**	**real**	Flop per word
l	**BSPl**	**real**	Flop
$g(x)$	**BSPgx** (x)	**integer** $\rightarrow$ **real**	Flop per word
$x.g(x)$	**xBSPgx** (x)	**integer** $\rightarrow$ **real**	Flop
g as a function of p	**fBSPg** (p)	**integer** $\rightarrow$ **real**	Flop per word
l as a function of p	**fBSPl** (p)	**integer** $\rightarrow$ **real**	Flop
$g(x)$ as a function of p	**fBSPgx** (p, x)	**integer*integer** $\rightarrow$ **real**	Flop per word
$x.g(x)$ as a function of p	**xfBSPgx** (p, x)	**integer*integer** $\rightarrow$ **real**	Flop

Table 4.3: Opal BSP Parameters

The functions **fBSPg**, **fBSPl**, **fBSPgx** and **xfBSPgx** provide the values of g_∞, l, $g(x)$ and $x.g(x)$ for a given number of processors. The values of **BSPg**, **BSPl**, **BSPgx** (x) and **xBSPgx** (x) are relative to the current number of processors, and are equivalent to **fBSPg (BSPp)**, **fBSPl (BSPp)**, **fBSPgx** (**BSPp**,x) and **xfBSPgx** (**BSPp**,x).

Opal also provides a language construct called **optBSPp** that is used to minimise an expression that is dependent upon **BSPp**. This construct is best explained through the use of an example:

```
bestp = optBSPp (float (x / BSPp) + 2.0 * float (fBSPg (BSPp)))
```

This causes the expression that is passed to **optBSPp** to be evaluated for all values of **BSPp** in the range $1 \leq$ **BSPp** $\leq p$, where p is the maximum number of processors available for execution. The

value returned is the value of **BSPp** that produces the minimum value of the expression[13]. This provides a useful mechanism for determining how many processes should be used to solve a given problem, an example of which is illustrated in chapter 6. It should be noted that the expression that is passed to **optBSPp** can only contain constant values, although it may use any of the identifiers and functions listed in table 4.3.

4.8 Allocating Processes to Processors

Opal provides a mechanism for mapping processes to processors through the use of the **on processor** construct. This mechanism is useful for ensuring that the processes of different groups are allocated to processors in a deterministic fashion, rather than letting the implementation randomly pick a processor.

Such a mechanism must be introduced into the language without making algorithms architecture dependent. For this reason Opal views the processors of a system as having identifiers in the range $1..BSPp$, and no assumptions are made regarding the locality of any one processor to any other, i.e. it cannot be assumed that processor i is close to processor $i + 1$.

The **on processor** construct is only allowed in group interfaces and process bodies, an example of which is show below:

```
replicated prefix process body slave (i: 1..10) is
    on processor i;
begin
    supersteps
end slave;
```

4.9 Summary

In the preceding sections we have defined the constructs that form the Opal language. We have taken the ideas of modular languages such as Ada and Modula-3 and applied them to the parallel environment, while at the same time integrating the direct mode features of the BSP model. In fact, Opal takes much of its syntax from the Ada programming language, although a much simplified

[13] **optBSPp** should not be confused with a conventional function. It is a language construct that evaluates its argument for all values of the unbound variable **BSPp**.

subset is used. We shall now summarise the main features of Opal, and relate them to the design rationale of chapter 3.

Shared Memory Inter-Process Communication

In section 4.2 we defined the structure of an Opal program and described how it consists of a number of groups which themselves contain processes. Processes consist of a specification and a body, where the specification is used to declare shared variables and the body describes the supersteps to be executed. Processes communicate with each other via shared variables which follow the BSP memory protocol, but a process may immediately update the shared variables that it owns by using *local dotted components*. Variables that are declared in the body of a process are guaranteed to be local and cannot be accessed by other processes, a rule that is checked by the compiler. These features achieve the goals that were first discussed in section 3.2.2.

Access to the BSP Parameters

The BSP parameters are available to the Opal programmer in the form of predefined identifiers and functions, a requirement that was defined in section 3.2.3. These allow algorithms to be written which can adapt to the architecture on which they are running. An example of this could be a parallel prefix sums group which uses t-ary trees, where at the point of execution the processes decide what the optimum value of t is, based upon the values of g, p and l.

Active Parallel Libraries

Groups provide an active library mechanism for parallel algorithms. They bring together the entry points of many processes into a single unified interface, a requirement that was listed in section 3.3. Other features of the Opal language such as unconstrained arrays, generic groups and packages enhance modularity and code re-use.

Each process must be a member of one group, and only processes in the same group bulk synchronise with each other and may communicate via shared variables. Processes in different groups cannot share variables and run asynchronously with respect to the processes in other groups. This prevents the supersteps of the user's algorithm from interfering with library supersteps, allowing all the library call mechanisms of section 3.3 to be supported.

An Opal program consists of a main group that requires the services of other groups. Each group

will use an exclusive copy of the groups that it requires, unless those groups have been declared as shared. The use of exclusive groups simplifies the performance analysis of Opal programs by reducing the amount of resource contention, as the resources in an exclusive group are only shared amongst the processes of the parent group rather than between the processes of all groups.

Generic groups provide a mechanism for making code more re-usable whilst retaining the static nature of the language. They allow a group to be designed with a variable number of processes and variable sized shared data structures. As with Ada and Modula-3, generic groups allow algorithms to be designed that will operate upon any type, with the type being supplied at instantiation.

Passive Sequential Libraries

While groups provide a mechanism for modularising parallel algorithms, Opal also has the ability to provide standard sequential abstract data types through the use of packages. Packages in Opal are sequential algorithms that have a well defined package interface such as those used in Ada and Modula-3. Such packages allow sequential algorithms to be re-used in parallel algorithms, e.g. a sequential sort package.

Preventing Nested Supersteps

The Opal constructs ensure that supersteps can never be nested, a feature that was discussed in section 3.3.1. While nested supersteps may introduce unwanted complications, it should still be possible to declare procedures that perform supersteps. This is safely accomplished in Opal using the concept of a *superstep procedure*.

Static Number of Processes

Opal is a static language in that a fixed number of processes are created during the initial startup phase after which no new processes are initiated. As discussed in section 3.5, this static property aids in the performance prediction of Opal algorithms, as issues such as process startup costs and parallel slackness do not need to be considered.

While static numbers of processes increase the performance prediction properties of algorithms, they decrease the possibility of employing code re-use. Generic groups can be used to address this problem, providing a mechanism where the number of processes is fixed at runtime, but may be instantiated to different values depending on the problem being solved, or the values of the BSP

parameters.

Static Source Checking

Opal is a strongly typed modular language, where not only local assignments but also remote assignments and rendezvous are checked for type correctness. Due to the nature of *dotted notation* it is easy for a compiler to check that a process cannot access the local variables of other processes. These two features accomplish the goals of the design rationale that were set out in section 3.7.

Chapter 5

Performance Prediction in Opal

5.1 Introduction

In this chapter we shall discuss the effects that groups and rendezvous have on the performance prediction of Opal programs. An insight is given into the complications that arise from incorporating a per message synchronisation into a BSP style language. Issues such as deadlock, message queueing and message servicing are discussed, and it is shown how Opal restricts these problems, thus allowing an accurate cost model to be produced.

We then examine how the complexity of an group should be stated, assuming that the source of an Opal library group is not available to the programmer, and must be treated as a black box in terms of complexity analysis.

We examine how Opal programs can be mapped to a set of processors, and introduce the concept of *parallel overlap*. We state that if Opal groups are used as the library mechanism then parallel overlap is likely to be employed, leading to a predictable algorithm. However, the group system of Opal may be abused, producing algorithms that are very hard to predict, e.g. producer-consumer pipelines. While such applications could be developed using the Opal group system this is not their intended purpose, and as such we shall not attempt to analyse their performance.

The concept of superstep collapsing is introduced, a method which is used to obtain a *global view* of the BSP machine. This allows us to combine the complexities of the subgroups of a program using methods which are compatible with the BSP model.

The library calling mechanisms defined in 3.3 are examined, and it is described how these map

to the processors of a real architecture. We show how all-to-all and one-to-all library calls are relatively easy to predict, while any-to-any calls could lead to cumbersome equations involving the maximum of many variables.

In the final sections of this chapter we will give an example of the performance prediction of an Opal program containing three groups, and show how the hierarchical nature of rendezvous aids in such analysis. This example is then expanded to include the performance prediction of Opal programs that contain multiple groups at many levels.

5.2 Problems Introduced by Message Synchronisation

Message passing languages and libraries such as OCCAM-3 [Barrett, 1992] and PVM [Geist *et al.*, 1993] are difficult to analyse because of the per message synchronisation that occurs. These complications are due to the effects of deadlock, message queueing and message servicing, each of which we shall describe in greater detail in the following sections.

Deadlock

A process will deadlock if it waits for an event that will never occur. This is a frequent problem with message passing languages, since it is easy for the programmer to get send and receive pairs mismatched. However, it is also possible for deadlock to occur in shared memory systems, e.g. a process waits for a shared variable to be set to a specific value, but no other process ever writes to this location therefore producing deadlock.

Message Queueing

If a number of processes all send a message along the same channel then message queueing will occur. In this case the performance of the channel becomes serialised and is very difficult to predict, unless precise information is available regarding how many processes are contending for the same resource.

Message Servicing

Message servicing is a situation similar to that of message queueing, where there is a finite time between a process accepting one message, and then being available to accept another. This delay

is caused by the receiving process performing computation before it is able to accept another communication. Such computation could be the result of a request to perform work, or could be due to housekeeping tasks that the receiving process has to perform between messages.

5.3 Opal Constructs that Aid in Performance Prediction

In the previous section it was shown how message passing mechanisms introduce a number of problems with performance prediction. Since the Opal rendezvous is effectively a message passing scheme we must consider how the language constructs help us in analysing an algorithm's performance.

Deadlock in Opal

Deadlock can occur in an Opal program for one of two reasons: either the group that we are rendezvousing with has terminated, or we are using an entry point that no process is willing to accept.

If we are using exclusive groups then the programmer can make use of the **terminate** alternative in a **select** statement to ensure that a process only terminates when its parent group terminates, guaranteeing that deadlock due to terminated groups cannot occur. However, it is possible to use a **select** statement without a **terminate** alternative, in which case the programmer should take great care to ensure that deadlock is not possible. Such care must also be taken with shared memory systems, as a badly designed algorithm can still cause deadlock in this environment.

It is the responsibility of the programmer to ensure that there is a process willing to accept an entry point. Since **accept** statements are at the very core of an Opal library it should be relatively easy for the programmer to check that all entry points have an accepting process.

Message Queueing in Opal

In section 4.2.5 we described how each group receives a separately executing version of any subgroups that it requires. We therefore know that only the processes in our own group will be competing for the resources of a subgroup. Since this contention will be at the superstep level, it is relatively easy to determine if two processes in the same group will be subject to queueing on a rendezvous call. Without the use of exclusive groups it would be very difficult to predict which processes are contending for a rendezvous, since the processes could be in different groups.

In the rest of this chapter we shall assume that there is no message queueing in Opal, i.e. exclusive groups are employed and no two processes are competing for the same rendezvous.

Message Servicing in Opal

The restrictions on the **accept** statement that were listed in section 4.5.5 ensure that the rendezvous servicing period is kept to a minimum. From these restrictions we know that an accepting process can only perform an amount of computation C_{pre} before it is ready to accept a rendezvous. Once this computation has been performed the process will accept a rendezvous, compute the results, and then return to accept another rendezvous.

5.4 Specifying the Performance of a Group

In this section we demonstrate how to describe the complexity of an Opal group. Groups are the library mechanism of Opal, and it is quite likely that an Opal program will contain groups that have been written by other programmers.

Opal performance prediction must be compositional, i.e. we must be able to take the complexity of a group which we know very little about, and combine it with a group we have written ourselves. This is essential if Opal groups are to be used as a mechanism for libraries since we must assume that we do not have access to the implementation of a library group.

Due to the restricted nature of **accept** statements we can see that a subgroup can only perform an amount of local computation C_{pre} before it encounters its first **accept** statement. The language constructs also guarantee that there are no supersteps before the first **accept** statement, and so no remote communication or synchronisation can take place.

We also know that after a subgroup has finished performing a rendezvous it either terminates, or returns to the original **accept** statement. Assuming that the group is using a **select** statement with a **terminate** alternative, it will perform an amount of computation, C_{post}, and then return back to accepting a rendezvous, only terminating when its parent does.

For these reasons the performance of a rendezvous call can be specified using an equation of the following form:

$$C_{pre} + C_{post} + C + H.g + n.l \tag{5.1}$$

This represents a subgroup that performs an amount of computation C_{pre} before it is ready to accept a rendezvous. During the rendezvous the group processes perform an amount of local computation C, and communicate a total of H words across all of their n supersteps. Once the rendezvous is complete an amount of computation C_{post} is performed before the group is ready to accept another rendezvous. We can further simplify this equation by merging the costs C_{pre} and C_{post} into C, and replacing $n.l$ with L, producing the following:

$$C + H.g + L \tag{5.2}$$

We shall therefore assume that when a programmer develops a group they will annotate the group interface with an equation of the above form. This will allow the user of such a group to analyse the performance of their algorithm, without requiring any knowledge of the implementation of the library group.

5.5 Parallel Overlap

In previous chapters we have discussed how an Opal program consists of a number of groups that form a tree, with the main group being at the root. Due to the restricted structure of the processes in the subgroups we know that upon execution they perform some local computation and then become dormant waiting for a rendezvous call[1]. We also know that once a process calls a rendezvous it cannot execute any further supersteps until the rendezvous is complete, effectively forcing the calling process to become dormant at the end of the superstep[2]. For these reasons it is reasonable to expect that the main group, and all of the subgroups, will share a set of processors, since only certain sets of processes will be executing on a processor at any point in time. We call this sharing of processors *parallel overlap*.

For example, consider a main group that requires the services of two subgroups. The structure of such a program, and the allocation of processes to processors, is shown in figure 5.1.

In this example there are four processors, with each process of the main group running on a separate processor. However, the subgroups only use two processes each as they are being called in an any-to-any type mechanism. Since these subgroups are only active when they are rendezvoused with, they share the same processors as the main group. Such a mapping of processes to processors can be performed with the **on processor** statement, as described in section 4.8.

[1]This is due to the restricted nature of **accept** statements, as defined in section 4.5.5.

[2]This causes all other processes in the group to become dormant due to barrier synchronisation.

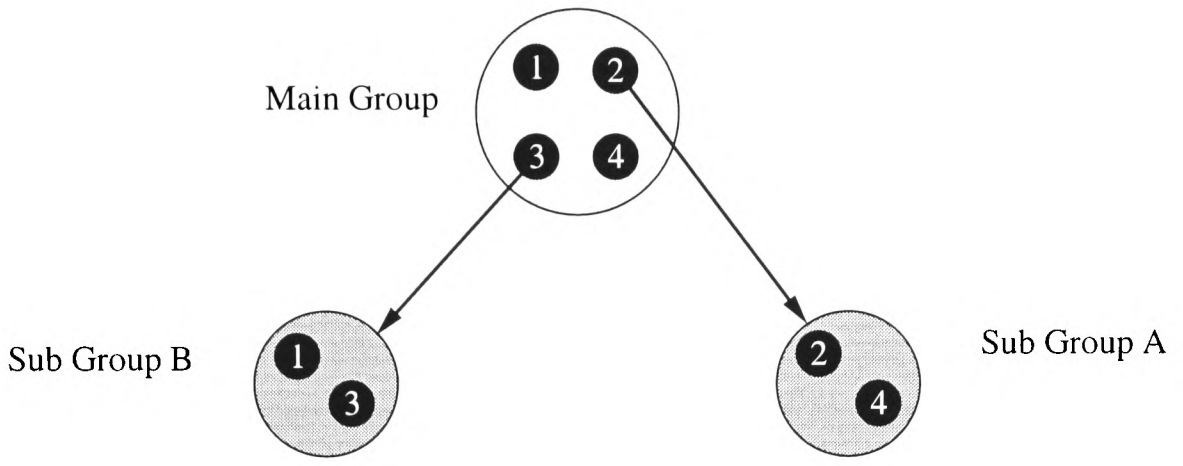


Figure 5.1: Example Opal Group Execution

We shall now formalise the notion of *parallel overlap*. Let us define the function $processors(G)$ as the function that returns the set of processors that group G executes on, where each processor is a number in the range $1..BSPp$. If there is a parent group, P, and n subgroups, $S_1..S_n$, then we say that a program employs parallel overlap if at every point in the group tree:

$$processors\ (P) = \bigcup_{i=1}^{n} processors\ (S_i) \tag{5.3}$$

$$\bigcap_{i=1}^{n} processors\ (S_i) = \{\} \tag{5.4}$$

Throughout the rest of this chapter we shall assume that Opal programs are employing *parallel overlap* as defined by the above equations. This is reasonable since parallel libraries will not perform any work until requested to do so, and therefore they should share the processors with the parent group. We exclude the cases where Opal groups are being used in a way that does not conform with a library scheme, e.g. if a programmer uses two groups, where one group takes half of the processors and performs I/O, and the other group takes the remaining processors and acts as a consumer. In these cases parallel overlap is not being employed. While such situations are possible they do not conform with the intended use of Opal groups, and so we shall exclude them from our analysis.

5.6 Superstep Collapsing

An important point to note about the rendezvous mechanism of Opal is that it does not cross superstep boundaries, i.e. we know that at the end of a superstep any rendezvous that were initiated

are completed. For this reason the complexity analysis of an Opal program is still compositional in units of supersteps, i.e. for a given group of processes we can compute the complexity of each superstep, and then sum these complexities to obtain the performance of the whole group. This is an important feature of the BSP model that has been maintained in the implementation of Opal groups. In the remaining sections of this chapter we shall only consider the complexities of single supersteps.

We utilise this hierarchy of supersteps by using the concept of *superstep collapsing*. This is when we collapse the multiple supersteps of the subgroups into a single superstep of the parent group. For example, if a subgroup has complexity $C + H.g + L$ then in terms of the parent group we consider this as a single superstep that performs an H-relation, where the synchronisation cost L is simply added into the cost of the computation C.

This method of superstep collapsing also applies to regular BSP performance prediction. For example, if a program executes n supersteps where in superstep i we perform an amount of computation c_i and implement an h_i-relation then standard BSP analysis states that the cost of the program, $T_{standard}$, will be:

$$T_{standard} = \sum_{i=1}^{n}(c_i + h_i.g + l) \tag{5.5}$$

However, we could consider the program as a single superstep that performs an amount of computation C, implements an H-relation, and costs L in synchronisation. We can see that $C = \sum_{i=1}^{n} c_i$, $H = \sum_{i=1}^{n} h_i$ and $L = \sum_{i=1}^{n} l$. Using a collapsed superstep we could predict the cost of the program, $T_{collapsed}$, to be:

$$T_{collapsed} = C + H.g + L \tag{5.6}$$

It is easy to verify that in this case $T_{standard} = T_{collapsed}$.

5.7 Library Calling Mechanisms

In section 3.3 the concept of an all-to-all, one-to-all and any-to-any library call was introduced. In this section we shall show how these different call mechanisms relate to the tree of groups that is created when an Opal program is executed.

In figure 5.2 we show how the different types of mechanisms create a group tree when one parent group uses the services of one of more subgroups. With an all-to-all mapping there can only be one subgroup, and each process in the parent group will rendezvous with an appropriate process in the subgroup. A one-to-all mechanism leads to the same group tree, except this time only one process in the parent group will rendezvous with a single process in the subgroup.

An any-to-any type library call will produce a number of different groups that are at the same level in the group tree. In this case the processes in the main group will be split into sets, where each set rendezvous with a different subgroup process. Since we are only considering algorithms that employ parallel overlap, we know that the processors in the subgroups will be the same as those used in the parent group.

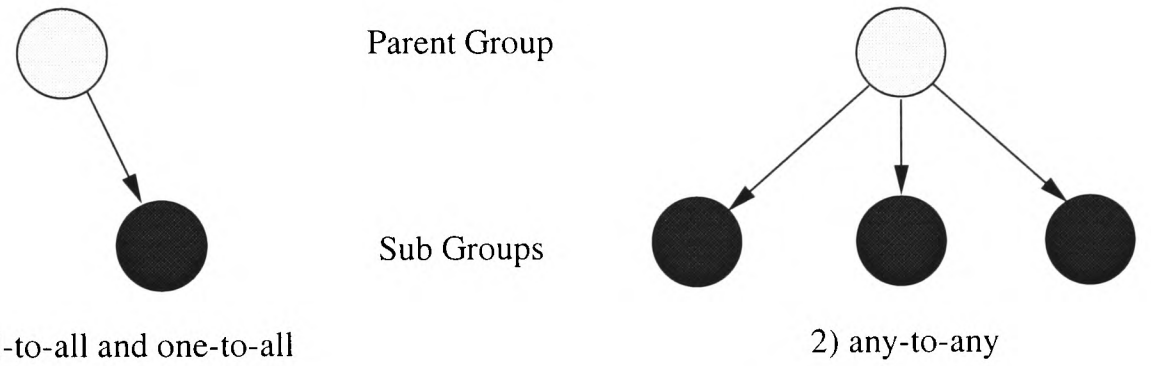


Figure 5.2: Calling Mechanism mapping to Group Tree

In terms of an Opal program the one-to-all mechanism is produced by having one process perform a superstep that calls a single rendezvous. The all-to-all mechanism would utilise entry arrays, where each process rendezvous with the same subgroup in the same superstep. The any-to-any mechanism can be produced in one of two ways; either the processes of the parent group each call rendezvous from different groups, or a single process calls multiple rendezvous in the same superstep. To aid in further discussion we shall define the *group depth* to be the depth of the group tree.

We must also consider how the processes of an Opal program are allocated to processors, and how this affects performance prediction. It should be noted that we are taking a global view of the BSP machine, and as such we collapse the supersteps of the subgroups into a single superstep of the parent group. We assume that the value of l for a given architecture is the worst case value for barrier synchronising any one subset. It should be noted that we do not need to consider bulk synchronising multiple subsets since the processes in each group, and therefore each subset, run asynchronously with respect to each other.

Figure 5.3 illustrates how we combine the h-relations of the different processes, depending on how

the processors are shared. We shall consider two situations; either the processes of the subgroups use the same set of processors as the parent group (parallel overlap), or they run on completely different processors.

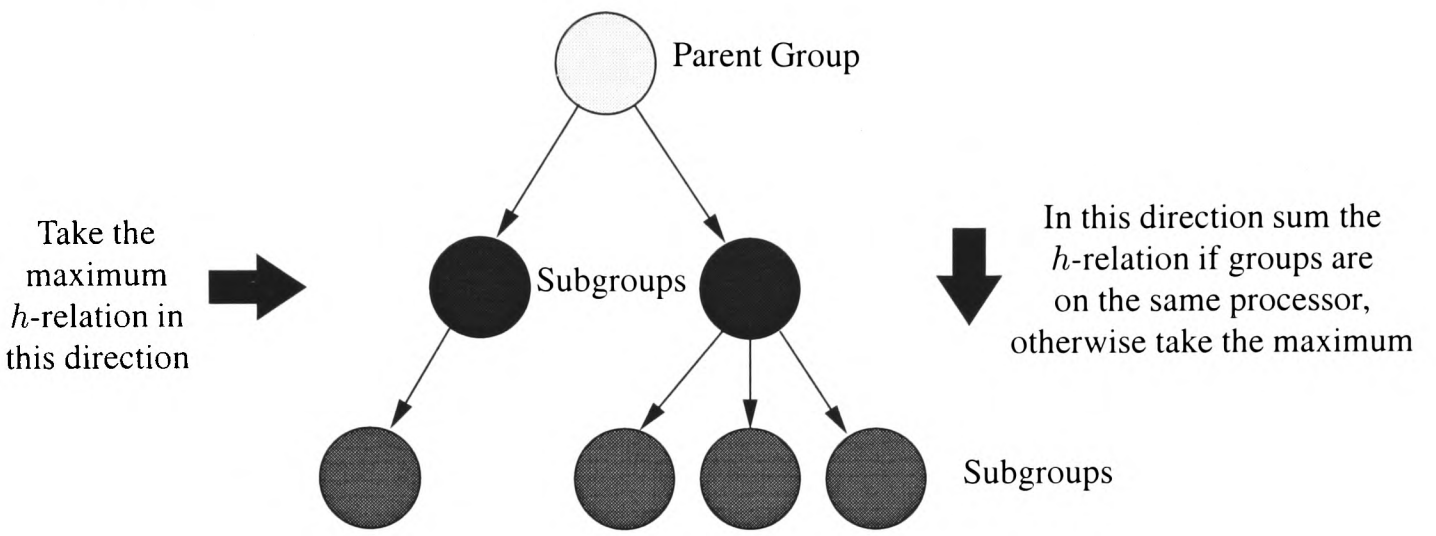


Figure 5.3: Opal Analysis

In this figure we show how in the horizontal direction of the group tree we should take the maximum of the h-relations of the subgroups. Here we are assuming that the processes in the subgroups never share processors, a valid assumption since we are only analysing programs that employ parallel overlap. The reason we take the maximum is because we are collapsing the supersteps of the subgroups into a single superstep, and then taking the maximum number of words transferred by any processor. This is the first step of the performance prediction, and effectively merges the complexity of the subgroups at the same level.

In the vertical direction we must sum the h-relations of the parent and subgroups if the processes are on the same processor, otherwise we take the maximum. The reasoning is that if they are sharing processors then we must take the total number of words being transferred from a processor, but if they are on different processors we take the maximum since the h-relations will be implemented in parallel. Since we are only considering algorithms that employ parallel overlap we shall always take the sum in the vertical direction.

If we compare figures 5.2 and 5.3 we can see that performance prediction using an all-to-all or one-to-all is much easier than any-to-any. This is because with all-to-all and one-to-all there is only one subgroup, and so the horizontal maximum is effectively removed. We can also remove the vertical maximum if we employ parallel overlap.

5.8 Performance Prediction: Group Depth 2

In this section we shall consider the performance prediction of an Opal program that consists of three groups; a main group and two subgroups, where the subgroups are called using an any-to-any mechanism as described in section 3.3. We shall assume that the programmers of the two subgroups have supplied a complexity analysis in the form of $C + H.g + L$, as in the following table:

Main group:	Implements an h_k relation, local computation c.
Subgroup 1:	Complexity $C_1 + H_1 + L_1$.
Subgroup 2:	Complexity $C_2 + H_2 + L_2$.

The first step we take is to combine the subgroups in the horizontal direction, obtaining a complexity of $\max(C_1, C_2) + \max(H_1, H_2).g + \max(L_1, L_2)$. We must now combine this with the h_k relation which is implemented in the parent superstep and allow for the r words that are transferred due to the parameters of the rendezvous. Since we are sharing processors we sum the h-relations, obtaining a total complexity, T, for the superstep:

$$T = c_k + \max(C_1, C_2) + [h_k + r + \max(H_1, H_2)].g + l + \max(L_1, L_2) \tag{5.7}$$

If there were more than two subgroups then we would expand the max functions to include the complexity of each subgroup, but the form will still be the same as equation 5.7.

We can now apply the same analysis but considering an all-to-all or one-to-all library call. In such a case there is only one subgroup and so we do not need to take the maximum in the horizontal direction. If the subgroup has complexity $C + H.g + L$ then equation 5.7 is simplified to:

$$T = c_k + C + (h_k + r + H).g + l + L \tag{5.8}$$

If the Opal group mechanism is truly being used to implement parallel libraries then we can also assume that no other work is being performed in the parents superstep, i.e. $c_k = h_k = 0$. This then gives us a complexity of:

$$T = C + (r + H).g + l + L \tag{5.9}$$

Assumptions

1. We assume that the synchronisations of the subgroups does not interfere with the synchronisation in the parent group, and vice-versa.

2. We have taken a global view of the BSP machine and not split it into a number of separately executing sub BSP machines.

3. The different subgroups do not interfere with each other while executing.

4. We do not allow for any costs associated with context switching.

Justification of Assumptions

1. It is likely that the subgroup will perform many synchronisations in order to perform the required task. Conversely, the parent group will only perform one synchronisation at the end of its superstep and then it will wait for the rendezvous to complete. Therefore any effect that the subgroup has on the synchronisation of the parent group will be minimal, since the subgroup's synchronisation costs will dominate.

2. It is likely that the parent group will become dormant very soon after issuing a rendezvous. This is because the parameters in the entry call can only be local variables and as soon as these are computed the user of the group is likely to call the rendezvous. Any statements that follow the rendezvous call are likely to rely on the results of the rendezvous and so must be executed in the next superstep. Therefore, in a typical situation the calling superstep would perform a small amount of computation, issue the rendezvous request, and then reach the end of its superstep.

3. We have employed superstep collapsing to view the multiple supersteps of the subgroups as a single superstep. We then take a global view of the BSP machine just as if it were a program that does not contain groups and so we take the maximum number of words h that any processor transfers. It could be argued that we should view the program as a set of separate BSP machines, and take the cost of the subgroups as $max(C_1 + H_1.g + L_1, C_2 + H_2.g + L_2)$, therefore giving $T = c_k + (h_k + r).g + l + max(C_1 + H_1.g + L_1, C_2 + H_2.g + L_2)$. Such analysis could be performed, and would be at most a factor of three better than equation 5.7. However, we take the view that we are considering the *global* properties of the machine and as such do not split it into a number of sub BSP machines with different values for g and l.

4. We assume that the context switching cost is small compared to the cost of the service the group is performing. Again this is reasonable since we expect the calling process to become dormant soon after issuing the request, therefore resulting in no further context switches.

5. If we use a buffered mechanism for h-relations then the calling process will not even attempt to communicate with its peers, or synchronise, until the rendezvous is complete.

While it would be possible to produce algorithms that do not conform to these assumptions, the vast majority of libraries produced will behave in the fashion described.

5.9 Performance Prediction: Opal Programs

In the previous example we considered a main group which had a group depth of two. In large parallel applications it is likely that many groups will be employed, forming a hierarchy of rendezvous calls for a single superstep of the parent group, as shown in figure 5.4.

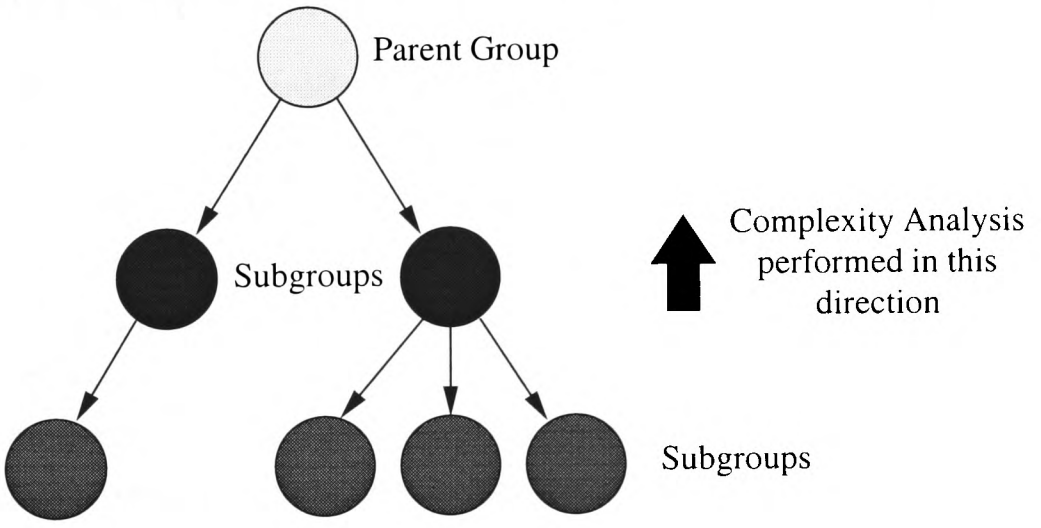


Figure 5.4: Complexity Analysis of Opal Programs

In order to predict the performance of this program we must first start with the groups at the leaves of the tree that do not perform any rendezvous. Using the standard BSP performance prediction methods we arrive at an equation that represents the complexity of such groups. We then use these results to predict the next layer up in the tree, obtaining a cost in the form of equation 5.7. We continue this traversal up the tree until we reach the root, at which point we have an equation representing the cost of the main groups superstep.

This method allows us to compute the cost of the parent group given the cost of the subgroups that it requires. For this reason the groups at the leaves of figure 5.4 may not be the real leaves

of the tree and may be library groups that the programmer has imported. This does not affect the performance prediction, since as long as we are given a complexity for the performance of these groups we can calculate the performance of the parent group.

5.10 Summary

The issues of deadlock, message queueing and message synchronisation were discussed in the context of message passing languages. It was demonstrated how the restrictions of the **accept** statement simplify these problems in the Opal language, leading to a rendezvous mechanism that is predictable.

It was noted that some of the groups in an Opal program will be written by other programmers, and that we must be able to analyse the performance of such programs without requiring access to the implementation of the library groups. This is possible if the programmers of the library groups provide equations representing the complexity of a call to each of the entry points. Such equations would be of the form $C + H.g + L$.

We introduced the concept of *parallel overlap* and limited our performance prediction to algorithms which employed such a method. This limitation is valid for programs that use the group mechanism to implement parallel libraries where sets of processors will be idle, waiting for a request to perform a service. However, it does not allow for algorithms that use the group mechanism in order to accomplish tasks such as pipelining, e.g. a producer-consumer pipeline where the two groups are executing on different processes and are intended to be active the majority of the time. Such programs are using the Opal library mechanism in a way that is not intended and are an area of future work for performance prediction.

Such assumptions of *well behaved* algorithms are common in other languages such as Split-C and Modula-3. For example, in Modula-3 the compiler can guarantee that all operations are type safe. However, the programmer may circumvent the restrictions of the language in a way that prevents the compiler from making such a guarantee. If this occurs then the programmer must add the **unsafe** keyword to their module declarations, indicating that the program is using the Modula-3 language in a way that was not intended.

Shared groups are analogous to the **unsafe** features of Modula-3. If a program utilises a shared group then no prediction can be made regarding its performance. However, we can still predict the portions of the program that do not interact with the shared group, e.g. if a program uses an I/O group for communication with a file system then we cannot predict the supersteps that read

and write files, but we can predict all other supersteps.

The concept of collapsing supersteps was introduced, where we view the multiple supersteps of the subgroups as a single superstep. We use this to take a global view of the architecture, rather than splitting it up into a number of separate BSP machines.

We then demonstrated how the performance of a group can be determined as long as we have a complexity analysis for any subgroups that it requires. This gives us a compositional method for determining the complexity of an Opal program, where we start at the leaves and calculate the complexity at each level until we reach the root.

A number of assumptions have been made regarding the behaviour of the subgroups. These assumptions are justified if the group mechanism really is being used as a method for utilising parallel libraries. In this environment it is likely that the calling process will use a complete superstep just for the purpose of issuing a rendezvous request. In this situation the calling process will become dormant very quickly, releasing any resources that are required by the subgroups, e.g. processor cycles and network bandwidth. These assumptions are further validated by the restrictive nature in which **accept** statements can be used, and the fact that the communication, and synchronisation, of the parent processes is likely to be buffered until after the rendezvous has completed.

In the following chapter we shall give examples of real Opal programs that use the group mechanism to implement libraries of code, and it will be seen how such implementations are predictable.

Chapter 6

Opal Algorithms

6.1 Introduction

In this chapter we shall present Opal implementations for the string edit, reduction, prefix sums, bucketing, iterative sorting and convex hull problems. Such examples will illustrate the constructs of the language that were defined in chapter 4, while also allowing verification of the performance prediction methods of chapter 5.

These examples have been chosen to demonstrate the parallel library mechanism of Opal. For example, we shall show how we can re-use the prefix sums group in order to implement a bucketing group, and then use this bucketing group to implement a parallel sort and convex hull algorithm.

The language design rationale of chapter 3 dictates that Opal should be architecture independent, allowing the same software base to be developed for a variety of parallel machines. In this chapter we demonstrate how algorithms can use the BSP parameters to automatically adapt to the architecture they are executing on. For example, we show how a prefix sums group can use the **optBSPp** function to automatically select the number of processes to use for a given problem size.

6.2 String Edit Distance

In this section we shall present an Opal implementation for the string edit problem that was described in section 2.3.1. Further discussions on this algorithm and its implementation can be found in [McColl, 1993b].

In the algorithm of section 2.3.1 a process was required for each character of the X string hence requiring $lenX$ processes in total. We implement a *coarse grained* version of the algorithm, where each of the p processes computes a vertical slice of the $DIST$ array of size $\left\lceil \frac{lenX}{p} \right\rceil$.

The implementation of the algorithm uses a single group called $main$ and a replicated process called $worker$. This requires the use of three files; the group interface main.grp, the $worker$ specification worker.spl and the $worker$ body worker.opl. In such a simple algorithm the group interface contains only two constant definitions as follows:

```
group main is
    maxlenX: constant := 100000; maxlenY: constant := 2048;
end main;
```

We must now provide a specification of the replicated $worker$ process that will compute the elements of the $DIST$ array as shown in figures 2.2 and 6.1.

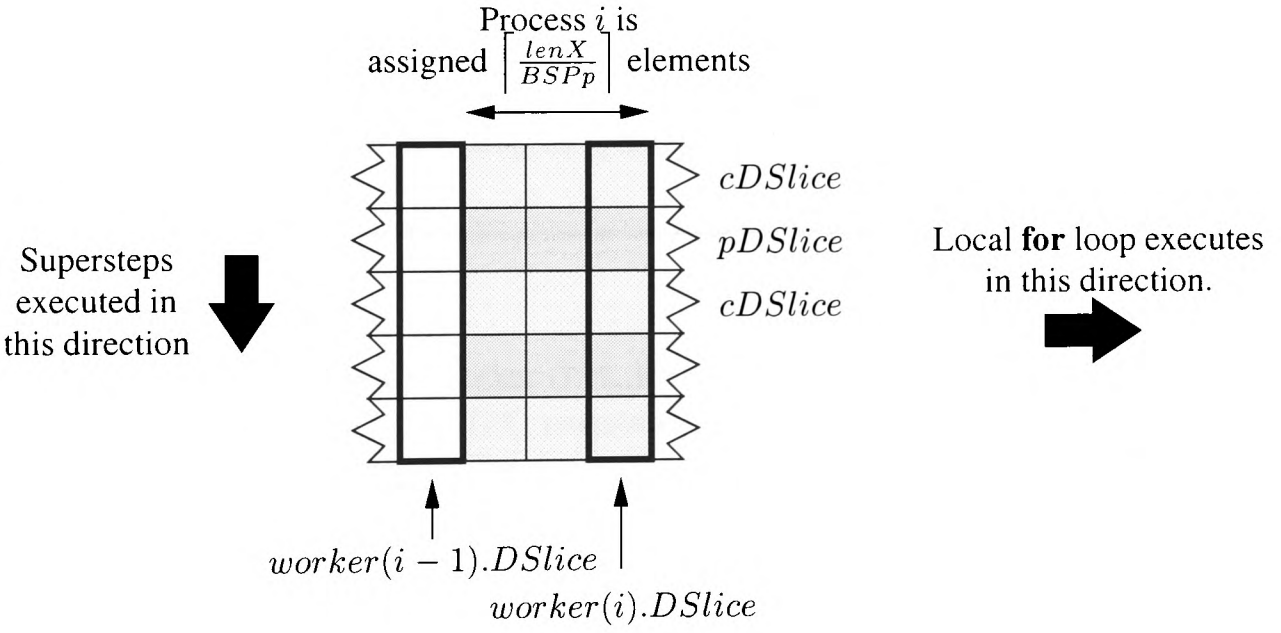


Figure 6.1: Coarse Grained Parallel Edit Distance

As we need to communicate the boundary values of the $DIST$ array with neighbouring processes, we require that this variable is held in shared memory. In order to improve efficiency we ensure that the slice of the $DIST$ array that a process is assigned to is held in the shared memory of that process. The specification of such a process, in the file worker.spl, is shown below:

```
with main;
replicated main process worker (i: 1..BSPp) is
    DSlice: array (0..main.maxlenY) of integer;
end worker;
```

This declares $BSPp$ processes that each contain an array $DSlice$ that will be used to hold the values of a vertical slice of the $DIST$ array. It should be noted that the worker process only declares a single vertical slice of the $DIST$ array in its shared memory, while in the body it will be working on $\left\lceil \frac{lenX}{BSPp} \right\rceil$ such slices.

Figure 6.1 shows a how a single process computes a slice of the $DIST$ array. The algorithm consists of a loop containing two supersteps. In the first superstep the process fetches the neighbouring values of the $DSlice$ array and in the second superstep it computes a horizontal segment of its $DIST$ slice. The last element of this horizontal segment is then assigned to the appropriate element of $DSlice$, ready for use by the next processor. To calculate the current horizontal slice only requires values from the previous horizontal slice, and so in each superstep we alternate between using two arrays $cDSlice$ and $pDSlice$.

The source code for such a $worker$ process is shown below:

```
with main, worker;
replicated main process body worker (i: 1..BSPp) is
    diagIDX: constant := 1; leftIDX: constant := 2;
    lneighbr: array (1..2) of integer;
    up, left, diag, nval, y, lenX, lenY, XBlockSize: integer;
    myY: array (1..main.maxlenY) of character;
    myX: array (0..main.maxlenX / BSPp) of character;
    pDslice, cDSlice: array (0..main.maxlenX / BSPp) of integer;
    oncur: boolean;
begin
    begin "initiliase" superstep
        -- Superstep to read the Y array and initialise variables
    end superstep;

    for j in 2 .. lenY + BSPp loop
        begin "Get surrounding vals #" & string(j) superstep
            y := j - i;
            if y > 0 and y <= lenY then
```

```
            -- Fetch the surrounding elements up, left and diagonal
            if i = 1 then
                lneighbr (leftIDX) := y; lneighbr (diagIDX) := y - 1;
            else
                lneighbr := worker (i - 1).DSlice (y - 1..y);
            end if;
            if oncur then up := pDslice (0); else up := cDslice (0); end if;
        end if;
    end superstep;

    begin "Update current val #" & string(j) superstep
        if y > 0 and y <= lenY then

            -- Compute the value for DSlice(0)
            nval := min (up, lneighbr (leftIDX)) + 1;
            if myX (0) = myY (y) then nval := lneighbr (diagIDX); end if;
            if oncur then cDslice (0) := nval; else pDslice (0) := nval; end if;

            -- Compute the value for the rest of the DSlice block
            for j in 1..XBlockSize - 1 loop

                if oncur then
                    left := cDSlice (j - 1); diag := pDslice (j - 1); up := pDslice (j);
                else
                    left := pDslice (j - 1); diag := cDSlice (j - 1); up := cDSlice (j);
                end if;

                nval := min (up, left) + 1;
                if myX (j) = myY (y) then nval := diag; end if;
                if oncur then cDSlice (j) := nval; else pDslice (j) := nval; end if;

            end loop;

            -- Update our shared memory with the last point we calculated
            #worker.DSlice (y) := nval;
            oncur := not oncur;
        end if;
    end superstep;

end loop;

end worker;
```

In this example we have demonstrated a number of features of the Opal language including variable length arrays, remote array slicing and local dotted component access. These features have been used to increase the efficiency of the algorithm in terms of space requirements, the number of remote communications and access to shared variables held in local memory.

6.2.1 Performance Prediction

We shall now consider the performance of the string edit algorithm. In order to keep our presentation brief we only concentrate on the main loop of the algorithm as this is where the most interesting communication is.

If we let n be the length of string Y then we can see that the **for** loop is executed $n + BSPp - 1$ times, with each iteration containing two supersteps. The first superstep may request a two element integer array from its left neighbour and may also be requested to send a two element array to its right neighbour. Let us assume that these requests are always performed, therefore leading to a 4-relation if $BSPp > 2$. If $BSPp = 2$ then a 2-relation is implemented and for $BSPp = 1$ no communication is required, giving a 0-relation.

The second superstep performs no communication but instead calculates $XBlockSize$ elements of the $cDSlice$ or $pDSlice$ array. This is a total computation of $\left\lceil \frac{m}{BSPp} \right\rceil$ c, where m is the length of string X and c is the computation time required to execute one iteration of the **for** loop. We can now see that the total time, T, for the algorithm is:

$$T = (n + BSPp - 1).\left(\left\lceil \frac{m}{BSPp} \right\rceil .c + 2.min(BSPp - 1, 2).g + 2l\right) \tag{6.1}$$

Table 6.1 shows the values of c on two parallel architectures; a Silicon Graphics Power Challenge (SGI) and an IBM SP2 [Agerwala *et al.*, 1995]. These were obtained by timing sequential code that performs part of the string edit algorithm[1].

Architecture	c
SGI	3.384170e-07
SP2	3.928815e-07

Table 6.1: c For The SGI and SP2 Architectures

Using these values and those of $N_{\frac{1}{2}}$ and g_∞ from tables D.1 and D.2, we can predict the perfor-

[1]The manufacturers processor specifications could also be used to calculate c, but this may lead to less accurate results.

mance of the string edit algorithm for $m = 10000$ and $n = 1000$. Figures 6.2 and 6.3 show the results of such predictions on the IBM SP2 and Silicon Graphics Power Challenge (SGI) architectures.

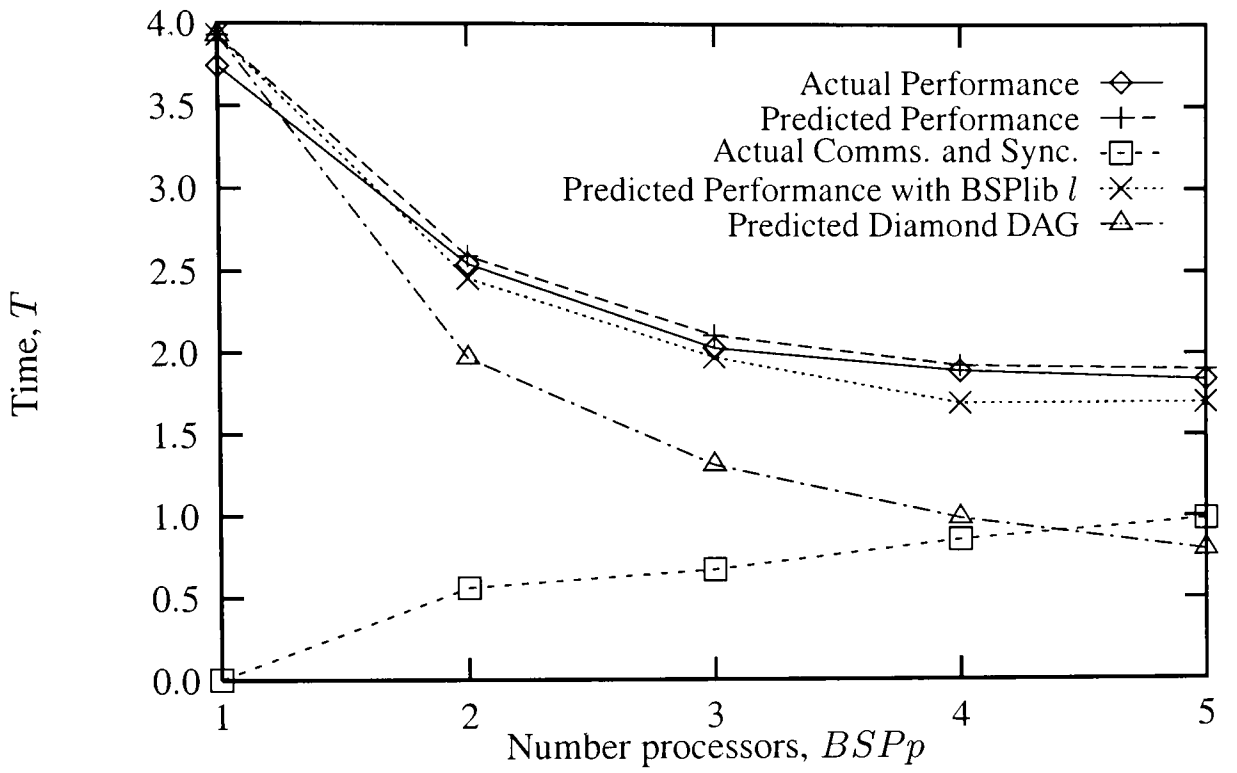


Figure 6.2: SP2 String Edit Performance.

It should be noted that we have used the most accurate value of g based on equation 2.1, i.e. $N_{\frac{1}{2}}$ and g_∞ were used to obtain a value for $g(x)$, where x is the x-relation being implemented. While this does lead to a slightly more complicated calculation, it does give the most accurate results.

It can be seen that the predicated and real performance of the algorithm are very close, with the error being in the order of 0.08 seconds (5%) for the SGI, and 0.07 seconds (4%) for the SP2. It is always the case that the predicted performance is worse than the real performance, possibly due to the fact that a structured h-relation is being implemented by the $worker$ processes, rather than a random one that was used to obtain the values in tables D.1 and D.2.

Figures 6.2 and 6.3 also show the actual amount of communication and synchronisation that was performed by the algorithm. The problem size was selected such that when the maximum number of processors was used the ratio of computation to communication and synchronisation was approximately 1:1. This is important since the usefulness of the BSP cost model is in the prediction of the communications performance and so we must ensure that the algorithm contains a good mixture of local computation and remote communication.

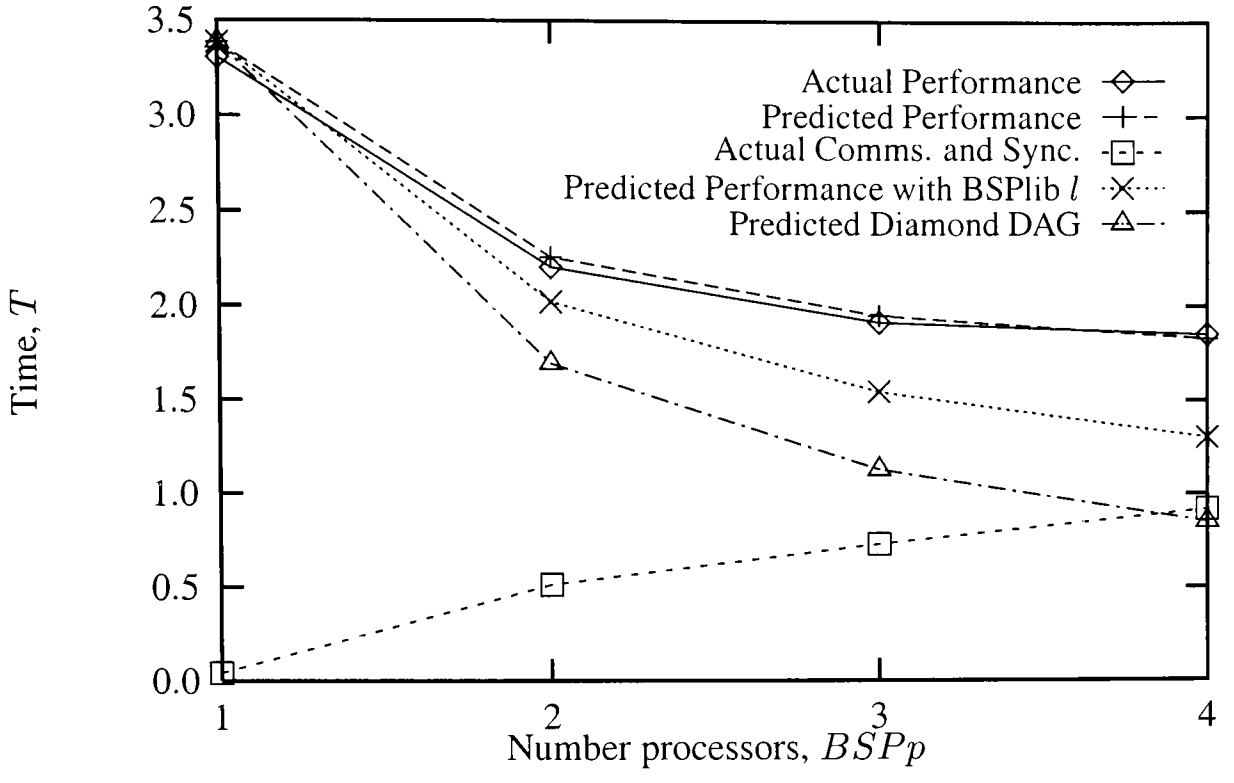


Figure 6.3: SGI String Edit Performance.

6.2.2 Parallel Speedup

As can be seen from the *actual performance* of figures 6.2 and 6.3 there is only a very modest parallel speedup as the number of processors is increased. By analysing equation 6.1 we can see that for a four processor string edit on the SGI approximately 0.58s is spent synchronising. This is over 25% of the total runtime. Similarly, for a five processor string edit on the SP2 approximately 0.62s is spent synchronising, again over 25% of the total runtime.

If we compare the Opal parameters of tables D.1 and D.2 to those of BSPlib [Skillicorn *et al.*, 1996] then we can see an order of magnitude difference[2]. For example, the value of l using BSPlib on the SGI is approximately ten times better than the value of l for Opal, even though they are running on exactly the same machines. If the Opal runtime system were as optimised as that of BSPlib then we would obtain a much improved parallel speedup, as is shown by the graph labelled *Predicted Performance with BSPlib l* in figures 6.2 and 6.3.

We are also using a non-optimal algorithm for string edit. If we were to use the diamond DAG scheduling technique [McColl, 1996a, McColl, 1996b] then the cost, T, of the string edit algorithm becomes:

[2]The Opal runtime system is experimental, and as such is not highly optimised.

$$T = \frac{nmc}{p} + (n+m)g + pl \tag{6.2}$$

Using equation 6.2 and the values of g_∞ and l from tables D.1 and D.2 we can predict the performance of the diamond DAG algorithm in Opal. Such a prediction is shown by the graph labelled *Predicted Diamond DAG* in figures 6.2 and 6.3.

From this analysis we can see that if we had a production Opal compiler, or used the diamond DAG algorithm, then a much improved parallel speedup could be obtained.

6.2.3 Automatic Architecture Adaption

We can use this example to demonstrate another feature of the BSP model and how it can be applied using Opal. In equation 6.1 the total time for the algorithm consists partly of communication and synchronisation, and partly of local computation. It will be the case that for a given problem size there will be a maximum number of processors to use, after which point using more processors will decrease the computation, but increase the communication by a disproportionate amount.

For example, consider the case of $m = 3500$ and $n = 500$ for the IBM SP2. Figure 6.4 shows the real and predicted results for such a situation, demonstrating that the optimum number of processors to use is three. There is a reasonable gap between the predicted and actual results due to the value of c we are using, i.e. c was measured for relatively large **for** loops, but in this case the loop is small causing cache effects to become more obvious.

If $lenX$ was known at the start of the program then we could replace the replicated range of the $worker$ process with:

```
i: 1.. optBSPp ( float (main.lenX / BSPp) * main.sfactor
                 + xfBSPgx (BSPp, 2 * min (BSPp - 1, 2)) + 2.0 * fBSPl (BSPp))
```

This utilises the **optBSPp** built in function, described in section 4.7, to minimise a function of **BSPp**, where the constant *main.sfactor* converts the loop iterations into floating point operations. Since the values used in the above code are the same as those used in the performance prediction of figure 6.4, the algorithm would correctly use only three processors for the case $m = 3500$.

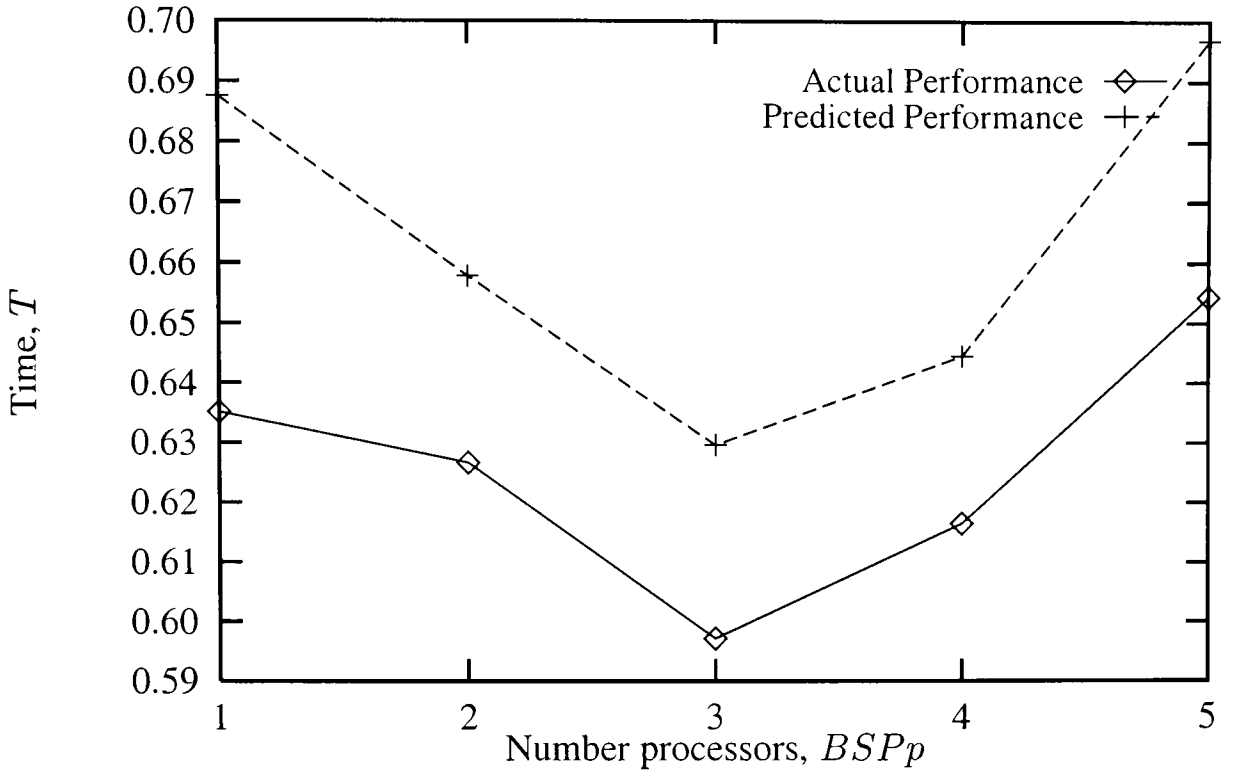


Figure 6.4: SP2 String Edit For $m = 3500$, $n = 500$.

Such an algorithm would be able to automatically adapt to the architecture that it is executing on, picking the optimum number of processors to use.

6.3 Reduction and Prefix Sums

The following example illustrates how to create a generic group that can perform reduction and prefix sums. The reduction operation takes an ordered set of values $(x_0, x_1, x_2..x_{n-1})$, a binary associative operator $+$ and returns $(x_0 + x_1 + x_2 + .. + x_{n-1})$. For example, if the binary associative operator is min then the reduction will return the minimum element of the set of values. The prefix sums operation takes the same input as the reduction algorithm but returns the set $(x_0, x_0 + x_1, x_0 + x_1 + x_0, .., x_0 + .. + x_{n-1})$.

The algorithm we implement follows that described in [Siniolakis, 1996a], but many other implementations can be found in [McColl, 1993a, Jájá, 1992, Leighton, 1992].

In Opal the reduction and prefix operators can be implemented in the same group as they perform very similar functions. The principle idea is to split the n values across the $BSPp$ processors such that each processor holds $\left\lceil \frac{n}{BSPp} \right\rceil$ values. The reduction operator can then be implemented using a t-ary tree as follows:

1. Each processor applies the associative operator + to the $\left\lceil \frac{n}{BSPp} \right\rceil$ values that it holds locally, storing the value obtained as its key in the t-ary tree.

2. A forward traversal of the tree is performed where each processor applies + to the values of its children, updating its key with the new value. At the end of this traversal, which takes $\log_t(BSPp)$ steps, the root processor holds the value of the reduction.

3. A reverse traversal of the tree is performed. This communicates the result from the root back to all other processors and is effectively a broadcast from the root.

The prefix sums operator is implemented in a very similar way except each node needs to record the values of its children for use in the reverse traversal. It can be implemented as follows:

1. Each processor applies the associative operator + to the $\left\lceil \frac{n}{BSPp} \right\rceil$ values that it holds locally, storing the value obtained as its key in the t-ary tree.

2. A forward traversal of the tree is performed where each processor applies + to the values of its children. The parent also records the value of its children for use in the reverse traversal. At the end of this step all processors that are an exact power of t will contain the appropriate value of the prefix sums.

3. A reverse traversal is performed where each parent passes the values of its children down the tree so that its children can apply a correction to their previously computed values.

We shall now described the group interface for such a generic reduction and prefix group. In order to make the group as re-usable as possible it is parameterised by the following variables:

1. The number of processors to use.

2. The degree, t, of the tree to be used.

3. The type, T, of the values that are being operated upon.

4. The functions to be applied to the data for the reduction and prefix sums.

The group interface has two entry points; one for reduction and one for prefix sums. The first entry, $prefix$, performs a prefix sums using the function sum that was supplied in the group instantiation. The second entry point, $reduction$, performs a reduction using the cmp function. Each

entry point accepts a single dimension unconstrained array. Such a group interface, $gPrefix$, is shown below:

```
-- The function cmp should return 0 if a = b, +'ve if a < b, -'ve if a > b

generic group gPrefix (   nprocs   : in integer;
                          t        : in integer;
                          T        : type;
                          cmp      : in function (a, b: in T) return integer;
                          sum      : in function (a, b: in T) return T     ) is

    entry prefix (1..nprocs) (data: in out array (<>) of T);

    entry reduction (1..nprocs) (data: in array (<>) of T; value: out T;
                                 proc: out integer; indx: out integer);
end gPrefix;
```

It should be noted that the entry points are in fact arrays of entries, using an all-to-all type library call, where the data has already been split across the processes. If it is the case that the values are not already distributed across the processes then this step must be performed before the prefix group is rendezvoused with[3].

The following segment of Opal code illustrates how the prefix sums are performed by a replicated $worker$ of the $gPrefix$ group. For clarity, the code to actually perform the prefix has been removed in order to demonstrate the use of rendezvous. For a full listing of the prefix group source code the reader is directed to chapter C.

The operation of this code can be described as follows:

1. Each $worker$ starts executing during the initial startup phase of the Opal program. The $worker$ then performs some initialisation and blocks at a **select** statement waiting for a rendezvous.

2. Depending on the rendezvous that is accepted the $worker$ processes then call an appropriate superstep procedure to perform the actual work. At the end of the **accept** statement the results are communicated back to the rendezvousing process.

[3] Alternatively, a prefix group could be written that does this distribution of data for you and which would only use a one-to-one type library call.

3. Each *worker* process then loops back to the **select** statement waiting for another rendezvous. If no rendezvous is received and the Opal program terminates, then the **terminate** alternative is selected and all the *worker* processes will finish execution.

```
with gPrefix, gPSlave, Math;
replicated gPrefix process body gPSlave (i:1..gPrefix.nprocs) is

    declare local variables required to perform the prefix

    procedure initialise () is
    begin -- Initialise values such as log_t(p).
    end initialise;

    superstep procedure doprefix
                (data: in out array (<>) of gPrefix.T;
                 sumf: in function (a, b: in gPrefix.T) return gPrefix.T) is
    begin
        body containing supersteps
    end doprefix;

    superstep procedure doreduct ( data: in array (<>) of gPrefix.T; value: out gPrefix.T;
                                   proc: out integer; indx: out integer;
                                   cmpf: in function (a, b: in gPrefix.T) return integer) is
    begin
        body containing supersteps
    end doreduct;

begin -- The following code is executed when gPSlave.opl is started
    initialise ();
    loop
        select
            accept prefix (i) (data: in out array (<>) of gPrefix.T) do
                doprefix (data, gPrefix.sum);
            end accept;
        or
            accept reduction (i) ( data: in array (<>) of gPrefix.T; value: out gPrefix.T;
                                   proc: out integer; index: out integer) do
                doreduct (data, value, proc, index, gPrefix.cmp);
            end accept;
        or
            terminate;
```

```
        end select;
    end loop;
end gPSlave;
```

It should be noted how the $worker$ process is accessing the generic parameters of the group using the normal dotted notation, e.g. the replicated index is $1..gPrefix.nprocs$, causing the correct number of processes to be started depending on the value of the generic parameter $nprocs$.

This library group is using a simple loop where it performs a small amount of initialisation and then blocks waiting for a request to perform work. When a request is received the processes become active and compute the result, returning back to the dormant state once the rendezvous has completed. This is typical of a library group, allowing us to perform *parallel overlap* as described in section 5.5 and giving a concrete example of some of the assumptions made in section 5.8.

The $worker$ process also has a specification in the file gPSlave.spl. This contains the shared variables for the prefix sums worker, a listing of which can be found in chapter C.

6.3.1 Performance Prediction

We can see that in step 1 of the algorithm each process is performing an amount of computation $\left\lceil \frac{n}{BSPp} \right\rceil .c$ where c is the time to perform one action of the $+$ operator. In step 2, the forward traversal of the tree, $\lceil log_t(BSPp) \rceil$ supersteps are performed with each superstep implementing a t-relation. In step 3 we perform a reverse traversal of this tree where another t-relation is implemented. Each process then fetches a correction factor from its neighbour[4] and applies this to its local data, resulting in an amount of computation $\left\lceil \frac{n}{BSPp} \right\rceil .c$. Summing these values and allowing for synchronisation, this gives a total cost, T, of:

$$T = 2. \left\lceil \frac{n}{BSPp} \right\rceil .c + 2 \lceil log_t(BSPp) \rceil .(t.g + 2l) + g + 3l \tag{6.3}$$

To accurately predict the execution time of the parallel prefix sums algorithm we measured the value of c using a sequential prefix sums algorithm, the results of which are shown in table 6.2.

Using these values and those of $N_{\frac{1}{2}}$ and g_∞ from tables D.1 and D.2, we can predict the performance of the prefix sums algorithm for $t = 3$ and $n = 80000$. Figures 6.5 and 6.6 show the results of such predictions, again demonstrating that the actual and predicted performance is very close.

[4] A 1-relation requiring one superstep.

Architecture	c
SP2	2.270e-7
SGI	8.615e-8

Table 6.2: c For Prefix Sums.

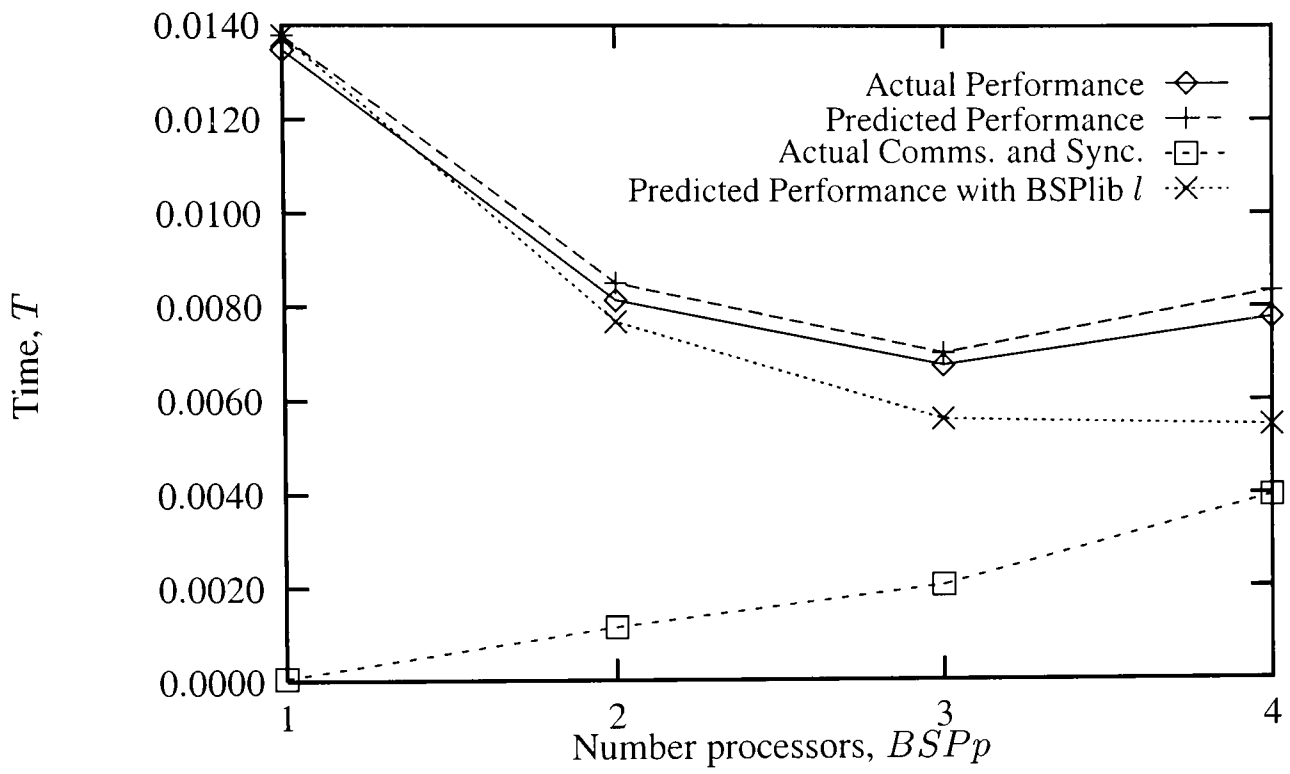


Figure 6.5: Prefix Sums Prediction for Silicon Graphics Power Challenge (SGI).

6.3.2 Parallel Speedup

For reasons described in section 6.2.2 the parallel speedup of figures 6.5 and 6.6 are not as good as they could be. For this reason the graphs labelled *Performance Prediction with BSPlib l* show the predicted performance if we were to use the BSPlib value of l. As can be seen from these graphs, using the BSPlib value of l leads to a reasonable parallel speedup.

Even though we can improve the parallel speedup of the prefix group it can still be seen that the performance gain in adding more processors is starting to diminish at around four processors. This is due to the problem size that has been selected, i.e. we have selected the problem size such that approximately 50% of the total run time is spent communicating[5], which means that after a number of processors have been added our algorithm becomes communication bound. With a larger problem size we could demonstrate a better parallel speedup for four processors or more.

[5]This was chosen so that we were not predicting an algorithm whose runtime mainly consisted of sequential code.

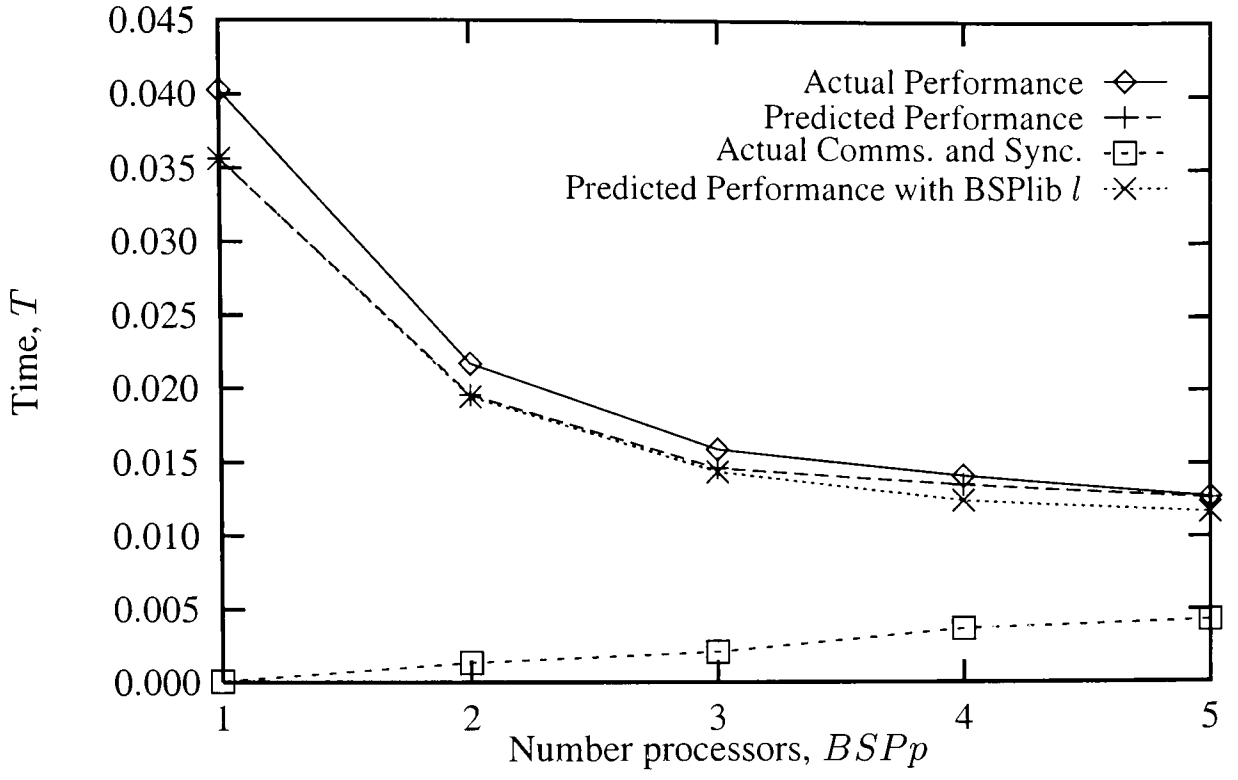


Figure 6.6: Prefix Sums Prediction for IBM SP2.

6.3.3 Automatic Architecture Adaption

In [Siniolakis, 1996a] it was calculated that the optimum t-ary tree to use for the prefix sums algorithm is such that $t = min(p, \left\lceil \frac{l}{g} \right\rceil)$. This can be used to make the prefix sums algorithm automatically adapt to the architecture that it is being executed on. In this example we can perform such automatic adaption at the generic instantiation stage:

---

with integers;
group intprefix **is new** gPrefix (**BSPp**, **min**(**BSPp**, **ceil** (**BSPl** / **BSPg**)),
integers.T, integers.cmp, integers.sum);
end intprefix;

This would create a prefix sums group that will use **BSPp** processors and a $min(p, \left\lceil \frac{l}{g} \right\rceil)$-ary tree. When this group is executed the Opal runtime system would substitute values for **BSPp**, **BSPg** and **BSPl** relative to the current architecture, therefore ensuring that the most efficient tree is used.

6.4 Bucketing Algorithm

Data bucketing is a common operation that is utilised in a number of parallel algorithms. The principal idea is that given a set of data we should partition it into a number of ordered buckets, with each bucket receiving approximately the same number of elements. The data is partitioned such that each bucket only receives elements that are less than its right neighbour.

This can be more formally specified as follows; given k buckets $(B_1, B_2, B_3, ..B_k)$ the bucketing algorithm will split a set of data D such that each bucket is of approximately equal size and such that all elements in B_i are less than those in B_j if $i < j$.

The algorithm that has been implemented is based on the deterministic bucketing algorithm of [Siniolakis, 1996a]. We shall pick the number of buckets, k, to be **BSPp** therefore allowing a set of data to be partitioned across **BSPp** processes. The operation of such an algorithm can be described as follows:

1. The data set D is distributed across the **BSPp** processes such that each process receives approximately the same number of elements. Note that this data does not currently conform to the ordering rule of bucketing - it is an initial distribution.

2. Each process performs a sequential sort on the local data that it holds and selects **BSPp** evenly spaced elements (splitters), ensuring that the last element selected is the last element it holds.

3. Processor 1 is then sent the splitters of all the other processors[6], and it proceeds to sort them. Since the splitters from each processor are in increasing order, processor 1 need not perform a full sort, but instead can use a **BSPp** way merge of each of the streams of splitters it received.

4. Processor 1 selects **BSPp** evenly spaced splitters from the splitter array it just merged and broadcasts them, using a binary tree, to all other processors.

5. Each processor now counts the number of its local data elements that would fall into the **BSPp** buckets defined by the splitters it just received. Let these counts be held in an array N of size **BSPp**.

6. **BSPp** prefix sums operations are then applied to N in order to sum the total number of elements in each bucket.

[6]This broadcast should be performed using a tree, but to keep the algorithm short we use a single step broadcast.

7. Each processor now knows exactly how many elements are in each bucket and can route its local data to specific positions in remote arrays.

The design of this Opal algorithm centres around the re-use of the prefix sums algorithm described in the previous section. While we do not use the exact $gPrefix$ group, since we require a multi way prefix, the principle is the same, i.e. we utilise the services of another group in order to implement the bucketing algorithm. We also use library packages to perform the sequential sorting and merging of data, therefore reducing the amount of code to be written and decreasing the probability of introducing programming errors.

The bucketing group consists of a generic group interface, gbucket.grp, and a replicated process in the files bucketWorker.spl and bucketWorker.opl. The parameters to the generic group include the number of processes to use and the maximum amount of data to be bucketed. Other parameters could also be passed, such as the type of the data to be bucketed and the t-ary tree to use in the broadcast. However, to keep this example simple these parameters have been hard coded, although it would be trivial to make them variable.

The $bucketWorker$ process specification declares shared variables for holding the data in each bucket and for communicating splitter elements. We shall not include the Opal specification for the $bucketWorker$ process, but instead concentrate on the implementation of the process body:

```
with gbucket, bucketWorker, multiprefix, SeqSort, SeqMultiMerge;

replicated gbucket process body bucketWorker (i:1 .. gbucket.nprocs) is
    log2nprocs, twojm1, twoj, twojp1: integer;
    localnumInBucket: array (1 .. gbucket.k) of integer;
begin
    accept perform (i) (data: in out array (<>) of integer; maxIndex: in out integer) do
    begin "local sort" superstep
        log2nprocs := round (log2 (float (gbucket.nprocs)));
        SeqSort.doSort (data, 1, maxIndex);
        #bucketWorker.data (1..maxIndex) := data (1..maxIndex);
        for j in 1 .. gbucket.k-1 loop
            #bucketWorker.splitters (j) := #bucketWorker.data (j * (maxIndex / gbucket.k));
        end loop;
        #bucketWorker.splitters (gbucket.k) := #bucketWorker.data (maxIndex);
        bucketWorker(1).allSplitters (i) := #bucketWorker.splitters;
    end superstep;
```

```
begin "p1 sort splits" superstep
    if i = 1 then
        SeqMultiMerge.performStepped (#bucketWorker.allSplitters,
                                      #bucketWorker.splitters, gbucket.k);
    end if;
    twojm1 := 1; twoj := 2; twojp1 := 4;
end superstep;

for j in 2..log2nprocs + 1 loop
    begin "bcast splits " & string (j-1) superstep
        if i >= twoj and i < twojp1 then
            #bucketWorker.splitters := bucketWorker (twojm1).splitters;
        end if;
        twojm1 := twoj; twoj := twojp1; twojp1 := twojp1 * 2;
    end superstep;
end loop;

declare
    l: integer;
begin "count bucks" superstep
    l := 1;
    for j in 1 .. gbucket.k loop #bucketWorker.numInBucket (j) := 1; end loop;
    for j in 1..maxIndex loop
        while #bucketWorker.data (j) > #bucketWorker.splitters (l)
                and l < gbucket.k loop
            inc (l);
        end loop;
        inc (#bucketWorker.numInBucket (l));
    end loop;
    localnumInBucket := #bucketWorker.numInBucket;
end superstep;

begin "multiprefix" superstep
    multiprefix.prefix (i) (#bucketWorker.numInBucket);
end superstep;

declare
    globindex, lindex: integer;
begin "distribute data" superstep
    lindex := 1;
    for j in 1 .. gbucket.k loop
        globindex := #bucketWorker.numInBucket (j) - localnumInBucket (j);
        if j = i then
            #bucketWorker.data (globindex + 1..globindex + localnumInBucket (j) - 1)
```

```
                        := data (lindex..lindex + localnumInBucket (j) - 2);
                    #bucketWorker.data (globindex + localnumInBucket (j)) := gbucket.marker;
                else
                    bucketWorker(j).data (globindex + 1..globindex + localnumInBucket (j) - 1)
                        := data (lindex..lindex + localnumInBucket (j) - 2);
                    bucketWorker (j).data (globindex + localnumInBucket (j)) := gbucket.marker;
                end if;
                lindex := lindex + localnumInBucket (j) - 1;
            end loop;
            maxIndex := bucketWorker (gbucket.nprocs).numInBucket (i);
        end superstep;

        begin "assign parameters" superstep
            data (1..maxIndex) := #bucketWorker.data (1..maxIndex);
        end superstep;
        end accept;
    end bucketWorker;
```

This process blocks on the *perform* rendezvous until it is requested to perform a bucketing task, employing an array of entry points and therefore utilising an all-to-all type library call mechanism. Once a rendezvous is accepted the processes sort the data in the "local sort" superstep, select *gbucket.k* splitters and broadcast them to processor 1.

Processor 1 then performs a multi way sequential merge on the **BSPp** splitter arrays it received, selecting an element to put in *#bucketWorker.splitters* every *gbucket.k* elements. These splitters are then broadcast to the other processes in the "bcast splits" supersteps.

After each process has counted the number of its elements that would fall into each of the defined buckets it requests a multi way prefix sums operation to be performed. This uses an all-to-all library call in the "multiprefix" superstep, utilising the services of the *multiprefix* group. It should be noted that up to this point the *multiprefix* group has been dormant, and that as soon as the *bucketWorker* process performs the rendezvous it also becomes dormant. This is a typical use of the Opal library mechanism and provides a concrete example of some of the assumptions made in section 5.8.

Once the multi way prefix sums operation has completed, each process routes the data to remote processes in the "distribute data" superstep. It should be noted that each process places a special marker value, *gbucket.marker*, at the end of each block of data that it routes. This allows processes

that use this group to further utilise the order of the blocks of data that are written[7].

Finally, once the data has been routed, the parameters of the **accept** statement are updated and passed back to the calling process.

6.4.1 Performance Prediction

To predict the performance of this algorithm we first analyse the *multiprefix* group. This group performs the same task as the normal *prefix* group except that it performs M prefixes in lock step. We can therefore see that if we scale the communication and computation costs of equation 6.3 by a factor of M then we will get the performance of a multiple prefix.

The *bucket* group uses *multiprefix* to perform prefixes on **BSPp** arrays each of size **BSPp**, i.e. $M = n = BSPp$. If we rewrite equation 6.3 using these values, and multiply it by a factor of M, then we obtain equation 6.4, the cost the *multiprefix* group.

$$T_m = 2.BSPp.c + 2\lceil log_t(BSPp)\rceil.(BSPp.t.g + 2l) + BSPp.g + 3l \qquad (6.4)$$

Figure 6.7 shows the actual and predicted performance of the *multiprefix* group. The predicted values were obtained by using equation 6.4 with the value of c taken from table 6.2 and the values of g_∞, $N_{\frac{1}{2}}$ and l from table D.1. Since the actual and predicted graphs are very close it shows that using the algorithm as a group does not invalidate the BSP cost model. It also shows that there is little or no interference between the h-relations of different groups.

It should be noted that equation 6.4 does not scale. In fact, as the number of processors increases the performance decreases as is shown by figure 6.7. However, this is a necessary part of the bucketing algorithm and these losses can be offset by performance increases in other areas.

We must now consider the performance of the bucketing algorithm not including the rendezvous with the *multiprefix* group. We can see from the first superstep called "local sort" that this performs a sort of n data items, and then sends a **BSPp** word array to processor 1. If the cost of performing the sort is C then the cost of this superstep is $C + BSPp^2.g + l$.

In the second superstep, "p1 sort splits", processor 1 performs a merge of the $BSPp^2$ splitters that it received. If the cost of merging one element is c then the cost of this superstep is $BSPp^2.c + l$.

The next series of supersteps, "bcast splits", broadcasts the $BSPp$ splitters that processor 1 has

[7] Since the local data is sorted, and we use an array slice to perform the routing, the block that each processor writes will also be sorted. This method of delimiting the data will be utilised in the next example.

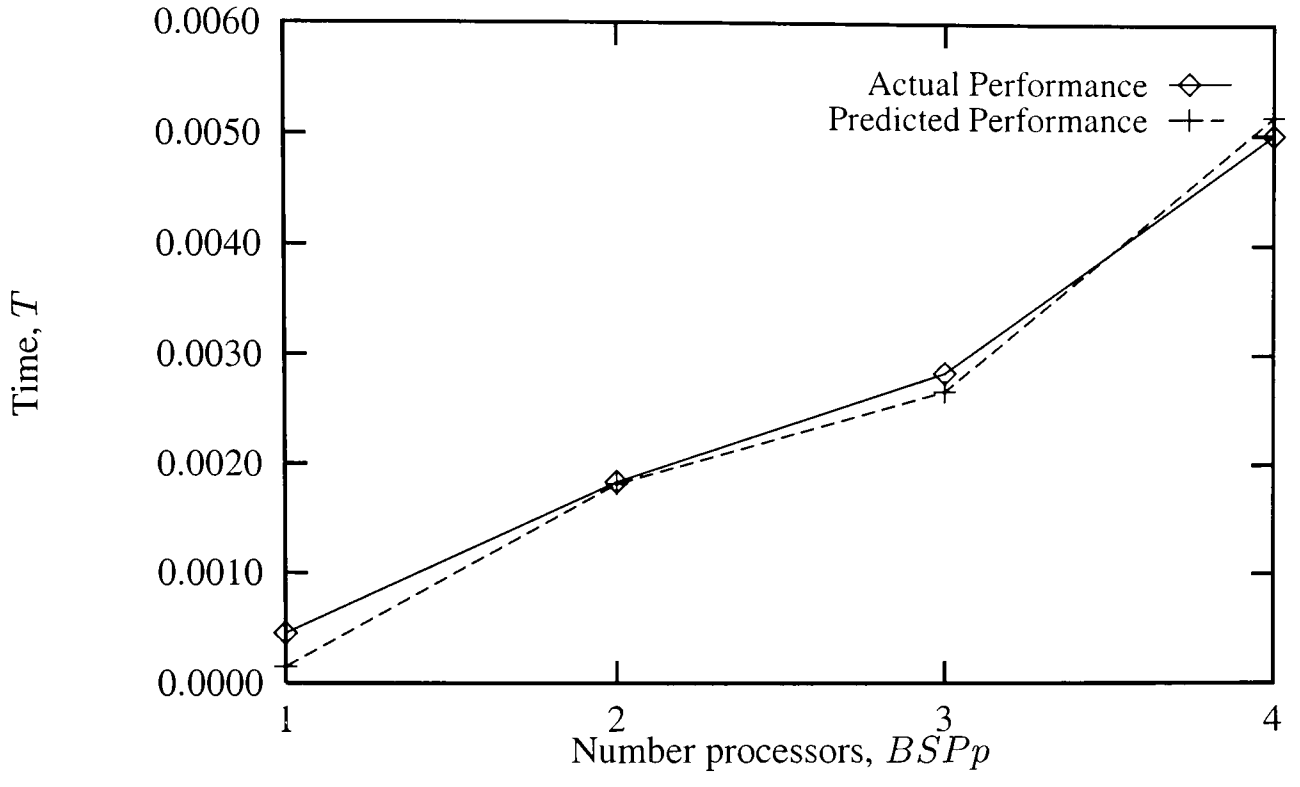


Figure 6.7: Multiprefix Prediction for Silicon Graphics Power Challenge.

selected to all other processors. The cost of this superstep is $\lceil log_2(BSPp) \rceil .(2g + l)$.

In the "count bucks" superstep each processor then counts the number of elements that fall into each bucket. If we assume that the cost of determining the bucket of each element is the same as merging an element then the cost of this superstep is $BSPp.c + \frac{n}{BSPp}.BSPp.c + l$.

The next superstep, "distribute data", distributes the data amongst the $BSPp$ processors. If we assume that the data is evenly bucketed, and that no processor has any elements that are in its bucket, then each processor will receive and transmit $\frac{n}{p}$ data elements to other processors, i.e. it exchanges the $\frac{n}{p}$ elements that it currently has for its own bucket. Ignoring the local computation this superstep costs $\frac{ng}{p} + l$.

The final superstep, "assign parameters", simply assigns a $\frac{n}{p}$ element array to the rendezvous actual parameters. This has cost $\frac{nc}{p} + l$.

Assuming that c is much less than g, if we sum the costs for all of these supersteps then we obtain the time, T_b, of the bucket group not including the rendezvous:

$$T_b = C + BSPp^2.g + \lceil log_2(BSPp) \rceil .(2g + l) + \frac{ng}{BSPp} + 5l \tag{6.5}$$

In order to accurately predict the bucketing algorithm we measured the time taken to perform a

sort using a sequential algorithm. The number of elements was chosen so that it exactly machines the different values of C in equation 6.5 when $n = 3000$. Table 6.3 shows the results of this benchmarking.

Number of Elements	C
3000	0.014814
1500	0.006364
1000	0.004322
750	0.003110

Table 6.3: C, the Time for Sorting on the SGI.

We must now consider the performance of the whole bucketing algorithm, including the rendezvous with the *multiprefix* group. As described in section 5.8, we must charge an amount for the communication of the rendezvous parameters. In this example each bucket process is rendezvousing with a unique multiprefix process, therefore using an all-to-all library call. Since the *bucketWorker* process does not perform any computation or communication during the rendezvous we use equation 5.9 to predict its performance.

During this rendezvous an array of integers of size **BSPp** is sent out, and then an array of approximately the same size is received back again, but with the multiprefix performed. This equates to a 2.**BSPp**-relation being implemented in order to transfer the rendezvous parameters.

If the time taken to complete the multiprefix algorithm is T_m, and the time taken to perform the bucketing algorithm minus the rendezvous superstep is T_b then using equation 5.9 the total time for the Opal program, T, is:

$$T = T_b + T_m + 2.BSPp.g + l \tag{6.6}$$

Figure 6.8 shows the actual and predicted performance of the whole bucketing algorithm for $n = 3000$, including the rendezvous with the multiprefix group. To generate the predicted performance we calculated T_b using equation 6.5 with the value of C from table 6.3 and the values of g_∞ and $N_{\frac{1}{2}}$ from table D.1. Equation 6.6 was then used to combine T_b and T_m into the predicted performance of the complete algorithm.

The majority of the error in figure 6.8 is a result of the total exchange that occurs in the "distribute data" superstep and is not due to the rendezvous. This is because the values of $N_{\frac{1}{2}}$ and g_∞ that were used to predict the performance were obtained by routing random h-relations and not total exchanges. We therefore predict that the "distribute data" superstep will be performed much

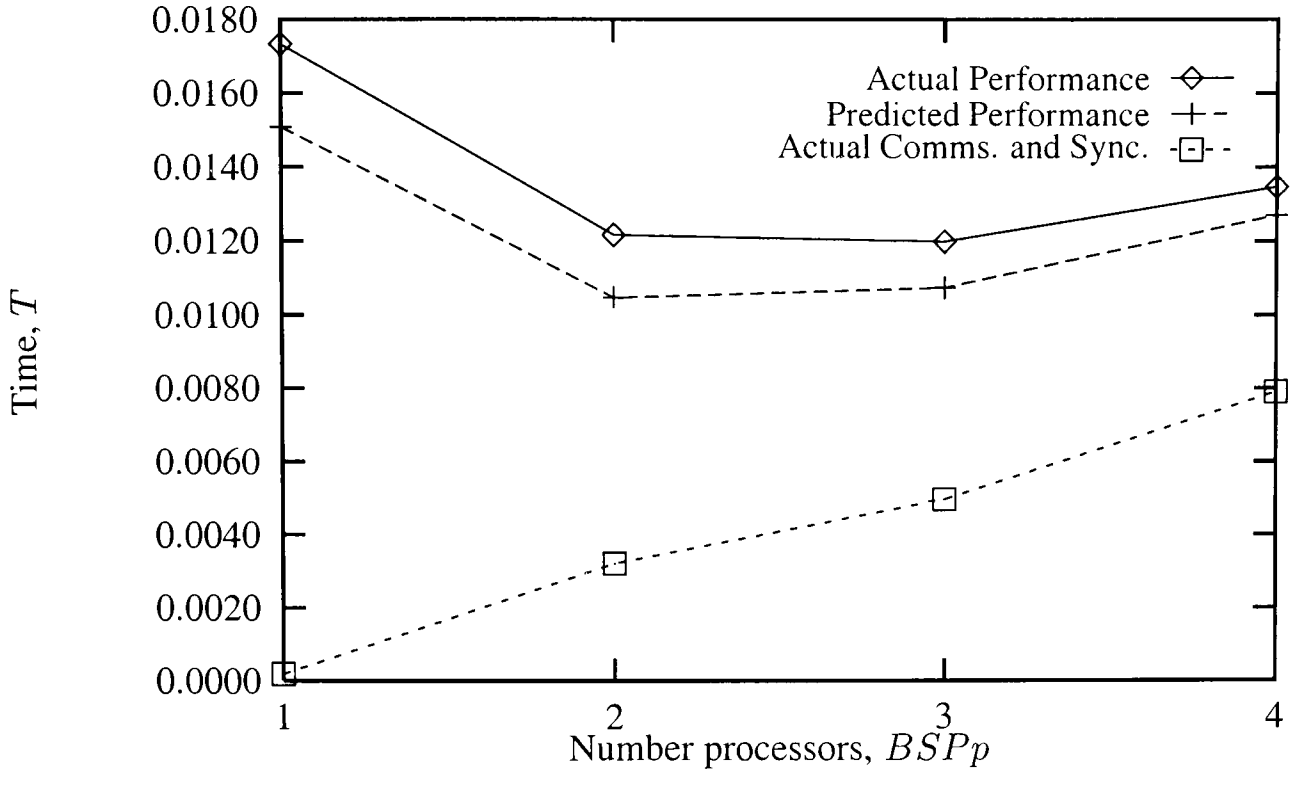


Figure 6.8: Bucketing Prediction for Silicon Graphics Power Challenge.

quicker than it actually is.

From these figures it can be seen that using Opal groups in this manner does not invalidate the BSP cost model. This is expected, since in this example the rendezvous is performed in its own superstep, resulting in no other communication being performed during the rendezvous. While we could move the rendezvous into the "count bucks" superstep[8] we must synchronise immediately after we issue the request as the results of the rendezvous are required in the next superstep. Therefore, even if the rendezvous was moved to this superstep there would be no communication or computation performed while the rendezvous is active.

6.4.2 Parallel Speedup

As has been the case with previous examples, figure 6.8 shows a rather poor parallel performance speedup. Again, this is due to the selection of the problem size n, i.e. n was selected so that approximately 50% of the time was spent communicating or synchronising.

By examining equation 6.5 we can see that there is a $BSPp^2g$ term due to the broadcasting of the splitters to processor 1. The broadcast used in this implementation of the bucketing algorithm is the naive one stage broadcast. Using a two-stage broadcast would reduce this cost to $O(BSPp.g+$

[8]This superstep contains only local computation which must be performed before the rendezvous is called.

l), making the algorithm more scalable.

We can therefore see that by using a more realistic problem size, and improving broadcasts, this algorithm should exhibit reasonable parallel speedup.

6.5 Iterative Sorting

In this section we demonstrate how the generic bucketing group can be used to implement a sorting algorithm, as described in [Siniolakis, 1996a, Siniolakis, 1996b]. The operation of the algorithm is very simple and can be described as follows:

1. Bucket the data so that each process holds approximately the same number of elements.
2. If each process now sorted its bucket then the whole of the data set would be sorted. However, we know from the operation of the bucketing algorithm that each process will receive **BSPp** ordered streams in its bucket, with each stream being separated by a marker. We can therefore construct a heap from the head of each stream and then iteratively remove an element from the heap, inserting an element from the same stream as the element that was removed.

The Opal source code for such a process is show below. This process utilises the bucketing group *bucket*, which in turn uses a multi prefix group, but the programmer of the sort process need not be aware of this. All the programmer would need to do would be to instantiate the generic bucket group with the appropriate parameters.

```
with sort, bucket, worker, SeqHeap;
replicated sort process body worker (i: 1..BSPp) is
    localData : array (1..2 * sort.maxPartition + BSPp) of integer;
    heap : array (0..BSPp,1..2) of integer;
    indexes : array (1..BSPp) of integer;
    N, maxIndex, blockSize : integer;
begin
    begin "init" superstep
        Read the input data, calculate the block size and other inititialisations
    end superstep;

    begin "bucket rendezvous" superstep
```

```
        bucket.perform (i) (localData, maxIndex);
    end superstep;

    declare
        j: integer; result: array (1..2) of integer;
    begin "sort + write" superstep
        j := 1; N := 0;
        for l in 1..bucket.k loop
            indexes (l) := j;
            if localData (j) != bucket.marker then
                SeqHeap.insert (localData (j), l, heap, N);
            end if;
            while localData (j) != bucket.marker loop inc (j); end loop;
            inc (j);
        end loop;

        j := 1;
        while N > 0 loop
            SeqHeap.remove (heap, N, result);
            #worker.data (j) := result (1); inc (indexes (result (2)));
            if localData (indexes (result (2))) != bucket.marker then
                SeqHeap.insert (localData (indexes (result (2))), result (2), heap, N);
            end if;
            inc (j);
        end loop;
        -- Output the results.
    end superstep;
end worker;
```

This example shows how we can construct Opal programs from library groups in a manner that follows good software engineering principles. In fact, the sort process further utilises a standard package of the Opal system, this being the *SeqHeap* package.

This example again demonstrates how we must barrier synchronise as soon as we perform the rendezvous with the bucketing group. If this synchronisation was not performed then we could not guarantee that the *localData* variable would be ready for the "sort + write" superstep.

6.6 Convex Hull

Given a set of n points in a plane the convex hull is the smallest polygon that contains all points. Alternatively the convex hull can be described as the shortest path surrounding all points. In this example we shall implement a BSP convex hull algorithm that is described in [Siniolakis, 1996a], again demonstrating the use of library groups.

Assuming that the n points are distributed evenly amongst the p processes, the parallel convex hull algorithm can be described as follows:

1. A reduction is initiated, using the min operator, to find the point with the minimum y coordinate. This point is on the convex hull.
2. Each process selects from its local points the point with the minimum polar angle with respect to the previously found point that is on the convex hull.
3. A reduction is initiated, using the min operator on the polar angles, to find the next point on the convex hull.
4. Steps 2 and 3 are repeated until we return to the point that we found in step 1.

The first step in the development of an Opal program for the convex hull problem is to define a type that can be used to represent points in a plane. We then need to define an operator that can compare two points and return the one with the minimum polar angle. To do this a package called $points$ is created in the file points.grp, and a package body in points.grp.body:

```
package points is
    type T is record
                x, y    : integer;    -- (x,y) coordindate.
                angle   : real;       -- Polar angle.
                index   : integer;    -- Array index in the local processor.
            end record;

    -- cmp compares the y coords if the a->angle is 400.0, otherwise it compares angles
    function cmp (a, b: in T) return integer;
    function sum (a, b: in T) return T;
end points;
```

The *points* group is then used to instantiate a prefix sums group that will perform reductions on points in a plane.

The process that performs the convex hull algorithm, *convexSlave*, is placed in the file convexSlave.opl with a process specification in convexSlave.spl. In this example there is no interprocess communication for *convexSlave* since the majority of the work is done by the prefix sums group, so we only include the *convexSlave* process body:

```
with convexhull, points, pointsprefix, InlineFileIO;
replicated convexhull process body convexSlave (i: 1..BSPp) is
    proc, indx, startproc, startindex, onhull, angleindx, hullidx: integer;
    anglearray: array (1..1) of points.T; minpoint: points.T;
    hullpoints: array (1..BSPp * convexhull.blockSize) of points.T;
    allpoints: array (1..convexhull.blockSize) of points.T;

    function theta (coord1, coord2: in points.T) return real is
    begin
        Returns a value with the same properties as the polar angle without using trig
    end theta;
begin
    begin "read data, initial reduction" superstep
        Read data in from a file.
        for j in 1..convexhull.blockSize loop
            allpoints (j)->index := j; allpoints (j)->angle := 400.0;
        end loop;
        pointsprefix.reduction (i) (allpoints, minpoint, startproc, startindex);
    end superstep;

    begin "initialise" superstep
        onhull := 0; hullidx := 1; minpoint->angle := 0.0;
    end superstep;

    loop
        declare a, lasta, mina: real;
        begin "calculate next convex hull point" superstep
            lasta := minpoint->angle; mina := 360.0; indx := 1;
            for j in onhull + 1..convexhull.blockSize loop
                a := theta (minpoint, allpoints(j));
                if (a > lasta) and (a < mina) then mina := a; indx := j; end if;
            end loop;
            allpoints (indx)->angle := mina; anglearray (1) := allpoints (indx);
```

```
            pointsprefix.reduction (i) (anglearray, minpoint, proc, angleindx);
        end superstep;

        declare tempP: points.T;
        begin "assign convex hull point" superstep
            if i = 1 then inc (hullidx); hullpoints (hullidx) := minpoint; end if;
            if i = proc then
                inc (onhull); tempP := allpoints (onhull);
                allpoints (onhull) := minpoint; allpoints (minpoint->index) := tempP;
            end if;
        end superstep;
        exit when proc = startproc and minpoint->index = startindex;
    end loop;

    Superstep where processor 1 outputs hullpoints to a file
end convexSlave;
```

It should be noted how the calls to the rendezvous are in supersteps that only contain local computation. The synchronisation is required immediately after the rendezvous since the next superstep requires the rendezvous results. Again this demonstrates how supersteps that contain rendezvous are likely to be relatively simple and that in most cases a barrier synchronisation will occur as soon as the rendezvous is requested.

6.7 Summary

In this chapter we have demonstrated how the group mechanism of Opal can be used to create libraries of parallel algorithms. These libraries can be parameterised through the use of generic groups, allowing the maximum scope for code re-use. Such mechanisms are employed by sequential languages such as Ada and Modula-3, but are lacking in parallel languages, as in this environment modules need to encompass the services of many processes. If parallel solutions are to be employed in large scale software projects then modular programming enforces good software engineering practices, minimising the amount of code that has to be developed and reducing the number of errors that will be introduced.

An example of such modular parallel programming is the iterative sorting implementation. This algorithm was implemented using the services of a parallel bucketing group and a sequential package for maintaining a heap. The implementation of the bucketing group also relies on the

services of a multi prefix group, but this is hidden from the programmer of the sorting algorithm. This technique was demonstrated again when we re-used a prefix group in order to implement a convex hull algorithm.

While modular programming has obvious benefits in the development of large scale parallel systems, it must be introduced into the BSP framework without compromising the cost model. We have illustrated that using the performance prediction methods of chapter 5, Opal programs that contain all-to-all group rendezvous can still be accurately analysed. Such predictions were performed via the analysis of source code and through the use of a profiling tool that is provided with the compiler. These examples illustrated that the use of *parallel overlap* and *superstep collapsing* does lead to accurate performance prediction results.

In section 5.8 we stated a number of assumptions that are made during the performance prediction of Opal programs that contain multiple groups. We have illustrated that if the Opal group mechanism is properly used as a method for providing parallel libraries then these assumptions are valid, i.e. the examples we have given show that:

- During a rendezvous, the calling process becomes dormant as soon as the rendezvous is issued. It is also the case that there will be a minimal amount of local computation and remote communication in the superstep that contains the rendezvous call.
- The accepting groups processes will perform many supersteps compared to the single superstep of the entry process.
- Execution of the subgroups causes minimal interference with the parent groups processes.

We have demonstrated how the Opal language can be used to make algorithms architecture independent, while also allowing them to adapt to the specific architecture they are executing on. This was illustrated through the instantiation of generic groups with expressions that contain the BSP parameters and through the use of the **optBSPp** function that provides a mechanism for determining the optimum number of processes to use for a given problem size.

In summary, sequential modular languages provide interfaces to library functions where a single thread of control passes through the library. Opal uses a similar paradigm to describe processes, where the services of a process are the shared variables that it exports. Opal uses a further level of abstraction to form a group interface, where the group describes the services of a set of cooperating processes. While such mechanisms make small parallel programs more difficult to write, as multiple interfaces need to be designed, they provide a robust framework for the development of large scale systems that are both portable and predictable.

Chapter 7

Conclusion

In this thesis we have presented a BSP language that offers architecture independent modular parallel programming in a robust environment, with predictable performance. This chapter summarises the work and its conclusions and also makes suggestions for further areas of study.

7.1 Summary

In chapter 1 we compared the success of the sequential software industry against the relative failure of its parallel counterpart. This was attributed to the lack of a parallel model of computation, similar to the Von Neumann model for sequential computing, which has limited the production of architecture independent, predictable, parallel software. We outlined a number of existing parallel models of computation such as the PRAM [Fortune and Wyllie, 1978], BSP [Valiant, 1990, Valiant, 1989, Skillicorn *et al.*, 1996], LogP [Culler *et al.*, 1993b], BSP* [Bäumker *et al.*, 1996, Bäumker *et al.*, 1995], LogGP [Alexandrov *et al.*, 1995] and CGM [Dehne *et al.*, 1993]. Of these models BSP was seen to be the simplest, while offering good performance prediction and portability properties [Bilardi *et al.*, 1996].

We then examined existing parallel languages and libraries including FORK [Hagerup *et al.*, 1991], Jade [Lam and Rinard, 1991, Scales *et al.*, 1991], Split-C [Culler *et al.*, 1993c, Culler *et al.*, 1993a], HPF [Forum, 1993], OCCAM-3 [Barrett, 1992], BSPlib [Hill *et al.*, 1997c], GPL [McColl and Miller, 1995, McColl, 1994a] and the Green BSP library [Goudreau *et al.*, 1995]. Of these languages only OCCAM-3 has made an attempt to provide a parallel library mechanism using modules, but this suffered from architecture dependence and lack of performance

predictability. While languages such as GPL and Split-C, and libraries such as BSPlib and the Green BSP library, offer predictable performance they have not examined any issues regarding parallel libraries, modularity or good software engineering principles.

It is often the case that large scale software projects rely on the decomposition of the problem into a number of separate tasks, with each task solved by a different programming team. Indeed, due to the *grand challenge* nature of parallel applications it is likely that such software projects will be as large, if not larger, than their sequential counterparts. Such decomposition of problems into a number of independent tasks relies on well defined interfaces between the different teams of programmers.

It was stated that sequential applications have succeeded due to their inherently portable nature and predictable performance. We can therefore see that while the BSP model has provided portability with predictable performance, no languages have been designed which allow the decomposition of a problem into a number of different elements, with stringent interfaces between each sub problem. The Opal language was designed to address this area and offers parallel modular programming with predictable performance, using the BSP model as its basis.

In chapter 2 we reviewed the BSP model and discussed issues such as *single superstep* and *double superstep* reads. We also presented the idea of *automatic algorithm adaption*, where an algorithm is written in such a manner that it can adapt to the BSP parameters of the architecture it is being executed on. While this technique has been discussed in [Knee, 1994b, Cheatham *et al.*, 1995] no real analysis has been performed as to what tools the programmer needs in order to apply this method.

The design rationale of the Opal language was discussed in chapter 3, where we defined the issues that were important in the design of a modular language for the BSP model. We described how the calls to parallel libraries may be either all-to-all, one-to-all or any-to-any and discussed how *passive* libraries mechanisms, such as procedure call, cannot easily support the any-to-any mechanism. For this reason we decided upon an active library mechanism, where a library consists of a group of processes that are executing asynchronously with respect to the processes in other groups.

In chapter 3 we noted that on some existing parallel architectures it is difficult to support dynamic processes [Skillicorn *et al.*, 1996] and that static processes not only simplify the implementation of runtime systems, but also improve the performance prediction properties of algorithms. For these reasons Opal was designed on the assumption that static processes should be employed and that other methods should be integrated into the language to ease the restrictions that this introduces.

The key rationale of this chapter was to produce a BSP language that provided libraries of parallel code, whilst not destroying the cost analysis features of the BSP model. Such a language should also follow good software engineering principles [Sommerville, 1989] and hence allow large software projects to be more viable.

In chapter 4 we defined Opal, a MIMD language designed to allow libraries of parallel code to be developed for the BSP model. We described how a process consists of a specification and a body, where variables declared in the specification are placed in shared memory and are accessible to all other processes. This modular approach was extended to groups, where a group provides an interface to the services of a set of processes.

To improve the amount of code re-use that could be performed, generic groups were introduced. These allow parameterised groups to be written, where at the point of instantiation actual values are provided for the generic parameters. This allows some of the restrictions of static processes to be alleviated, whilst also allowing groups to be instantiated based upon the values of the BSP parameters, i.e. automatic algorithm adaption.

The Opal language provides further support for automatic algorithm adaption through the use of the **optBSPp** function. This function takes an expression as a parameter and evaluates it for all values of the BSP parameters[1], returning the value of **BSPp** that would minimise the expression. This allows algorithms to perform operations such as selecting the optimum number of processors to use based upon the cost analysis of the algorithm.

Other constructs such as unconstrained arrays, packages, variable length arrays and generic types allow code to be developed which is portable in nature. Such mechanisms are present in many modular sequential programming languages. The language was also designed such that inter-process communication and library calls can be statically type checked by the compiler. thus ensuring that the maximum number of errors are caught at the development stage of the software.

Although parallel libraries are of obvious benefit in the development of large scale software projects, they complicate the BSP cost model by introducing subset synchronisation[2]. In chapter 5 we examined these complications and suggested methods that could be used in the performance prediction of Opal algorithms that contain multiple groups. It was shown how the effects due to deadlock, message queueing and message servicing have been reduced to a minimum by the careful design of the Opal language.

[1] It evaluates it for all values of the BSP parameters based upon the target architecture where there is a finite number of processors.

[2] The subset synchronisation of Opal groups is restricted and is designed to be used for library calls only, i.e it is not designed so that two sets of processes could be continuously executing, where one set performs I/O and the other set perform computation.

As previously stated, it is likely that the different groups of an Opal program are developed by different teams of programmers. For this reason it cannot be assumed that the user of a group has access to the group's implementation and it is the case that if they did have access then they would be breaking the principles of modular programming. Therefore the performance prediction of Opal programs must not rely on access to the source code of library groups. This is possible through the use of *superstep collapsing*, where the multiple supersteps of a subgroup can be considered as a single superstep of the parent group. Using this method the complexity of a group can be described by an equation of the form $C + H.g + L$ which can then be combined with the complexity of the parent superstep, therefore preventing the need to analyse the source code of the group itself.

In chapter 5 we introduced the concept of *parallel overlap*, where the processes of different groups share processors. This is viable since it is likely that only one group of processes will be executing at a time, i.e. as soon as a set of processors requests the services of another group they will become dormant and then the other groups processors will become active. The sharing of processors can be made deterministic through the use of the **on processor** statement that allows processes to be placed onto processors using an architecture independent method.

We then presented performance prediction techniques for the various types of library call mechanisms: all-to-all, one-to-all and any-to-any. These cost analysis techniques were based on a number of assumptions on the use of Opal groups, the primary one being that groups are properly used as a library mechanism and not as a way of having permanently active subsets of processes.

The assumptions of chapter 5 were shown to be valid through the example algorithms presented in chapter 6. In this chapter we presented a simple algorithm for the string edit problem and showed how its performance can be accurately predicted on two different architectures: a Silicon Graphics Power Challenge (SGI) and an IBM SP2. Using this example we also demonstrated how the **optBSPp** function could be used to automatically select the optimum number of processors to use for a given problem size.

The next example we described, a reduction and prefix sums group, illustrated the use of generic groups and the rendezvous mechanism. We demonstrated how the performance prediction of such an algorithm leads to accurate results. Automatic architecture adaption was illustrated through the use of the generic group parameters, demonstrating how an optimum t-ary tree can be selected.

We then presented an algorithm for performing data bucketing. This algorithm used the services of a multi prefix group whose design is very similar to the prefix group previously described. This illustrated how the use of Opal groups can reduce the amount of code that has to be written for a specific algorithm, therefore reducing the number of errors introduced. We also used this

example to demonstrate that the performance of Opal algorithms that use multiple groups can still be accurately predicted. This is an important result and shows that libraries can be added to the BSP model in a way that does not destroy the cost model.

The final two examples, iterative sorting and convex hull, illustrated how we can build solutions based on a hierarchical structure of groups. For example, the iterative sorting algorithm used the services of the bucketing group, which itself used the multi prefix group. Such hierarchies of groups are an important feature of the Opal language and would be the principal method of decomposition in large scale software projects.

For these examples to be produced and tested in a real environment, an Opal compiler was implemented. The compiler allowed the language constructs to be verified in a real implementation, ensuring that all the features of the language could be implemented efficiently. It also allowed the example Opal algorithms to be executed on real architectures, verifying that the performance prediction properties of the BSP model had not been destroyed. Further discussion on the compiler and its implementation can be found in appendix B.

7.2 Further Work

In this section we shall discuss further work that could be performed in the areas of dynamic data structures, buffering schemes and MIMD language constructs.

7.3 Dynamic Data Structures

Currently the Opal language does not make any provisions for dynamic data structures. While such data structures can always be replaced with static versions, dynamic data structures can often make an algorithm more concise and easier to follow. However, pointers must be introduced into a parallel language with care, ensuring that none of the design goals of the language are broken [Welch, 1991]. The issues regarding the integration of pointers into the Opal language are as follows:

- Should we distinguish, at the language level, between a pointer that refers to local memory and a pointer that refers to remote memory.
- How do we ensure that the programmer can visually determine when shared pointer dereferencing is being performed. If the dereferencing looks like a normal assignment then the

programmer may miss the communication in their cost analysis.

Part of the design rationale of the Opal language dictated that remote variable communication must be clearly visible to the programmer. The method of segregating shared variables into a process specification also allowed the compiler to pre-allocate shared memory. We can keep these two properties by distinguishing between local and shared pointers, therefore having two heaps: a local and a shared heap[3]. The compiler can then pre-allocate the static portion of the shared variables and use normal memory management techniques for the shared heap. It should be noted that when we refer to a shared heap we are describing a heap data structure that each process would contain, not a single heap that is shared across all processes.

Such mechanisms can be introduced into Opal using two new distinct types; **ref** and **ref shared**. A **ref** type is a local pointer and may only be declared in the body of a process, e.g. **ref integer** declares a pointer to an integer on the local heap. A **ref shared** type is a remote pointer and may be declared in a process specification or body. However, due to the Opal scoping rules only variables that are declared in a process's specification can be accessed by other processes. Therefore, although a **ref shared** variable that is declared in a body is a pointer to a remote memory location, the pointer variable itself cannot be accessed by other processes.

We also introduce two new functions; **new** and **free**. The **new** function accepts a type as a parameter and returns an allocated pointer to this type, e.g. **new** (**ref integer**) would supply a pointer to an integer on the local heap and **new** (**ref shared integer**) provides a pointer to an integer on the shared heap of the current process. The **free** function can then be used to free pointers when they are no longer required. It should be noted that we do not provide an *"address of"* function.

There must also be a method of dereferencing a pointer value. This could be performed by introducing two new operators; "$*$" and "$*.$". The "$*$" operation would only be capable of dereferencing local pointers, while the "$*.$" could only dereference global pointers.

We see that shared pointers could be implemented in a way in which the programmer can easily spot remote dereferencing, while still allowing the compiler to allocate shared memory as efficiently as possible.

However, there is still a minor problem. For example, consider the statement $*.x := proc.var$, where x is a locally declared **ref shared integer** and var is a remote variable on the process $proc$. Such a statement would require two communications and synchronisations; one to fetch the value of the remote variable and another to write the value of this variable to the remote

[3]It would be possible to use just one heap, but distinguishing between local and shared heap variables may be advantageous for some parallel architectures.

location referred to by x. The compiler must be able to detect such situations and reject them. This is already performed by the compiler, as it detects statements such as $p1.var := p2.var$ and flags them as a compile time error. In fact such statements look very similar to the above pointer dereferencing, due to the choice of the remote variable dereferencing operator "*.". Similarly expressions such as $y := *.proc.ptr$ would directly dereference the value of the remote pointer and must be detected by the compiler.

The following example illustrates how a process could traverse the elements of a shared link list, incrementing a value in each element of the list:

```
group main is
    type T is ref shared record
        x: integer; next: T;
    end record;
end main;

replicated main process worker (i: 1..BSPp) is
    head: main.T;
end worker;

replicated main process body worker (i: 1..BSPp) is
    ptr: main.T;
begin
    begin "get head" superstep
        ptr := worker(1).head;
    end superstep;
    while ptr != null loop
        begin "dereference" superstep
            localx := *.ptr->x;
        end superstep;
        begin "update and assign" superstep
            *.ptr->x := localx + 1; ptr := *.ptr->next;
        end superstep;
    end loop;
end worker;
```

While this is a very simple example it illustrates how shared pointers could be used. Such a mechanism would allow dynamic data structures to be created that span a number of processes.

7.4 Shared Variable Buffering

In section 4.3.4 we specified that writing into a remote variable can occur at any time during a superstep, but that the value written is the value at the point at which the assignment occurred. Similarly, reading from a remote location can occur at any point during the superstep and the local variable which is the target of the read can be updated at any point.

These decisions were made based on the fact that shared variable communication is very easy to distinguish at the source code level and so it is the responsibility of the programmer to ensure the correct buffering. However, such a scheme could lead to unnecessary synchronisations, e.g. it requires two supersteps for two processes to read and write the same shared variable.

In BSPlib [Hill *et al.*, 1997c] the programmer has direct access to the buffering scheme via the use of different communication primitives. Such buffering schemes could be introduced into the Opal language with the use of a **buffered** type modifier. This construct would only be valid in the specification of a process and would indicate the buffering to be performed on a per variable basis. Using this technique we can expand table 4.1 into that of table 7.1.

Assignment	$p.y$ declared **buffered**	$p.y$ not declared **buffered**
$x := p.y$	$p.y$ is read at the end of the superstep, **before** all writes. x is updated at the end of the superstep.	$p.y$ is read at any point during the superstep, x is updated at any point during the superstep.
$p.y := x$	$p.y$ is updated at the end of the superstep, **after** any reads. The value used in the write is the value of x at the point of assignment.	$p.y$ is updated at any point during the superstep. The value used in the write is the value of x at the point of assignment.
$p.y := p.x$	Invalid assignment that is detected by the compiler.	Invalid assignment that is detected by the compiler.

Table 7.1: Opal Variable Buffering Using the **buffered** Construct

Using the **buffered** mechanism of table 7.1 would allow the programmer to control the buffering scheme that is used. For example, using **buffered** variables would allow two processes to read and write the same variable in the same superstep without interference, but at the expense of using extra memory.

7.5 MIMD Constructs

Although the Opal language is MIMD in nature, it does not allow the same data structure to be distributed across replicated *and* non-replicated processes. For example, consider a replicated process that declares a shared data structure in its specification, representing the data that is distributed across all of the processes. It could be the case that the process at the beginning and end of this distributed data structure are to perform significantly different algorithms than the internal processes. Such an algorithm would be suitable for an Opal MIMD solution, using a replicated process and two non-replicated processes. However, this would not allow us to reference the shared data structure using a uniform dotted notation; i.e. if the internal replicated processes are called $iworker$ and the boundary processes $bworker$, then we would have to have an **if** statement in the $iworker$ processes to determine where to fetch the data from.

This can be resolved by having multiple replicated bodies export to the same replicated specification, as in the following example:

---

replicated main **process** worker (i:1..**BSPp**) **is**
    declare data structures
end worker;

replicated main **process body** worker (i:2..**BSPp** - 1) **is**
    body for internal processes
end worker;

replicated main **process body** worker (i:1..1) **is**
    body for the left boundary process
end worker;

replicated main **process body** worker (i:**BSPp**..**BSPp**) **is**
    body for the right boundary process
end worker;

In this example we have one replicated process specification, but three bodies that export to it. It would allow the boundary processes to execute a different algorithm, while still permitting dotted notation such as *worker(j-1).data* to be uniformly used. Such a mechanism would allow the MIMD style of programming to be used to its full benefit and could be a useful addition to the Opal language.

Appendix A

Sequential Language Constructs in Opal

A.1 Introduction

In this chapter we describe the sequential constructs of the Opal language. For a descriptions of the parallel constructs in Opal the reader is directed to chapter 4.

A.2 Data Types

The basic data types in Opal are shown in table A.1.

Type	Example Literal
integer	12
real, longreal	12.2
boolean	**true**
character	'c'
string	"a string"

Table A.1: Basic Opal Data types

The following sections describe how these types can be combined to form arrays, records, and user defined types. It should be noted that the Opal type system does not include mechanisms such as enumerations, sub-typing and opaque types. While these types are obviously useful, they have been successfully implemented in conventional sequential languages and could be easily incor-

porated into Opal. In order to simplify the implementation of a compiler, these type mechanisms have been excluded.

A.2.1 Arrays

The basic data types can be used to form arrays, in a similar manner to other imperative languages. An example of an Opal array definition is shown below:

---

array (1..20, 5..**BSPp** + 5) **of integer**;

This declares a two dimensional integer array with 20 elements in the first dimension, and $BSPp$ elements in the second. The bounds of the second dimension of this array use a non compile time constant, $BSPp$. This allows arrays to be declared whose size is not known at compile time, a requirement that was specified in section 2.4 on adaptive algorithms.

A.2.2 User Defined Types

A type declaration associates a name with a given type, allowing variables to be declared with a type that suggests their use. The following example declares a type T that is **integer**:

---

type T **is integer**;

The convention of giving an abstract data type the name T is used throughout Opal, and is a feature that has been borrowed from Modula-3 [Harbison, 1992]. Typically, T is used to represent the abstract data type of a package or group. For example, Opal includes a package *integers* that performs common operations on integers. In this package T is defined as above, and *integers.T* would be used to declare variables that this package operates upon. Obviously, we could perform this task without user defined types, but such declarations aid in program readability.

A.2.3 Records

Records provide a convenient method of grouping together related variables that may have different types. The following example defines a record type with fields x, y and $angle$, and uses this type to declare a variable $point$:

```
type T is record
            x, y: integer; angle: real;
        end record;

point: T;

begin
    point->x := 10; point->y := 20; point->angle := 0.0;
end;
```

This example also shows how to reference the fields of a record using the -> operator, a mechanism which should not be confused with pointer dereferencing in C.

A.3 Variable Declarations

The following code is an example of how variables are declared in Opal:

```
BufLen  : constant := 10;
flag    : boolean;
x, y    : integer;
buffer  : array (1..BufLen) of character;
```

Identifiers are case sensitive and can only consist of alphanumeric characters. It should be noted that constants do not need to have a type associated with them as this can be derived from the constant's value.

A.4 Type Checking and Conversions

Opal is a strongly typed language, using structural type equivalence. It is a compile time error to assign two variables of a different type. Array assignments are also checked to ensure that both arguments have the correct size and shape. The checking of array shapes will be performed at compile time if constant indexes are used, otherwise a runtime check will be inserted.

Opal has a number of built in type conversion functions as shown in table A.2. It should be noted that Opal does not employ type promotion, i.e. it is not possible to assign an **integer** to a **real** without the use of a type conversion function.

Type Conversion Function	Explanation
float (i)	Converts **integer** to **real**
round (r)	Converts **real** to the nearest **integer**
floor (r)	Converts **real** to the nearest **integer** less than r
ceil (r)	Converts **real** to the nearest **integer** greater than r
ord (c)	Converts **character** to **integer** ASCII
val (i)	Converts **integer** ASCII to **character**
string (x)	Converts any basic type to **string**

Table A.2: Type Conversion Functions

A.4.1 Boolean Operators

Table A.3 shows the Opal operators and their types, where t represents any of the basic data types. The boolean operators are short circuited, i.e. in an expression such as "a **and** b", b is only evaluated if a evaluates to **true**.

A.4.2 Built In Functions

The built in functions that are available in Opal are shown in table A.4. These functions can be used throughout an Opal program, including the index values of array declarations.

Operator	Explanation	Type
!=	Inequality	$t * t \rightarrow$ **boolean**
<, >	Less than, greater than	$t * t \rightarrow$ **boolean**
=	Equality	$t * t \rightarrow$ **boolean**
<=	Less than or equal	$t * t \rightarrow$ **boolean**
>=	Greater than or equal	$t * t \rightarrow$ **boolean**
not	Boolean negation	**boolean** $\rightarrow$ **boolean**
and	Boolean conjunction	**boolean** * **boolean** $\rightarrow$ **boolean**
or	Boolean disjunction	**boolean** * **boolean** $\rightarrow$ **boolean**
xor	Boolean exclusive or	**boolean** * **boolean** $\rightarrow$ **boolean**

Table A.3: Opal Operators

Operator	Explanation	Type
first	Index of first element	**array** $\rightarrow$ **integer**
last	Index of last element	**array** $\rightarrow$ **integer**
abs	Absolute value	**integer** $\rightarrow$ **integer** or **real** $\rightarrow$ **real**
mod	Modulo	**integer** * **integer** $\rightarrow$ **integer**
min	Minimum	**integer** * **integer** $\rightarrow$ **integer**
max	Maximum	**integer** * **integer** $\rightarrow$ **integer**
inc	Increment	**integer** $\rightarrow$ **integer**
dec	Decrement	**integer** $\rightarrow$ **integer**
sqrt	Square Root	**real** $\rightarrow$ **real**
log2	Logarithm Base 2	**real** $\rightarrow$ **real**
&	String Concatenation	**string** * **string** $\rightarrow$ **string**
+, -, *, /	Number operators	**integer** * **integer** $\rightarrow$ **integer**, **real** * **real** $\rightarrow$ **real**
**	Exponentiation	**integer** * **integer** $\rightarrow$ **integer**, **real** * **integer** $\rightarrow$ **real**

Table A.4: Built In Functions

A.5 Statements

The following sections describe the different types of statements that can be used in an Opal program. Statements must be terminated with a semicolon, and the empty statement is represented by the keyword **null**.

A.5.1 Assignment

The assignment operator in Opal is :=. Variables of any type can be assigned to each other, including records and arrays. In the case of arrays one may assign single elements, complete

arrays, or slices. The following Opal code gives some examples of assignments:

```
x := y;              -- Simple variable assignment.
a(1) := b(2,3);      -- Array element assignment.
a := c(1..10,4..6);  -- Array assignment where the c array is sliced.
```

The above code also serves as an example of how comments are introduced using --. Comments last until the end of the line on which they start and may appear anywhere in an Opal program.

An assignment statement is allowable if the expression on the left hand side represents a variable, and has the same type as the right hand side. In the case of arrays this type checking may require a runtime check that the arrays are of the same size and shape.

Slices provide a mechanism for referencing only parts of an array. For example, if we only wish to assign to the first half of an array then we can do so using an array slice. Such slices can appear on either one side or both sides of an assignment statement. Any dimension of an array may be sliced, and this can be intermixed with normal array indexing, e.g. c(1..10, 3, 4..6) is a valid slice. These slices are especially useful when multiple processes need to update certain portions of a remote array.

A.5.2 Conditional Statements

There are two conditional statements in Opal, these being **if** and **case**. The following code gives an example of how these statements are used:

```
if x = y then
    statements
elsif x = y + 1 then
    statements
else
    statements
end if;

case x of
    when 1 => statements
    when 3..4 | 7..9 => statements
    when others => statements
end case;
```

It is not required that an **if** statement contains an **else** part, and any number of **elsif** elements may be used. Similarly a **case** statement does not require a **when others** part. It should be noted that the expressions in the conditionals are only evaluated if they are required.

A.5.3 Loops

There are three types of loop statements as illustrated in the following Opal code fragment:

```
for i in 1..n loop        -- i is automatically declared.
    statements
end loop;

while not finished loop
    statements
end loop;

loop
    statements
end loop;
```

Any of these loops can contain the keyword **exit** or **exit when** *boolean_expression*. The **exit** statement causes the innermost enclosing loop to terminate and continue execution after the **end loop** statement. The **exit when** *boolean_expression* statement has the same affect, but only if

boolean_expression evaluates to true.

In the case of a **for** loop, the body will only be executed if the indicated range is non empty. A **for** loop can be reversed in the following manner:

```
for j in 1..n reverse loop
    statements
end loop;
```

It should also be noted that the index variable in a **for** loop is automatically declared.

A.5.4 Blocks

A block may be used at any point where a statement would be valid. Blocks allow variables to be declared with limited scope, an example of which is shown below:

```
declare
    minval, maxval : integer;
begin
    statements
end;
```

A block must contain at least one statement in its body, i.e. if an empty block is required then the **null** statement must be used.

A.6 Summary

Opal provides the familiar sequential constructs that are available in many programming languages. These constructs could obviously be extended to include other facilities such as objects and polymorphism, but this is beyond the scope of this thesis which concentrates on parallel issues.

Appendix B

Overview of the Opal Compiler

B.1 Introduction

In this chapter we shall describe the Opal compiler that was used to implement the example algorithms of chapter 6. Implementation of the compiler gave a useful insight into the efficiency of the language constructs and provided a concrete tool with which to evaluate algorithms and their predicted performance. However, the development of a compiler has been a time consuming task and it should be emphasised that our implementation is not the most efficient possible - it is designed more as a tool for the evaluation of the Opal language.

We start this chapter with an introduction to some of the standard libraries that are present in Opal. Such modules allow useful programs to be developed with the minimum of effort.

We will then give an overview of the Opal compiler, describing its design and the runtime system that it employs. Currently the underlying parallel runtime system consists of PVM [Geist *et al.*, 1993], which leads to portability across a number of architectures. When the compiler was begun the only BSP environment available was that of the Oxford BSP library [Miller, 1993], but this was still in its infancy and would have required adaption in order to support multiple groups. During the development of the compiler a number of other BSP libraries were introduced such as the Green BSP Library [Goudreau *et al.*, 1995] and BSPlib [Hill *et al.*, 1997c], but these still did not support multiple groups or subset synchronisation. For these reasons, and the fact that PVM offered a stable base available for a number of operating systems, it was decided to remain with PVM as the core parallel library.

We will next demonstrate how the Opal compiler is used and what options are available to the programmer. For example, although the compiler uses the *double superstep mode* for remote variable referencing, it is possible to investigate the properties of *single superstep mode*, via the use of a compiler flag.

In subsequent sections we discuss the tools that are available to aid in the performance prediction of Opal algorithms. Such tools resemble those of BSPlib, the Oxford BSP Toolset [Hill *et al.*, 1997c], and it is the case that the profiling output of Opal programs can be displayed using tools available with BSPlib.

Finally, we list the restrictions that the current compiler places on the Opal source code. These restrictions were made in order to simplify the development of a compiler, and are solely implementation issues that can be resolved in future versions.

B.2 Standard Libraries

The Opal language is based on a modular programming approach, where a number of parallel or sequential libraries are developed and then re-used in the development of future algorithms. For this to be successful a number a standard libraries should be made available to the programmer, some of which we shall discuss in this section.

B.2.1 SystemIO

The *SystemIO* group provides entry points for input and output to the system console. It must be implemented as a shared group since the console is a single resource that is used by all processes.

```
shared group SystemIO is
    -- Character input and output
    entry GetChar    (c: out character);
    entry PutChar    (c: in character);
    -- Integer intput and output
    entry OutInt     (x: in integer);
    entry InInt      (x: out integer);
    -- Floating point intput and output
    entry OutReal    (x: in real);
    entry InReal     (x: out real);
    -- Boolean input and output
```

```
    entry OutBool    (b: in boolean);
    entry InBool     (b: out boolean);
    -- String input and output
    entry OutString  (s: in string);
    entry InString   (s: out string);
    -- Output a carriage return
    entry NewLine    ();
end;
```

The most commonly used entry point is *SystemIO.OutString*, where the string can be the result of many concatenations, an example of which is shown below:

```
SystemIO.OutString ("Processor " & string (i) & " is ready\n");
```

B.2.2 XSystemIO

The *XSystemIO* group provides a mechanism for segregating the output of different processes. When the *XSystemIO* group is started a set of panned windows are created on the host console, where each window may be claimed by a process and used for output. This provides a useful debugging tool if many processes are outputting debug information and prevents the jumbled output that would occur if *SystemIO* were used.

```
shared group XSystemIO is
    entry CreateWindow (name: in string; tag: out integer);
    entry CloseWindow  (tag: in integer);
    entry OutChar      (tag: in integer; c: in character);
    entry OutInt       (tag: in integer; x: in integer);
    entry OutReal      (tag: in integer; x: in real);
    entry OutBool      (tag: in integer; b: in boolean);
    entry OutString    (tag: in integer; s: in string);
    entry NewLine      (tag: in integer);
end;
```

B.2.3 FileIO

The *FileIO* package provides access to the file system of the parallel architecture. Files can be opened and created using mechanisms similar to those in the C language.

```
package FileIO is
    type T is integer;
    procedure Open     (filename: in string; tag: out integer; mode: in string);
    procedure Close    (tag: in T);
    procedure Out      (tag: in T; text: in string);
    procedure OutInt   (tag: in T; n: in integer);
    procedure OutLine  (tag: in T; line: in string);
    procedure InLine   (tag: in T; line: out string);
    procedure InInt    (tag: in T; n: out integer);
    procedure InChar   (tag: in T; c: out character);
end;
```

B.2.4 String

This package provides functions that operate on strings. It should be noted that the built in function **string** can also be used to convert the basic data types to strings and is often more convenient to use than the equivalent function in the *string* package. However, such conversion functions are included in the *string* package for completeness.

```
package String is
    function Length    (s: in string) return integer;
    procedure Concat   (s1: in out string; s2: in string);
    procedure FromChar (chars: in array (<>) of character; s: out string);
    procedure FromInt  (i: in integer; s: out string);
    procedure FromBool (b: in boolean; s: out string);
    procedure ToChar   (s: in string; chars: out array (<>) of character);
    function ToInt     (s: in string) return integer;
    function ToBool    (s: in string) return boolean;
end;
```

B.2.5 Math

The *Math* package offers functions for the various math routines.

```
package Math is
    function exp    (x: in real) return real;
    function log    (x: in real) return real;
    function logt   (t: in integer; x: in real) return real;
    function ln     (x: in real) return real;
    function sin    (x: in real) return real;
    function cos    (x: in real) return real;
    function tan    (x: in real) return real;
    function asin   (x: in real) return real;
    function acos   (x: in real) return real;
    function atan   (x: in real) return real;
    function sinh   (x: in real) return real;
    function cosh   (x: in real) return real;
    function tanh   (x: in real) return real;
    function srand  (x: in integer);
    function rand   () return integer;
end;
```

B.2.6 Sequential Heap

The *SeqHeap* package provides a sequential implementation of the heap data structure, an abstract data type that is often required in the implementation of parallel algorithms.

```
package SeqHeap is
    procedure upheap (k: in integer; heap: in out array (<>,<>) of integer);
    procedure insert (v1, v2: in integer; heap: in out array (<>,<>) of integer;
                      N: in out integer);
    procedure downheap (k: in integer; heap: in out array (<>,<>) of integer;
                        N: in integer);
    procedure remove (heap: in out array (<>,<>) of integer; N: in out integer;
                      result: out array (1..2) of integer);
end SeqHeap;
```

B.2.7 Sequential Search

The *SeqSearch* package provides an implementation of the sequential binary search algorithm.

```
package SeqSearch is
    procedure search (x: in array (<>) of integer; y: in integer; pos: out integer);
end SeqSearch;
```

B.2.8 Sequential Sort

An operation that is required in many parallel algorithms is that of sorting a local data structure. To aid in the modular development of algorithms Opal provides a quick sort implementation in the *SeqSort* package.

```
package SeqSort is
    procedure doSort (data: in out array (<>) of integer; from: in integer; to: in integer);
end SeqSort;
```

B.2.9 Integers

The *integers* package provides functions that are applied to integers and is mainly used for instantiating generic packages that use integers as the generic type.

```
package integers is
    type T is integer;

    function cmp (a, b: in T) return integer;
    function sum (a, b: in T) return T;
    function sub  (a, b: in T) return T;
end integers;
```

B.2.10 Reals

The *reals* package is similar to the *integers* package except that it contains functions for use on floating point numbers. This package is mainly used for providing instantiations of generic groups.

```
package reals is
    type T is real;

    function cmp (a, b: in T) return integer;
    function sum (a, b: in T) return T;
    function sub  (a, b: in T) return T;
end reals;
```

B.3 Overview of the Opal Compiler

In this section we shall present an overview of the internals of the Opal compiler. We shall simplify the description by concentrating on general mechanisms rather than specific issues such as unconstrained arrays, separate compilation and array slices. While such constructs are difficult to integrate into the compiler they are simply an implementation problem.

The majority of the Opal compiler has been written using Modula-3 [Harbison, 1992], with a small amount of C [Kernighan and Ritchie, 1988] for the lower level parsing and lexical analysis routines. The parse tree is constructed from Modula-3 objects, a selection of which are shown in figure B.1.

Each object in figure B.1 has methods associated with it such as *TypeCheck* and *Compile*. To type check the whole program we simply invoke the *TypeCheck* method at the root of the tree, and this will propagate through each object. Using Modula-3 objects in this way allows new language constructs to be integrated into the compiler with the minimum of effort.

The compiler takes an Opal program and produces C source code with calls to PVM, which is then compiled using the system C compiler. This makes the compiler very portable since almost every architecture has a C compiler and PVM is available for a wide variety of machines. It is even the case that the compiler can cross-compile for other architectures, e.g. on a SunOS machine it is possible to produce target executables for use on a Silicon Graphics SGI or IBM SP2.

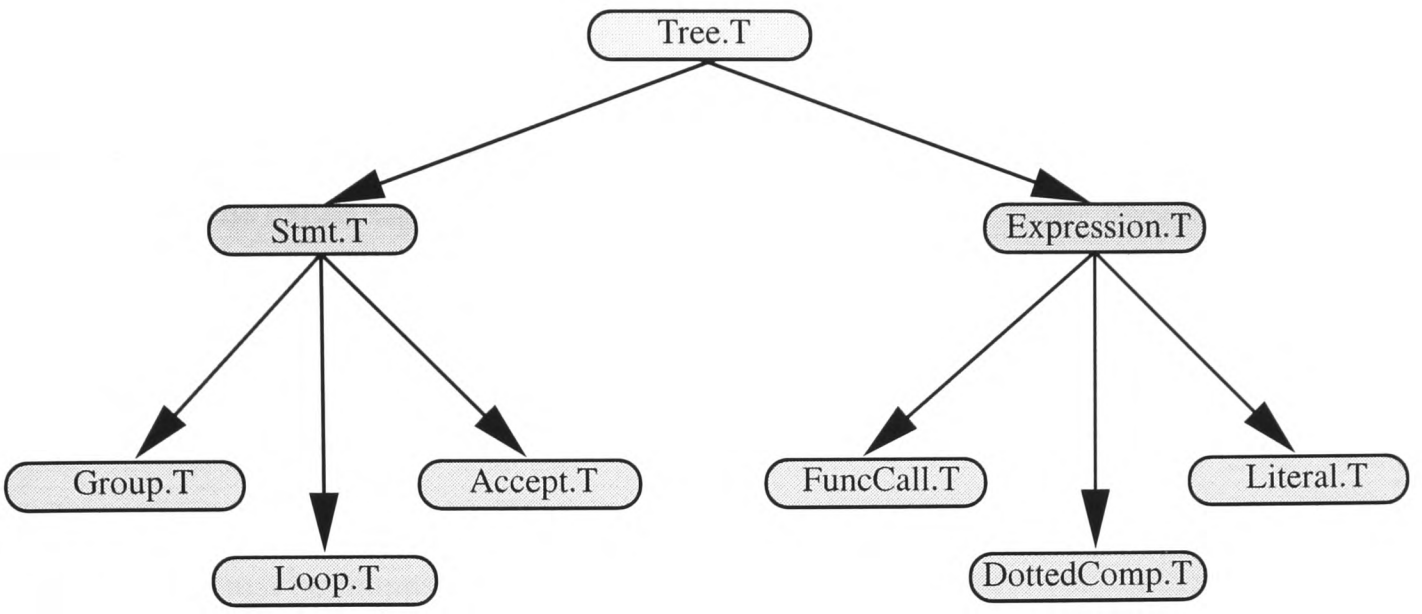


Figure B.1: Opal Compiler Objects

Shared memory is implemented through the use of message passing, where each shared memory element has a tag associated with it. If a process wishes to update a remote variable then it simply sends a message with the appropriate tag and supplies the data to be written. While this method is not ideal for real shared memory architectures, it does make the runtime system of the compiler identical for all architectures.

Similarly, rendezvous are implemented as messages that pass between two processes. However, care must be taken with rendezvous due to the accept statement which effectively multiplexes a number of message channels.

The compiler also implements a *usage* phase. During this phase information on the usage of expressions is gathered, relating statements to the expressions that they read and modify. Such information is used in single superstep mode in order to pre-fetch remote variables by moving the read to the beginning of the superstep[1]. This information could also be used in double superstep mode to identify variables that are used in two or more remote assignments in the same superstep.

Such a compiler offers the ability to experiment with the Opal language in a reasonably efficient environment. However, it is the case that runtime systems could be developed for specific architectures, leading to a more efficient implementation of Opal, but this is outside the scope of this thesis.

[1]The pre-fetching of variables can only be performed when the variable is uniquely identified, i.e. if the remote variable is an array element which is itself indexed by a variable then it cannot be pre-fetched.

B.4 Using the Compiler

The compiler uses a command line interface which resembles that of gcc. In its simplest form the compiler would be called with the Opal source files consisting of a single group, for example:

```
unix% Opal -O2 -o main main.grp worker.opl worker.spl
foreman.opl foreman.spl
```

This would produce an optimised[2] executable program called `main`. In order to execute this program PVM must be manually started, after which the user simply types `main`.

B.4.1 Subgroup Compilation

The Opal compiler allows for the separate compilation of multiple groups, as is normally the case for modular languages. However, it should be noted that if group A requires group B then group A must be compiled before B, thus ensuring that there are no cycles in the import structure of modules.

As described in previous chapters, there must be a group which is designated as being the main group. When this group is compiled it produces an executable that will automatically start all of the subgroups that it requires. For implementation reasons the compiler must be able to distinguish the main group from the subgroups. This is performed through the use of the -c flag, which informs the compiler that the group being compiled is a subgroup.

B.4.2 Generic Group Compilation

Before a generic group can be used in an instantiation it must be compiled with the -c flag. This produces a symbol table for use in the compilation of the generic group instantiation. The following compilation illustrates the two steps that are involved in creating a *prefix* group that is an instantiation of the generic *gPrefix* interface.

```
unix% Opal -O2 -c gPrefix.grp gPSlave.spl gPSlave.opl
unix% Opal -O2 -o prefix prefix.grp gPSlave.spl gPSlave.opl
```

[2] Optimised in the sequential sense.

In this example the generic group *gPrefix* is implemented by the *gPSlave* replicated process. In the first compilation we create a symbol table for the *gPrefix* group, which is then used in the second compilation to form the instantiated group *prefix*.

B.4.3 Standard Compiler Options

The full set of compiler flags are listed in table B.1. During the prototyping of the compiler a number of issues described in the design rationale were investigated. Such issues included single versus double superstep reads and the merging of h-relations. These configurations are still supported by the compiler, and can be activated using the options in table B.1, allowing these issues to be further investigated.

Option	Description
-c	Indicates that this is not the main group.
-macks	Merge write acknowledgements when using non merging h-relations.
-memchk	Use error checking version of malloc for the runtime system.
-memstats	Produce statistics on memory usage of the runtime system.
-noblock	Issue h-relations immediately and do not merge.
-norcheck	Do not generate code for array bound and shape checking.
-o <filename>	Output filename for the executable.
-p <version>	PVM version.
-prof	Generate profiling data.
-pvmdr	Use PVM direct routing.
-single	Perform single superstep reads.
-L <path>	Add <path> to the list of search directories for group interfaces.
-O	Perform basic sequential optimisations.
-O2	Perform advanced sequential optimisations.

Table B.1: Opal Compiler Options

B.5 Performance Prediction Tools

The Opal compiler allows the gathering of profile data through the use of the -prof flag. If a group is compiled with this flag then when it is executed two files are created; *<group name>.bsp* and *<group name>.bsplib*.

The *.bsplib* file contains profiling data in a format that is compatible with the BSPlib tools [Hill *et al.*, 1997c]. This allows the computation and communication structure of the group execution to

be investigated and displayed using the methods described in [Hill *et al.*, 1996].

The *.bsp* file contains profiling data that can be examined using the `opprof` tool that is supplied with the compiler. This tool analyses the profiling data and determines the total run time and the total time spent communicating, performing local computation and synchronising.

The `opprof` tool also allows the profiling data to be used to predict the performance of the same algorithm but on a different architecture, as do the BSPlib tools. The main difference between this tool and those of BSPlib is that it records the name of the superstep, thus allowing individual supersteps to be analysed. This is required when we are predicting the performance of algorithms that perform rendezvous.

B.6 Compiler Restrictions

All of the Opal language constructs have been implemented in the current compiler, however there are some restrictions that should be noted:

- Array slices cannot be used as actual parameters to procedure or rendezvous calls. The compiler will detect this as an error condition.
- If an array is used in a remote assignment then its bounds and shape are not checked until after the assignment has occurred. This could corrupt the memory of the target process, but this situation will still be detected by the source process.

These are minor restrictions that should not affect the development of Opal algorithms and are situations that are always detected by the compiler.

B.7 Summary

The Opal compiler that has been developed has allowed concrete example algorithms to be produced and their predicted performance analysed. While the language could have been described without developing a compiler it would have been harder to demonstrate that it meets the design goals, e.g. that the group structure still conforms to the BSP cost model.

However, compiler implementations are extremely time consuming, especially when modular environments are involved and so the current implementation is not designed to be the most efficient

possible. With this in mind the compiler implementation was developed in an object oriented style that allows new language constructs to be added with the minimum of effort, therefore allowing them to be analysed in real environments.

Such a compiler implementation has also allowed us to verify that the language constructs are implementable with reasonable efficiency. It is often the case that languages are designed that make compiler production an extremely difficult task, e.g. the array alignment features of High Performance FORTRAN [Forum, 1993].

The standard libraries that are supplied with Opal provide the building blocks for parallel algorithms, reducing the amount of code that has to be developed for a given application. Such libraries also allow console and file access in a way that is independent of the parallel architecture.

Of course, these libraries can be supplemented by groups that are developed by programming teams, therefore producing an environment where applications can be built by the composing library programs. Candidates for such library programs include the parallel prefix sums, reduction, bucketing and sorting generic groups that were described in chapter 6. Such groups could be considered as an *extended parallel group library* and would provide the building blocks for many BSP algorithms.

Due to the investigative nature of this thesis, the Opal compiler has options for supporting a number of features that were discussed in the design rationale. The initial reason for this was so that the different options could be investigated and an informed decision made on the best choice, e.g. single versus double superstep mode. However, to allow further investigation of these options they have not been removed from the compiler and can be turned on through the use of compiler flags.

The profiling support of the Opal compiler allows the execution of BSP algorithms to be accurately investigated. The profiling data produced by the execution of an Opal algorithm has been made compatible with that of BSPlib, therefore allowing existing tools to be re-used. However, the naming of Opal supersteps provides a useful mechanism of identifying individual areas of an algorithm, and is exploited through the use of a new `opprof` tool.

Appendix C

Prefix Group Source Code

C.1 Introduction

In this chapter we give the complete source code listing for the generic prefix group as described in section 6.3. The implementation of this group consists of three files:

1. The generic group interface held in the file gPrefix.grp.
2. The replicated specification of the process that implements the prefix group, held in the file gPSlave.spl.
3. The replicated body of the process that implements the prefix group, held in the file gP-Slave.opl.

The following sections give the Opal source code that is contained in each of these files.

C.2 gPrefix.grp

```
-- The function cmp should return 0 if a = b, +'ve if a < b, -'ve if a > b

generic group gPrefix (   nprocs   : in integer;
                          t        : in integer;
                          T        : type;
                          cmp      : in function (a, b: in T) return integer;
                          sum      : in function (a, b: in T) return T     ) is

    entry prefix (1..nprocs) (data: in out array (<>) of T);

    entry reduction (1..nprocs) (data: in array (<>) of T; value: out T;
                                 proc: out integer; indx: out integer);
end gPrefix;
```

C.3 gPSlave.spl

```
with gPrefix;

replicated gPrefix process gPSlave (i: 1.. gPrefix.nprocs) is

    value : gPrefix.T;
    children: array (1..gPrefix.nprocs, 1..gPrefix.t) of gPrefix.T;

    minind, minproc : integer;

end gPSlave;
```

C.4 gPSlave.opl

```
--*************************************************************************
--*
--* t-ary algorithm for computing prefix sums. Algorithm is as below:
--*
--* Forward traversal, j varies from 1 to logbase t of nprocs:
--*
--*       For processes whose i is an exact power of j to the t:
--*            Take the value of your t children, perform gPrefix.func on them and store in
--*            #gPSlave.value. Store the value of your k-th child in #gPSlave.children (j,k+1).
--*            At this point in the algorithm #gPSlave.children (j,0) contains no useful data.
--*
--* Reverse traversal, j varies from logbase t of nprocs down to 1:
--*
--*       For processes whose i is an exact power of j to the t:
--*
--*            Three cases (in this order, as cases are not disjoint):
--*
--*                 1) You are a special node at this level.
--*                 A special node is such that i = j to the power t. Special nodes take their left
--*                 but one #gPSlave.children and put it in the #gPSlave.children (j,1). Then
--*                 starting at the 3rd child they accumulate the children from left to right.
--*
--*                 2) You are your own parent.
--*                 Take your parents k-th child value at the next higher level, where k is your
--*                 child index with respect to your parent. Put this value in your left
--*                 #gPSlave.children, and then accumulate your children from left to write.
--*
--*                 3) Your parent is not yourself.
--*                 Take your parents k-th gPSlave.children value at the next higher level (k
--*                 defined as above) and put it in your own #gPSlave.children (j,1) value. Then
--*                 accumulate your #gPSlave.children values from left to right.
--*
--* Final superstep:
--*
--*       If gPrefix.t is a divisor of i then add your #gPSlave.value into your first data value,
--*       and then accumulate this from left to right.
--*       If gPrefix.t is not an exact divisor of i then you need to get the k-th value of your
--*       parents gPSlave.children, k defined as above, and then accumulate this from left to
--*       right in your own data.
```

```
--*       Note that processor 1 has no correction to apply, and simply computes the prefix sum
--*       on its own local data.
--*
--*************************************************************************

with gPrefix, gPSlave, Math;
replicated gPrefix process body gPSlave (i:1..gPrefix.nprocs) is

    on processor i-1;

    twojm1, twoj, twojp1, logtBSPp, tPlogtBSPp   : integer;
    neighbourVal                                 : gPrefix.T;
    neighbourValArray                            : array (1..gPrefix.t) of gPrefix.T;
    neighbourIndArray                            : array (1..gPrefix.t) of integer;
    neighbourInd, neighbourNum                   : integer;
    numchildren                                  : array (1.. gPrefix.nprocs) of integer;
    childInd, indexToCompute, startk             : integer;
    diffParent                                   : boolean;

    procedure initialise () is
    begin
        logtBSPp := floor (Math.logt (gPrefix.t, float (gPrefix.nprocs)));
        tPlogtBSPp := pow (gPrefix.t, logtBSPp);
        if tPlogtBSPp ! = gPrefix.nprocs then
            --This part allows for gPrefix.nprocs not being an exact power of gPrefix.t
            inc (logtBSPp);
            tPlogtBSPp := pow (gPrefix.t, logtBSPp);
        end if;
        twojm1 := 1; twoj := gPrefix.t; twojp1 := gPrefix.t * twoj;
    end initialise;

    superstep procedure doprefix
                (data: in out array (<>) of gPrefix.T;
                 sumf: in function (a, b: in gPrefix.T) return gPrefix.T) is
    begin
        begin "compute local prefix (bucks)" superstep

            --Set #gPSlave.value to the prefix sum of the data we have.
            #gPSlave.value := data (first (data));
            for k in first (data) + 1..last (data) loop
                #gPSlave.value := sumf (#gPSlave.value, data (k));
            end loop;

        end superstep;
```

```
--Now go from the leaves to the root of the t-ary tree. At the end of this for loop each
--process which is an exact power of gPrefix.t will have computed the correct offset.
--We need to save the partial results for each j in a structure called "children". This
--is used for the reverse traversal of the tree, so that processes that are not an exact
--power of gPrefix.t can compute the correct result.

for j in 1 .. logtBSPp loop

    begin "pfix bucks (data) " & string (j) superstep

        neighbourNum := 0;

        if i mod twoj = 0 or i = gPrefix.nprocs then

            neighbourNum := gPrefix.t - 1;

            if i mod twoj = 0 then
                neighbourInd := i - twoj;
            else
                --Pretend we are the next multiple of twoj greater than i.
                --Next multiple of twoj greater than i is twoj - i mod twoj + i.

                neighbourInd := (twoj - i mod twoj + i) - twoj;

                --Now pretend we are increasing the neighbour index, looking at
                --each neighbour. We do this just to find out how many
                --neighbours we really have.
                neighbourNum := 0;
                while neighbourInd + (neighbourNum + 1) * twojm1 < i loop
                    inc (neighbourNum);
                end loop;
            end if;

            numchildren (j) := neighbourNum;

            for k in 1..neighbourNum loop
                neighbourInd := neighbourInd + twojm1;
                #gPSlave.children (j,k+1) := gPSlave (neighbourInd).value;
            end loop;
        end if;

end superstep;
```

```
        begin "pfix bucks (comp) " & string (j) superstep

            for k in 1..neighbourNum loop
                #gPSlave.value := sumf (#gPSlave.value, #gPSlave.children (j, k + 1));
            end loop;

            twojm1 := twoj; twoj := twojp1; twojp1 := twojp1 * gPrefix.t;

        end superstep;

    end loop;

    --Now traverse the t-ary tree from the root to the leaves, computing a correction factor
    --at each node.

    for j in reverse 1..logtBSPp loop

        begin "rev pfix bucks (data) " & string (j) superstep

            twojp1 := twoj; twoj := twojm1; twojm1 := twoj / gPrefix.t;
            startk := gPrefix.t + 1;
            diffParent := false;

            if i mod twoj = 0 or i = gPrefix.nprocs then

                if i mod twojp1 = 0 or i = gPrefix.nprocs then
                    neighbourInd := i;
                    childInd := numchildren (j+1) + 1;
                else
                    childInd := (i / twoj) mod gPrefix.t;
                    neighbourInd := i + (gPrefix.t - childInd) * twoj;
                    neighbourInd := min (neighbourInd, gPrefix.nprocs);
                end if;

                if i = neighbourInd or i = twoj then

                    if i = neighbourInd and j != logtBSPp then
                        --Processor is its own parent.
                        #gPSlave.children (j,1) := #gPSlave.children (j+1,childInd);
                        startk := 2;
                    else
                        --Processor is special.
                        #gPSlave.children (j, 1) := #gPSlave.children (j, 2);
                        startk := 3;
```

```
            end if;

        elsif i != neighbourInd then

            --Processor has a different parent.
            #gPSlave.children (j,1)
                := gPSlave (neighbourInd).children (j+1,childInd);
            diffParent := true;
            startk := 2;

        end if;

    end if;

end superstep;

begin "rev pfix bucks (comp) " & string (j) superstep

    if diffParent then
        #gPSlave.value := sumf (#gPSlave.value, #gPSlave.children (j, 1));
        end if;
        for k in startk..gPrefix.t loop
            #gPSlave.children (j, k) := sumf (#gPSlave.children (j, k),
                                              #gPSlave.children (j, k - 1));
        end loop;

    end superstep;

end loop;

--Perform the final correction. This takes the value that has been computed on the
--forward and reverse traversal of the tree and applies it to the local data.

begin "Final bucks correction (data)" superstep

    childInd := i mod gPrefix.t;
    indexToCompute := -1;

    if childInd != 0 and i != 1 and i != gPrefix.nprocs then
        neighbourInd := min (i + (gPrefix.t - childInd), gPrefix.nprocs);
        #gPSlave.children (1,1) := gPSlave (neighbourInd).children (1, childInd);
        indexToCompute := 1;
    elsif (childInd = 0 and i != 1) or i = gPrefix.nprocs then
        if childInd = 0 then childInd := gPrefix.t; end if;
```

```
            indexToCompute := childInd;
        end if;

    end superstep;

    begin "Final bucks correction (comp)" superstep

        if indexToCompute ~= -1 then
            data (first (data)) := sumf (data (first (data)),
                                         #gPSlave.children (1, indexToCompute));
        end if;

        for k in first (data) + 1..last (data) loop
            data (k) := sumf (data (k), data (k-1));
        end loop;

    end superstep;

end doprefix;

superstep procedure doreduct ( data: in array (<>) of gPrefix.T; value: out gPrefix.T;
                               proc: out integer; indx: out integer;
                               cmpf: in function (a, b: in gPrefix.T) return integer) is
begin
    begin superstep

        --Set #gPSlave.value to the minimum value of the local data we have.

        #gPSlave.value := data (first (data));
        #gPSlave.minind := 1;
        #gPSlave.minproc := i;

        for l in first (data) + 1..last (data) loop
            if cmpf (data (l), #gPSlave.value) > 0 then
                #gPSlave.value := data (l);
                #gPSlave.minind := l;
            end if;
        end loop;

    end superstep;

    --Now go from the leaves to the root of the t-ary tree. At the end of this for loop the
    --process with index gPrefix.nprocs will contain the minimum value.
```

```
for j in 1 .. logtBSPp loop

    begin "pfix bucks (data) " & string (j) superstep

        neighbourNum := 0;

        if i mod twoj = 0 or (i = gPrefix.nprocs) then

            neighbourNum := gPrefix.t - 1;

            if i mod twoj = 0 then
                neighbourInd := i - twojm1;
            else
                --Pretend we are the next multiple of twoj greater than i. Next
                --multiple of twoj greater than i is i + twoj - i mod twoj.
                neighbourInd := i + twoj - i mod twoj - twojm1;
                --Now decrease the neighbour index until we find a neighbour
                --whose index is less than our own.
                while neighbourInd >= i loop
                    dec (neighbourNum);
                    neighbourInd := neighbourInd - twojm1;
                end loop;
            end if;

            for k in 1..neighbourNum loop
                --Find the minimum value of our neighbours, and correct our
                --own minimum value if required.
                neighbourValArray (k) := gPSlave (neighbourInd).value;
                neighbourIndArray (k) := neighbourInd;
                neighbourInd := neighbourInd - twojm1;
            end loop;

        end if;

    end superstep;

    declare
        procWithMin: integer;
    begin "pfix bucks (comp) " & string (j) superstep

        procWithMin := 0;

        for k in 1..neighbourNum loop
```

```
                if cmpf (neighbourValArray (k), #gPSlave.value) > 0 then
                    procWithMin := k;
                    #gPSlave.value := neighbourValArray (k);
                end if;
            end loop;

            if procWithMin ! = 0 then
                #gPSlave.minind := gPSlave (neighbourIndArray (procWithMin)).minind;
                #gPSlave.minproc := gPSlave (neighbourIndArray (procWithMin)).minproc;
            end if;

            twojm1 := twoj; twoj := twojp1; twojp1 := twojp1 * gPrefix.t;

        end superstep;

    end loop;

    --Now traverse the t-ary tree from the root to the leaves, copying the minimum value
    --down the tree. This is effectively a broadcast of the minimum value.
    for j in reverse 1..logtBSPp loop

        begin "rev pfix bucks " & string (j) superstep

            twojp1 := twoj; twoj := twojm1; twojm1 := twoj / gPrefix.t;

            if i mod twoj = 0 or (i = gPrefix.nprocs) then
                neighbourNum := gPrefix.t - 1;
                if i mod twoj = 0 then
                    neighbourInd := i - twojm1;
                else
                    --Pretend we are the next multiple of twoj greater than i. Next
                    --multiple of twoj greater than i is i + twoj - i mod twoj.
                    neighbourInd := i + twoj - i mod twoj - twojm1;
                    --Now decrease the neighbour index until we find a neighbour
                    --whose index is less than our own.
                    while neighbourInd >= i loop
                        dec (neighbourNum);
                        neighbourInd := neighbourInd - twojm1;
                    end loop;
                end if;

                for k in 1..neighbourNum loop
                    --Copy the minimum value to our children.
```

```
                gPSlave (neighbourInd).value := #gPSlave.value;
                gPSlave (neighbourInd).minind := #gPSlave.minind;
                gPSlave (neighbourInd).minproc := #gPSlave.minproc;
                neighbourInd := neighbourInd - twojm1;
            end loop;

        end if;

    end superstep;

end loop;

--Finally set the parameters to the minimum value found, the processor this minimum
--value is located on, and its index.

begin superstep
    proc := #gPSlave.minproc;
    indx := #gPSlave.minind;
    value := #gPSlave.value;
end superstep;
```

```
    end doreduct;


begin -- The following code is executed when gPSlave.opl is started
    initialise ();
    loop
        select
            accept prefix (i) (data: in out array (<>) of gPrefix.T) do
                doprefix (data, gPrefix.sum);
            end accept;
        or
            accept reduction (i) ( data: in array (<>) of gPrefix.T; value: out gPrefix.T;
                                   proc: out integer; index: out integer) do
                doreduct (data, value, proc, index, gPrefix.cmp);
            end accept;
        or
            terminate;
        end select;
    end loop;
end gPSlave;
```

Appendix D

Experimental Results

The BSP model consists of a number of parameters that are used in the cost analysis of algorithms. In this section we shall discuss the measurement of the communication parameters, g and l, on two parallel architectures: an IBM SP2 and a Silicon Graphics Power Challenge.

D.1 BSP Parameters

The parameters of a BSP machine are not only dependent on the architecture being used, but also on the runtime system. For this reason we must benchmark a parallel architecture using the same runtime system that normal algorithms would use, i.e. using an Opal program.

The Opal program we used to measure g and l would time the communication of a number of h-relations, where h was taken from the set $\{8, 16, 32, .., 32768, 65536\}$. For each h-relation we measured four different methods of forming such a communication, where each process would send or receive at most x messages of size $\frac{h}{x}$, for $x = 2^i . \forall i . 0 \leq i \leq 3$. A mixture of balanced and unbalanced h-relations were used, where a balanced h-relation is one in which each processor sends and receives *exactly* h words. While performing such h-relations we also measured the synchronisation performance, l.

Figure D.1 shows the results of running the benchmark program on a four processor Silicon Graphics Power Challenge (SGI)[1]. As described in section 2.2.3, we expect that for a fixed x the value of g should vary according to the following equation:

[1]The benchmark program was compiled to use merging h-relations, as were the example algorithms of chapter 6.

$$g(h) = \left(\frac{N_{\frac{1}{2}}}{h} + 1 \right) g_\infty \tag{D.1}$$

Figure D.1 demonstrates that the results are as expected, i.e. for small values of h the $N_{\frac{1}{2}}$ parameters dominates the value of g, but for large messages g asymptotically reaches the value of g_∞.

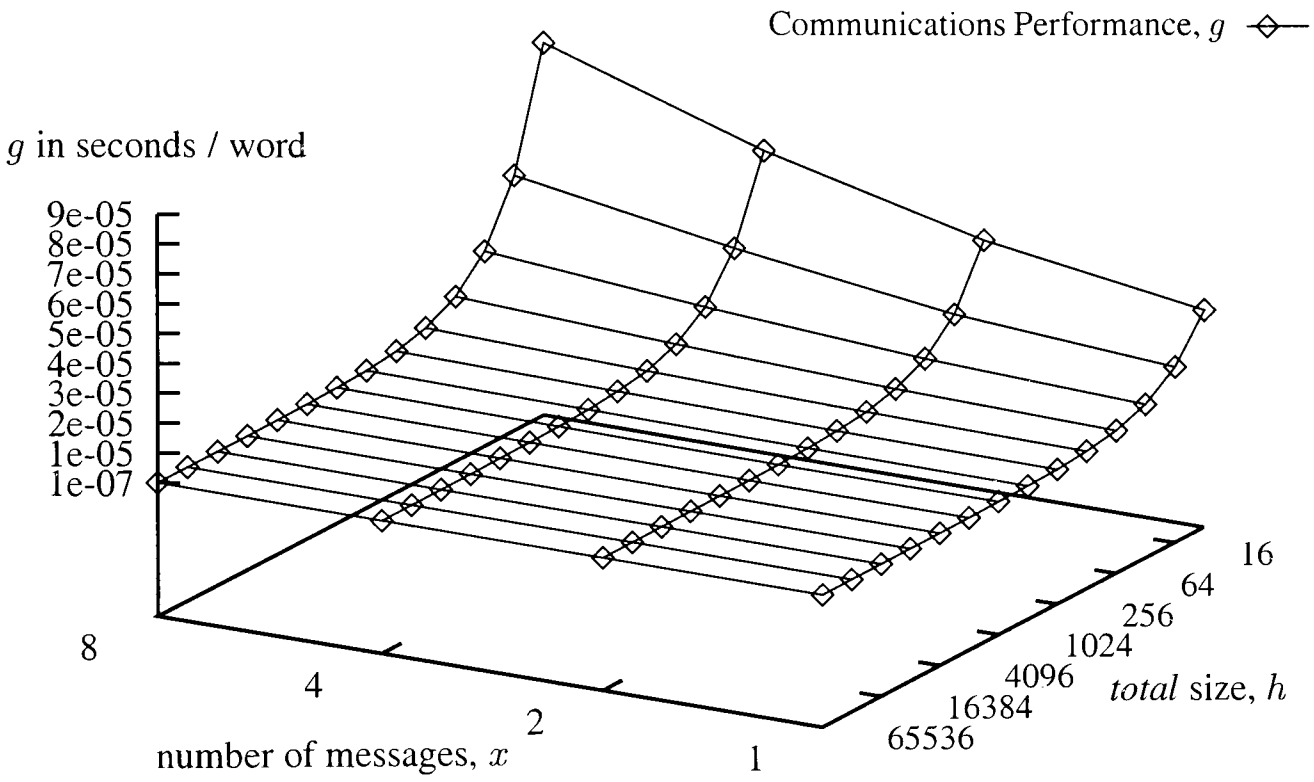


Figure D.1: SGI Communications Performance, g

This graph also demonstrates that even if messages are merged into one communication then for small h the value of g can vary depending on x. However, for large messages there is little difference in sending $\frac{h}{x}$ messages of size x, or x messages of size $\frac{h}{x}$.

D.1.1 Architectures Benchmarked

IBM SP2

The IBM SP2 consists of six RS6000 Power 2 processors interconnected by a proprietary cross-bar. Each processor has its own local memory and cannot directly access the memory of other processors.

Silicon Graphics Power Challenge (SGI)

The Silicon Graphics Power Challenge consists of four TFP processors. This is a shared memory machine, with each processor performing cache snooping on the address and data buses.

D.1.2 Results

Tables D.1 and D.2 show the results of running the benchmark program for a range of processors on both the SGI and SP2 architectures. The values for $N_{\frac{1}{2}}$ and g_∞ were obtained by fitting four curves of the form of equation D.1, with one curve for each value of x, and then taking the average.

BSPp	s	$N_{\frac{1}{2}}$	g_∞		l	
	MFlop	words	Flop / word	μs / word	Flop	μs
1	55	-	-	-	1155	21
2		2.414e+3	6.732	0.1224	7326	133
3		3.059e+3	6.716	0.1221	12199	221
4		2.179e+3	10.07	0.1831	15834	287

Table D.1: $N_{\frac{1}{2}}$, l and g_∞ for the SGI

The value of l was obtained by measuring the synchronisation time for each superstep in the benchmark program and then averaging over the number of supersteps. The value of l for one processor represents the overhead in calling the synchronisation routine, even though there are no processes to synchronise.

BSPp	s	$N_{\frac{1}{2}}$	g_∞		l	
	MFlop	words	Flop / word	μs / word	Flop	μs
1		-	-	-	312	12
2		8.463e+2	10.494	0.4036	3673	141
3	26	9.756e+2	10.249	0.3942	5327	204
4		1.119e+3	9.9996	0.3846	6575	252
5		1.378e+3	9.2092	0.3542	8034	309

Table D.2: $N_{\frac{1}{2}}$, l and g_∞ for the SP2

D.2 Summary

In this section we have demonstrated that the value of g can be accurately modelled by an equation of the form $\left(\frac{N_{\frac{1}{2}}}{h}+1\right).g_\infty$, where h is the h-relation being implemented [Miller, 1994]. Using this result we have obtained values for $N_{\frac{1}{2}}$ and g_∞ on the Silicon Graphics SGI and IBM SP2 architectures, these values being those that were used in the performance prediction of chapter 6.

Bibliography

[Abolhassan *et al.*, 1991] F Abolhassan, J Keller, and W J Paul. On the cost-effectiveness of PRAMs. In *Proc.* 3^{rd} *IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1991.

[Abolhassan *et al.*, 1993] F Abolhassan, R Drefenstedt, J Keller, W J Paul, and D Scheerer. On the Physical Design of PRAMs. *Computer Journal*, 36(8):756–762, December 1993.

[Agerwala *et al.*, 1995] T Agerwala, J L Martin, J H Mirze, D C Sadler, D M Dias, and M Snir. SP2 System Architecture. In *IBM Systems Journal*, volume 34(2), pages 152–184, 1995.

[Aggarwal *et al.*, 1989] A Aggarwal, A K Chandra, and M Snir. On Communication Latency in PRAM Computations. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[Aggarwal *et al.*, 1990] A Aggarwal, A K Chandra, and M Snir. Communication Complexity of PRAMs. *Theoretical Computre Science*, 71:3–28, 1990.

[Alexandrov *et al.*, 1995] A Alexandrov, M F Ionescu, K E Schauser, and C Scheiman. LogGP: Incorporating Long Messages into the LogP Model. In *Proc. of the* 7^{th} *Annual Symposium on Parallel Algorithms and Architectures (SPAA'95)*, July 1995.

[Bach *et al.*, 1997] P Bach, M Braun, A Formella, J Friedrich, Th Grün, and C Lichtenau. Building the 4 Processor SB-PRAM Prototype. In *Proc. of the Hawaii* 30^{th} *International Symposium on System Science, (HICSS-30)*, volume 5, pages 14–23, January 1997.

[Barnes, 1989] J Barnes. *Programming in Ada.* Addison-Wesley, 1989.

[Barrett, 1992] G Barrett. The Occam 3 Reference Manual. Technical report, INMOS, March 1992. Draft Copy, http://www.hensa.ac.uk/parallel/occam/documentation/manual3.ps.Z.

[Bäumker *et al.*, 1995] A Bäumker, W Dittrich, F Meyer auf der Heide, and I Rieping. Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP Model. In *Proc. of the European Symposium on Algorithms*, pages 17–30, 1995.

[Bäumker *et al.*, 1996] A Bäumker, W Dittrich, F Meyer auf der Heide, and I Rieping. Realistic Parallel Algorithms: Priority Queue Operations and Selection for the BSP* Model. In *Proc. of the* 2^{nd} *International Euro-Par Conference (Euro-Par'96)*, volume II, pages 369–376, August 1996.

[Bilardi *et al.*, 1996] G Bilardi, K T Herley, and A Pietracaprina. BSP vs LogP. In *Proc. of the* 8^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 25–32, 1996.

[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.*, 1995] T Cheatham, A Fahmy, D Stefanescu, and L Valiant. Bulk Synchronous Parallel Computing - A Paradigm for Transportable Software. In *Proc.* 28^{th} *Hawaii International Conference on System Science*, January 1995.

[Cheng, 1993] D Y Cheng. A Survey of Parallel Programming Languages and Tools. Technical Report RND-93-005, NASA Ames Research Center, Moffet Field, CA 94035-1000, March 1993.

[Cole, 1989] M Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.

[Culler *et al.*, 1993a] D Culler, A Dusseau, S C Goldstein, A Krishnamurthy, S Lumetta, T von Eicken, and K Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing*, November 1993.

[Culler *et al.*, 1993b] D Culler, R Karp, D Patterson, A Sahay, K E Schauser, E Santos, R Subramonian, and T von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc.* 4^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1993.

[Culler *et al.*, 1993c] D Culler, R Martin, A Dusseau, and K E Schauser. Fast Parallel Sorting: from LogP to Split-C. In *Workshop on Portability and Performance for Parallel Processing*, 1993.

[Darlington *et al.*, 1995] J Darlington, Y K Guo, H W To, and Y Jing. Skeletons for Structured Parallel Composition. In *Proceedings of the* 15^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[de la Torre and Kruskal, 1991] P de la Torre and C P Kruskal. Towards a Single Model of Efficient Computation in Real Parallel Machines. In *Proceedings of Parallel Architectures and Languages Europe, (PARLE)*, pages 6–24, 1991.

[Dehne *et al.*, 1993] F Dehne, A Fabri, and A Rau-Chaplin. Scalable Parallel Computational Geometry for Coarse Grain Multicomputers. In *Proc. of the ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[Fortune and Wyllie, 1978] S Fortune and J Wyllie. Parallelism in random access machines. In *Proc. of the* 10^{th} *Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[Forum, 1993] The High Performance Fortran Forum. High Performance Fortran Language Specification. Technical report, Rice University, May 1993.

[Geist *et al.*, 1993] A Geist, A Beguelin, J Donyarra, W Jiang, R Manchek, and V Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory, 1993.

[Gerbessiotis and Siniolakis, 1996a] A V Gerbessiotis and C J Siniolakis. Deterministic Sorting and Randomized Median Finding on the BSP model. In *Proc.* 8^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, June 1996.

[Gerbessiotis and Siniolakis, 1996b] A V Gerbessiotis and C J Siniolakis. Primitive Operations on the BSP Model. Technical Report PRG-TR-23-96, Oxford University Computing Laboratory, 1996.

[Gerbessiotis and Valiant, 1992] A V Gerbessiotis and L G Valiant. Direct Bulk-Synchronous Parallel Algorithms. Technical Report TR-10-92, Center for Research in Computing Technology, Harvard University, 1992.

[Gibbons and Spirakis, 1993] A Gibbons and P Spirakis, editors. *Lectures on parallel computation*. Cambridge University Press, 1993.

[Goudreau *et al.*, 1994] M W Goudreau, S B Rao, and T Tsantilas. A Study of the BSP Model: Algorithms and Implementation. Technical Report CS-TR-94-04, University of Central Florida, Orlando, FL 32816, 1994.

[Goudreau *et al.*, 1995] M W Goudreau, K Lang, S B Rao, and T Tsantilas. The Green BSP Library. Technical Report CS-TR-95-11, Department of Computer Science, University of Central Florida, 1995.

[Goudreau *et al.*, 1996] M Goudreau, K Lang, S Rao, T Suel, and T Tsantilas. Towards Efficiency and Portability: Programming with the BSP Model. In *Proc. of the* 8^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 1–12, 1996.

[Hagerup *et al.*, 1991] T Hagerup, A Schmitt, and H Seidl. FORK: A High-Level Language for PRAMs. In *Proc. of PARLE '91*, 1991.

[Harbison, 1992] S Harbison. *Modula-3*. Prentice Hall, 1992.

[Heywood and Leopold, 1995] T Heywood and C Leopold. Models of Parallelism. In J R Davy and P M Dew, editors, *Abstract Machine Models for Highly Parallel Computers*. Oxford University Press, 1995.

[Heywood and Ranka, 1992a] T Heywood and S Ranka. A Practical Hierarchical Model of Parallel Computation I: The Model. *Journal of Parallel and Distributed Computing*, 16:212–232, 1992.

[Heywood and Ranka, 1992b] T Heywood and S Ranka. A Practical Hierarchical Model of Parallel Computation II: Binary Tree and FFT Algorithms. *Journal of Parallel and Distributed Computing*, 16:233–249, 1992.

[Hill and Skillicorn, 1997] J M D Hill and D B Skillicorn. Lessons learned from implementing BSP. In *High Perfomance Computing and Networking (HPCN'97)*, volume 1225, pages 762–771. Springer-Verlag, April 1997.

[Hill *et al.*, 1996] J M D Hill, P I Crumpton, and D A Burgess. The theory, practice, and a tool for BSP performance prediction applied to a CFD application. In *EuroPar'96*, number 1124 in Lecture Notes in Computer Science, pages 697–705. Springer-Verlag, August 1996.

[Hill *et al.*, 1997a] J M D Hill, S R Donaldson, and D B Skillicorn. Portability of performance with the *BSPLib* communications library. In *Programming Models for Massively Parallel Computers, (MPPM'97)*, London, November 1997. IEEE Computer Society Press.

[Hill *et al.*, 1997b] J M D Hill, S A Jarvis, C Siniolakis, and V P Vasilev. Portable and Architecture Independent Parallel Performance Tuning using a Call-Graph Profiling Tool: A Case Study in Optimising SQL. Technical Report TR-17-97, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, May 1997.

[Hill *et al.*, 1997c] J M D Hill, W McColl, D Stefanescu, M Goudreau, K Lang, S Rao, T Suel, T Tsantilas, and R Bisseling. BSPlib, The BSP Programming Library. Technical report, Oxford University Computing Laborartory, May 1997. http://www.bsp-worldwide.org.

[Hoare, 1985] C A R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Inmos, 1988] Inmos. *OCCAM-2 Reference Manual*. Prentice Hall, 1988.

[Jájá, 1992] J Jájá. *Parallel Algorithms*. Addison-Wesley, 1992.

[Juurlink and Wijshoff, 1996] B H H Juurlink and H A G Wijshoff. Communication primitives for BSP computers. *Information Processing Letters*, 58:303–310, 1996.

[Kernighan and Ritchie, 1988] B Kernighan and D Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[Kessler and Seidl, 1995] C W Kessler and H Seidl. Fork95 Language and Compiler for the SB-PRAM. In *Proceedings of the* 5^{th} *International Workshop on Compilers for Parallel Computers*, June 1995.

[Knee, 1994a] S J Knee. BSP Programming on the TAP and EC40 Nodes. Internal Report, Marconi Radar Systems Limited, August 1994.

[Knee, 1994b] S J Knee. Program Development and Performance Prediction on BSP Machines using Opal. Technical Report TR-18-94, Programming Research Group, Oxford University Computing Laboratory, 1994.

[Koelbel *et al.*, 1990] C Koelbel, P Mehrota, and J Rosendale. Supporting shared data structures on distributed memory architectures. In *Proc. of the* 2^{nd} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990.

[Lam and Rinard, 1991] M S Lam and M C Rinard. Course-Grain Parallel Programming in Jade. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, 1991.

[Lecomber, 1995] D Lecomber. Object-Oriented Programming with BSP++. In *Proc. of the PPECC Workshop on Parallel and Distributed Computing*, 1995.

[Leighton, 1992] F Leighton. *Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.

[Maggs *et al.*, 1995] B M Maggs, L R Matheson, and R E Tarjan. Models of Parallel Computation: A Survey and Synthesis. In *Proc. of the* 28^{th} *Hawaii International Conference on System Sciences (HICSS)*, volume 2, pages 61–70, January 1995.

[McColl and Miller, 1995] W F McColl and Q Miller. The GPL language: Reference Manual. Technical report, ESPRIT GEPPCOM Project, Oxford University Computing Laboratory, October 1995.

[McColl, 1993a] W F McColl. General Purpose Parallel Computing. In Gibbons and Spirakis [1993], chapter 13, pages 337–391.

[McColl, 1993b] W F McColl. Special Purpose Parallel Computing. In Gibbons and Spirakis [1993], chapter 12, pages 261–336.

[McColl, 1994a] W F McColl. BSP Programming. In G E Blelloch, K M Chandy, and S Jagannathan, editors, *Specification of Parallel Algorithms. Proc. DIMACS Workshop, Princeton, May 9-11, 1994*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 21–35. American Mathematical Society, 1994.

[McColl, 1994b] W F McColl. The BSP Approach to Architecture Independent Parallel Programming. Technical report, Oxford University Computing Laboratory, December 1994.

[McColl, 1996a] W F McColl. Scalable computing. In J van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 41–61. Springer-Verlag, 1996.

[McColl, 1996b] W F McColl. Universal computing. In L Bouge, P Fraigniaud, A Mignotte, and Y Robert, editors, *Proceedings of Euro-Par '96 Parallel Processing*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[Mehlhorn and Vishkin, 1984] K Mehlhorn and U Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984.

[Miller, 1993] R Miller. A library for Bulk Synchronous Parallel programming. In *Proc. of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, pages 878–883, December 1993.

[Miller, 1994] R Miller. *Two approaches to architecture-independent parallel computation*. PhD thesis, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1994.

[Nash *et al.*, 1995] J M Nash, P M Dew, M E Dyer, and J R Davy. Parallel Algorithm Design on the WPRAM Model. In *Abstract Machine Models for Highly Parallel Computers*, pages 83–102. Oxford University Press, 1995.

[Perrott, 1992] R H Perrott. Parallel language developments in Europe: an overview. *Concurrency: Practice and Experience*, 4(8):589–617, December 1992.

[Ranade, 1989] A G Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, May 1989.

[Scales *et al.*, 1991] D Scales, M Rinard, M Lam, and J Anderson. Hierarchical Concurrency in Jade. In *Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science, pages 50–64. Springer-Verlag, August 1991.

[Siniolakis, 1996a] C Siniolakis. Direct Bulk-Synchronous Parallel Algorithms in Computational Geometry. Technical Report TR-10-96, Programming Research Group, Oxford University Computing Laboratory, 1996.

[Siniolakis, 1996b] C Siniolakis. On the complexity of BSP sorting. Technical Report TR-9-96, Programming Research Group, Oxford University Computing Laboratory, 1996.

[Skillicorn *et al.*, 1996] D B Skillicorn, J M D Hill, and W F McColl. Questions and Answers about BSP. Technical Report TR-15-96, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, 1996.

[Sommerville, 1989] I Sommerville. *Software Engineering*. Addison-Wesley, third edition, 1989.

[Valiant, 1982] L G Valiant. A scheme for fast parallal communication. *SIAM Journal on Computing*, 11:350–361, 1982.

[Valiant, 1989] L G Valiant. Bulk-syncronous parallel computers. In M Reeve and S E Zenith, editors, *Parallel Processing and Artificial Intelligence*, pages 15–22. Wiley, 1989.

[Valiant, 1990] L G Valiant. A bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Welch, 1991] P H Welch. Securely Managed Pointers. *WoTUG Newsletter 15*, July 1991.