Descriptive Simplicity in Parallel Computing

Marcus I. Marr

Doctor of Philosophy
University of Edinburgh

1997

Abstract

The programming of parallel computers is recognised as being a difficult task and there currently exist a wide selection of parallel programming languages and environments. This thesis presents and examines the **Hierarchical Skeleton Model (HSM)**, a model of parallel programming that combines ease of use, portability and flexibility.

The model is based on the exploitation of *nested parallelism* in parallel algorithms expressed using a *hierarchy* of *algorithmic skeletons*. The model acknowledges that not all forms of parallelism can be expressed clearly using skeletons and allows the use of *ad hoc* parallelism within the controlled framework of the skeleton hierarchy.

The thesis describes the HSM model and defines the syntax and semantics of the HSM language. The model and language are evaluated using three problems and compared against solutions written using the Fork95++ language in a shared memory environment and the C++ language with the Message Passing Interface (MPI) in a distributed memory environment.

The thesis concludes that the combination of the HSM model and language with an ad hoc parallel base model proved successful in tackling the problems with clearer and more concise code than either of the alternative languages.

Acknowledgements

I would like to thank my supervisor Murray Cole for his excellent supervision and guidance. I am particularly grateful for his timely reading of several drafts and the suggestions he made.

I would also like to thank Mydge for her enduring patience and for looking after me during the lighter and darker moments of the research.

I cannot ignore the friendship and entertainment given by my past and present office mates: Martin, Rob, Isabel, Christina, Anna, Thomas, Thomas and Marcio. Fred gets a mention all of his own for his invaluable proof-reading, comments and coffee.

Finally, I would like to thank my parents and siblings for their encouragement and support over the years.

Declaration

I declare that this doctoral thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. The following article was published during my period of research.

- M. Marr and M. Cole. Hierarchical Skeletons and ad hoc Parallelism. In E. D'Hollander, G.R. Joubert, F.J. Peters, and D. Trystram, editors, *Parallel Computing: State-of-the-Art and Perspectives*, volume 11 of *Advances in Parallel Computing*, pages 673-676. Elsevier, 1996.

Table of Contents

Chapter 1 Introduction 1

1.1 Models of Parallelism 2

1.1.1 The Message Passing Model 4

1.1.2 The Shared Memory Model 5

1.1.3 The Hierarchical PRAM Model 7

1.1.4 The Bulk Synchronous Parallel Model 7

1.2 Structured Parallelism 9

1.2.1 P3L 10

1.2.2 SKIL 11

1.2.3 SCL 11

1.3 Developing a New Model of Parallel Programming 12

1.3.1 The Skeleton Language 12

1.3.2 The Base Level Language 13

1.4 Contributions of the Thesis 14

I The Model **16**

Chapter 2 The Hierarchical Skeleton Model **17**

2.1 An Overview of the Hierarchical Skeleton Model 18

2.2 The Skeleton Nodes of the HSM 19

2.3 The Partitioning of a HSM node 22

2.4 The Leaf Nodes of the HSM 23

2.5 Summary 23

Chapter 3 Data Types in the HSM **25**

3.1 Data Structure Declaration and Creation 26

3.2 The HSM Scalar Types 27

3.3 The HSM Array Type 29

3.3.1 Array Declaration 29

3.3.2 Array Creation 30

3.3.3 Array Assignment 30

3.3.4 Array Concatenation 32

3.3.5 Array Querying 33

3.3.6 Array Initialisation 34

3.3.7 Array Transforming/Casting 35

3.4 The HSM Struct Type 36

3.5 Defining New Types 37

3.6 Other Data Types 37

Chapter 4 Skeletons in the HSM **39**

4.1 The Skeleton Function Syntax 40

4.2 Data Structure Partitioning 41

4.3 The if Skeleton 42

4.4 The while Skeleton 43

4.5 The map Skeleton 44

4.6 The zip Skeleton 45

4.7 The reduce Skeleton 46

4.8 The divide Skeleton 48

4.9 The scan Skeleton 49

4.10 The filter Skeleton 50
4.11 The pipe Skeleton 51
4.12 Other Skeletons 53

Chapter 5 A Shared Memory Target Architecture for the HSM 55

5.1 The HSM Node Hierarchy 55
5.2 The HSM Data Structures 56
5.3 The HSM/Fork95++ Interface 57
5.4 A HSM/Fork95++ Example 59

Chapter 6 A Distributed Memory Target Architecture for the HSM 60

6.1 The HSM Node Hierarchy 61
6.2 The HSM Data Structures 62
6.2.1 The HSM Scalar Types 64
6.2.2 The HSM Array Type 64
6.2.3 The HSM Compound Structure Type 68
6.2.4 Data Structure Distribution after Assignment 68
6.3 The HSM/C++/MPI Interface 70
6.3.1 The HSM Scalar Types 70
6.3.2 The HSM Array Type 72
6.3.3 The HSM Index Type 73
6.3.4 The HSM Compound Structure Types 75
6.4 A HSM/C++/MPI Example 75

II Evaluation 77

Chapter 7 Evaluation of the HSM Model 78

7.1 The Problems 78

7.1.1 The Image Thresholding Problem 78

7.1.2 The Merge-Sort Problem 79

7.1.3 The All-Pairs Shortest Paths Problem 79

7.2 The Languages 79

7.3 The Software Metrics 80

7.3.1 Solution Size 82

7.3.2 Solution Complexity 82

7.4 The Language Comparison 83

Chapter 8 The Image Thresholding Problem 84

8.1 The HSM Skeleton Superstructure 85

8.2 The Pure HSM Solution 86

8.3 The HSM/Fork95++ Solution 88

8.4 The HSM/C++/MPI Solution 91

8.5 The Pure Fork95++ Solution 95

8.6 The Pure C++/MPI Solution 97

8.7 Summary 98

Chapter 9 The Merge-Sort Problem 102

9.1 The HSM Skeleton Superstructure 103

9.2 The Pure HSM Solution 105

9.3 The HSM/Fork95++ Solution 107

9.4 The HSM/C++/MPI Solution 107

9.5 The Pure Fork95++ Solution 111

9.6 The Pure C++/MPI Solution 113

9.7 Summary 115

Chapter 10 The All-Pairs Shortest Paths Problem **118**

10.1 The HSM Skeleton Superstructure 119

10.2 The Pure HSM Solution 121

10.3 The HSM/Fork95++ Solution 124

10.4 The HSM/C++/MPI Solution 126

10.5 The Pure Fork95++ Solution 129

10.6 The Pure C++/MPI Solution 129

10.7 Summary 132

III Discussion 136

Chapter 11 Discussion **137**

11.1 Structured Parallelism 138

11.1.1 Nested Parallelism 138

11.1.2 Flexible Problem Sizes 138

11.1.3 Flexible Machine Sizes 139

11.1.4 Use of Ad Hoc Parallelism 139

11.2 The HSM Language 140

11.2.1 Syntax 140

11.2.2 Data Types 142

11.2.3 Skeletons 144

11.2.4 Efficiency 146

11.3 The HSM Base Models 147

11.3.1 The Shared Memory Base Model 147

11.3.2 The Distributed Memory Base Model 148

11.3.3 A BSP Base Model 151

11.3.4 A Threaded Base Model 151

11.4 An HSM Cost Model 152

11.5 An HSM Compiler 153

11.5.1 Compiler Design 154

11.5.2 Compiler Optimisations 154

11.6 Extending the HSM Model 155

Chapter 12 Conclusions **157**

12.1 Sequential and Parallel Base Languages 157

12.2 The HSM Language 159

12.3 The HSM Base Model Interface 159

12.4 Further Work 160

12.5 The Future of Parallel Computing? 160

12.6 Summary 161

Appendix A The HSM Language Grammar **162**

Appendix B The Image Thresholding Code **166**

B.1 The HSM Skeleton Superstructure 166

B.2 The Pure HSM Solution 169

B.3 The HSM/Fork95++ Solution 170

B.4 The HSM/C++/MPI Solution 173

B.5 The Pure Fork95++ Solution 178

B.6 The Pure C++/MPI Solution 181

Appendix C The Merge-Sort Problem **185**

C.1 The HSM Skeleton Superstructure 185

C.2 The Pure HSM Solution 188

C.3 The HSM/Fork95++ Solution 192

C.4 The HSM/C++/MPI Solution 193

C.5 The Pure Fork95++ Solution 198

C.6 The Pure C++/MPI Solution 201

Appendix D The All-Pairs Shortest Paths Problem 204

D.1 The HSM Skeleton Superstructure 204

D.2 The Pure HSM Solution 205

D.3 The HSM/Fork95++ Solution 209

D.4 The HSM/C++/MPI Solution 211

D.5 The Pure Fork95++ Solution 212

D.6 The Pure C++/MPI Solution 215

Bibliography 220

Chapter 1

Introduction

Contents

1.1 Models of Parallelism 2
1.2 Structured Parallelism 9
1.3 Developing a New Model of Parallel Programming . 12
1.4 Contributions of the Thesis 14

The programming of parallel computers is recognised as being a difficult task and there currently exist a wide selection of parallel programming languages and environments. This thesis presents and examines the **Hierarchical Skeleton Model (HSM)**, a model of parallel programming that combines ease of use, portability and flexibility.

The model is based on the exploitation of *nested parallelism* in parallel algorithms expressed using a *hierarchy* of *algorithmic skeletons*. The model acknowledges that not all forms of parallelism can be expressed clearly using skeletons and allows the use of *ad hoc* parallelism within the controlled framework of the skeleton hierarchy.

This chapter introduces some of the existing models of parallel computation and programming languages, and in particular discusses the recent field of structured parallel computing. The HSM model is then compared with the other models and the design decisions taken for the model are discussed.

Part I defines the HSM model and language, with Chapters 2, 3 and 4 presenting the programming model and the syntax and semantics of the HSM language. Chapters 5 and 6 relate the model to shared memory target architectures and distributed memory target architectures respectively.

The model and language are evaluated in Part II, with Chapters 7, 8, 9, and 10 tackling a selection of non-trivial parallel algorithms. The solutions are contrasted with those obtained using the lower level parallel language/environment of the target architectures.

Lastly, the effectiveness of the model is discussed in Chapter 11 of Part III and the conclusions presented in Chapter 12. The full grammar is listed in Appendix A and code listings for the three sample problems are given in Appendices B, C, and D respectively.

1.1 Models of Parallelism

A model is simply an abstract view of a system which allows the user to ignore lower level details and concentrate on the main features. In parallel computing, Heywood and Leopold[HL93] suggest a loose three level hierarchical framework of model types:

1. Architectural (Technology) Models
2. Computational (Algorithmic) Models
3. Programming (Semantical) Models

A **model of parallel architecture** abstracts the hardware and operating system that make up a machine, defining amongst other things how the processors communicate and synchronise. McColl[McC94] suggests that a "standard" architectural model is emerging to facilitate scalable parallel computing:

> *[The architectural model] is likely to consist of a collection of (workstation-like) processor-memory pairs connected by a communications network which can be used to efficiently support a global address space. Both message passing and shared memory programming styles will be efficiently supported on these architectures.*
>
> ([McC94])

The recent general purpose Cray T3D and T3E parallel computers both conform to this architectural model. The memory resources are distributed with each processor having a local memory module, but the processor interconnection network allows for global memory access without the intervention of the processors. Shared memory is therefore supported, but a non-uniform two-level memory hierarchy exists with private memory accesses costing less than shared memory accesses.

Although some machines allow multiple processes to execute on a single processor the current operating systems of the above machines restrict the execution to one process per processor. The processes in the above machines execute independently and may synchronise with one another using a combination of software and hardware mechanisms.

The asynchronous execution of processes allows a *multiple-instruction-multiple-data* (MIMD) programming style, although in practice most parallel applications are written in a *single-program-multiple-data* (SPMD) style. The SPMD computer executes the same program text on each processor, though the execution will probably follow different paths on different processors. Any set of MIMD programs can be replaced by a single, merged, program, so there is no loss in restricting to the SPMD programming style[McB94].

A **model of parallel computation** gives "an abstract view of a class of architectures while accurately reflecting costs and resources of those architectures"[HL93]. It is also necessary to facilitate progress in high performance computing:

> *[The computational model] is the idealisation of computation that computer architects strive to support [with] the greatest possible performance. The model is the specification of the computational engine that language and operating systems designers can assume as they seek to enhance the power and convenience of parallel machines.*
> ([ICC+96])

A **model of parallel programming** lies on top of the computational model and should be designed:

> *... for the purpose of providing enough abstraction so that large programs can be organised, reasoned about, and verified correct. Hence, their defining characteristic is simplicity of use.* ([HL93])

The HSM model defined in this thesis falls into this class of parallel programming models.

It is difficult to separate the models of computation from the models of programming, and most existing work on models of parallelism has centred around computational models. For this reason the parallel languages proposed have mostly been low level languages with direct mappings to the underlying computational model. Various criteria have been suggested for evaluating the computational models, for example *scalability*, *portability* and *predictability*[HMS+97]. When considering the computational and programming models together, additional criteria include *programmability*, *ease of understanding*, and the existence of a *software development* *methodology*[ST96]. The key areas remain *portability* and *efficiency*.

The following sections describe some of the existing models of parallel computation and programming.

1.1.1 The Message Passing Model

Message passing has to date been one of the two most significant models of parallel programming due in large part to the relatively simpler hardware requirements of the underlying machine and the ability to exploit existing sequential algorithms. Recent work in establishing architecture-independent message passing libraries, such as MPI[Wal94] and PVM[SGDM94], has ensured that parallel programs are portable and execute efficiently. This is especially true with the emerging standard model of parallel architecture described above.

A good overview of message passing environments is given by McBryan:

> *A message passing program consists of P sequential programs, one run in each process. Each sequential program uses message passing instructions to synchronise with and to access memory of other programs. Typically these instructions come from a limited set of instructions that define the Message Passing Environment. Usually the instructions are made available to programmers in the form of a Message Passing Library. The programming environment then consists of a standard uniprocessor programming language such as C along with the message passing library.* ([McB94])

The Message Passing Interface (MPI)[For94] is one such library of communication routines. The interface uses process groups and communication contexts to provide a logical grouping of communicating processes, starting with one group representing all the processes in the machine and the single communication context MPI_COMM_WORLD. During execution the processes can be formed into additional groups. Each group can have its own communication context, created by 'splitting' existing communicators, though this only really becomes useful for the collective communication routines which will be described shortly.

Processes communicate with one another by sending packets of information using the *point-to-point* communication routines such as MPI_Send() and MPI_Recv(). MPI includes a wide selection of routines to offer different sorts of synchronisation in the sending which can be employed to improve the efficiency of an implementation at the cost of increased program complexity. The *type* of the values can be simple basic types such as integer and real, or *derived* types can be created by the programmer.

MPI also offers a range of *collective communication* routines which can be used to perform a common operation across all the processes in a specific communication context. Examples include MPI_Bcast() to broadcast a value from one process to all the others, and MPI_Scatter() to distribute a one-dimensional array of values over all the processes. There is also limited scope for performing *collective computation* within a communication context, e.g. MPI_Reduce() to perform a tree of binary operations, such as addition, on one value from each process.

The message passing programming model can be used to create portable and efficient code with the conceptual cost of using a low level of abstraction. The code is potentially scalable, but may require significant effort from the programmer. There is no established software development methodology and performance prediction is still only an embryonic research area.

1.1.2 The Shared Memory Model

The shared memory programming model is the other significant model of parallel programming due to the relatively higher level programming possible than that of message passing. The parallel machine is composed of several processors with a single common memory space accessible to all the processors. The processors may execute with implicit synchronisation, for example in data parallel programming, or with explicit synchronisation through the use of locks and semaphores.

Traditionally, data parallel programming has been seen as a SIMD programming model. The model is particularly suited to some physical simulations which require mathematical operations on large matrices of data.

Shared memory programming does not have to be SIMD based, as demonstrated by the Fork95 shared memory programming language. The FORK language was developed by Hagerup *et al*[HSS92, HSS90, Sch91, RS92, Sei93] for expressing PRAM algorithms for a physical realisation of a PRAM machine called the Saarbrücken PRAM (SB-PRAM)[AKP91a, AKP91b, DS92]. The language has been developed into the more realistic and usable Fork95 language by Kessler and Seidl[KS95a, KS95b, KS97]. Fork95 is based on the 'C' language, with additional constructs and type specifiers. The most important new constructs are the **start** {}, **farm** {} and **fork** () {} constructs. **start** {} is used to synchronise the processor into a single group at the beginning of a SPMD program, while **farm** {} is used to remove the synchronisation constraints.

Variables can be shared over all the processors in a group or can be private to each individual processor. In this way expressions are either shared, i.e. they involve no private variables, or private. The symbols ' `@`' and '`$`' represent the group number and rank number respectively for each processor.

The `fork` construct is used to split a group of processors into sub-groups:

```
fork ( exp1; @=exp2; $=exp3 ) {
    ...
}
```

The first expression is shared and decides how many sub-groups the processors will split into. The second expression is private and specifies which sub-group each processor will join. The third expression, also private, sets the *rank* of each processor in the new sub-groups.

The Fork95 language has a selection of powerful atomic multi-prefix operators that are inherited from the SB-PRAM, namely add; max; and; and or. For example the multi-prefix add operator takes one shared and one private argument:

mpadd(&shvar, *expr*);

The private value from evaluating the private expression is atomically added to the shared integer ' shvar' and the function returns the old value of ' shvar'. With all the processors synchronised, the processor with the ith largest physical processor ID receives the (private) value $s_0 + \sum_{j=0}^{j=i-1} e_j$ where s_0 is the value of ' shvar' prior to the execution and e_j the value of *expr* for the processor with $=j$. After execution ' shvar' contains the global sum $s_0 + \sum_j e_j$.

Further developments of the Fork95 language suggested by the authors[KT96] include a Fork95++ language based on 'C++'. Chapter 5 describes an HSM interface for the hypothetical Fork95++ language, which is used in Chapters 8, 9 and 10 to express the solutions more elegantly, though Fork95 implementations could have been written instead.

The shared memory/Fork95 model allows the programmer to write scalable parallel programs with predictable performance which are portable over all shared memory architectures. The assumption of uniform cost shared memory is unfortunately unrealistic for many mainstream parallel architectures, and the need for continuous implicit synchronisation in the Fork95 language is another problem in that programs will execute inefficiently on architectures with even a modest synchronisation cost.

1.1.3 The Hierarchical PRAM Model

Heywood and Ranka propose the Hierarchical PRAM model (H-PRAM)[HR92a, HR92b] to give a computational model with a cost model that recognises data locality:

> *The H-PRAM is a model which balances simplicity of use, and reflectivity of the costs and resources of architectural models employing direct networks. It is the most general of the "PRAM with locality" type models.* ([ICC+96])

The authors augment a generic SIMD PRAM language with a `partition{}` instruction to partition the machine and its resources, and propose two variants of the model. Processors in the *private* variant of the H-PRAM are only able to access memory within their current sub-groups and the sub-groups have to explicitly *inherit* data from the parent group during the partitioning. In contrast, processors in the *shared* variant of the H-PRAM may access any data they may know about through the scoping of the processor hierarchy. The Fork95 language can be viewed as a language for the shared variant H-PRAM.

The H-PRAM/Fork95 model should permit more efficient programs than the PRAM/Fork95 model by reducing the need to synchronise the processors and by recognising the performance gains which can be achieved by exploiting data locality. The model, however, still suffers from inefficiency when implemented on mainstream architectures which rely on software synchronisation.

1.1.4 The Bulk Synchronous Parallel Model

McColl and Valiant propose the Bulk Synchronous Parallel (BSP) model of parallel computation[Val90, McC92, McC93a], to provide a "simple, unified framework for the design and programming of all kinds of general purpose parallel systems"[McC94]. The model parameterises the architectures using four parameters, namely p, s, l and g. p is the number of processors, s is a measure of their speed, l is a measure of the time required to synchronise the processors and g is a measure of the system-wide communication cost relative to processor speed. Any parallel computing system can be regarded as a BSP computer by allocating the appropriate BSP parameters.

A computation for a BSP machine is composed of a sequence of *super-steps*, where each super-step consists of a sequential block of code followed by a barrier synchronisation. Communication takes the form of non-local memory accesses which take place during the synchronisation step. A BSP library[Mil93] exists for the 'C' language where the super-steps are started and finished with `bsp_sstep()` and `bsp_sstep_end()` function calls, and non-local memory access is explicitly requested through `bsp_fetch()` and `bsp_store()` calls.

The BSP library is still in development, with the most recent reference document [HMS+97] replacing the super-step start and end functions with a single `bsp_sync()` instruction, and the non-local memory access calls with `bsp_put()` and `bsp_get()`.

In comparison to using the message passing of MPI or PVM, BSP offers a higher level of abstraction and a cost model for performance analysis and prediction. The authors recognise, however, that the *direct-remote-memory-access* of the BSP model is less convenient for computations involving irregular or data dependent communications. The latest version of the BSP library has therefore included Bulk Synchronous Message Passing routines, including `bsp_send()` and `bsp_recv()` and the various attendant functions. There are plans to extend the BSP library to include versions of the MPI collective routines, such as broadcast, scatter, reduce and scan.

One way in which the BSP cost model achieves the relatively high level of abstraction is by "renouncing" locality as a performance optimisation. This is a contentious issue, as "locality is the prime resource that one can extract from problems to boost the performance of their solutions"[HL93], although the BSP authors argue that the adverse affect on performance is not significant for most application domains.

Locality is also a natural result of a programming model and methodology based on decomposition, where parallelism may be nested through an algorithm. Unfortunately, the BSP model does not allow for subsets of processors to be synchronised independently, so the nested parallelism of a computation must be flattened out before it can be expressed in the BSP model. The inability to synchronise a subset of processors is an artefact of the BSP programming model rather than the BSP model of computation:

> *It is not yet clear what a MIMD, subset-synchronising language should be like if it is to retain the characteristics of BSP.* ([SHM96])

Existing work on a model of parallel programming for the BSP computational model has therefore concentrated on globally synchronised PRAM-style languages, although the model is compatible with a variety of language styles. Existing BSP-based languages include GPL[McC94] and BSP-L[CFSV94].

The BSP model and programming languages allow the programmer to write scalable and portable programs with efficient and predictable performance using a slightly higher level of programming than that offered by the simpler message passing environments. There is no software development methodology, however, and the need to flatten all the parallelism into global super-steps reduces the ease of use of the model.

1.2 Structured Parallelism

One programming approach which has the potential to satisfy most of the criteria given at the start of Section 1.1 is *Skeleton Oriented Programming*. The parallelism of an algorithm is encapsulated within a hierarchy of algorithmic skeletons[Col89, DFH+93] giving the programmer a high level abstract programming model. The model removes the need for the programmer to have to consider the details of mapping the parallelism to the target architecture, leading to potentially fewer mistakes in the coding of an algorithm and clearer code, i.e. the model improves programmability and the ease of understanding of the code.

The hierarchy of algorithmic skeletons also encourages a *stepwise-refinement* programming methodology, in which each level of parallelism is decomposed into further levels of parallelism. The methodology requires that efficient implementations of each of the skeletons have been created for the target architecture. Since those skeletons may utilise an arbitrary number of processors this allows the programmer to write scalable, efficient and portable code without sacrificing program performance. Abstraction and stepwise refinement are two of the fundamental design concepts at the heart of software engineering[Pre92]. Other fundamentals include modularity, a hierarchical software architecture, a control hierarchy and data structure. Each of these concepts is encouraged by the use of a hierarchy of algorithmic skeletons.

The high level of abstraction removes the specific details of the parallelism from the code and allows the programmer to concentrate on the algorithm, thus improving the programmability and ease of understanding of the programming model. Lastly, the regular structure of the hierarchy of skeletons can be used to map a cost model onto the underlying model of computation. This cost model can be used not just to look at the performance of a program on a particular machine, but also to estimate or predict performance on alternative machines.

Research into suitable cost models for structured parallelism is still developing quickly, but one such model is the CLUMPS model of parallel computation[Cam94, Cam96] which consists of an architectural model and a cost model. The architectural model provides for the partitioning of the processor interconnection network into sub-networks, resulting in a multi-level memory hierarchy with non-uniform memory access costs. The cost model is reflective of the (potentially) reduced regional costs and lends itself to the recursive-partitioning nature of nested algorithmic skeletons.

Early work in the area of using algorithmic skeletons by Cole concentrated on describing each program using a single skeleton[Col89]. The computations which "fleshed out" the structure were specified and computed sequentially. Subsequent work by various groups has been addressing the complications that arise by allowing the composition and nesting of algorithmic skeletons. In each case, the computations have been expressed using a sequential language, such as ML[Bra95, Bra92], Haskell[RS93, Rab93], common Lisp[Ble95], Hope[JS91], 'C'[Pel93, BK96b], and Fortran[DGTJ95].

The following sections describe three of the above skeleton programming languages and highlight the main features and skeletons. We focus on the languages which use an imperative base language, i.e. 'C' or Fortran, which is in common with the proposed HSM model, rather than the functional base languages which categorise the other models.

1.2.1 P3L

Danelutto *et al* propose the P^3L language[DMO+92, Pel93, DP93a, DP93b] to express structured parallel computations. The methodology requires the programmer to express the parallelism using a hierarchical composition of second order *constructors*, similar to skeletons, using sequential 'C' as the base language. The range of constructs includes farm, pipe, map, reduce, loop and geometric constructors. The compiler exploits the parallelism of the constructors using pre-defined implementation templates which have been targeted for specific architectures.

1.2.2 SKIL

Botorog and Kuchen propose the SKeleton Imperative Language (SKIL) for skeleton oriented programming [BK96b, BK95, BK96c, BK96a] which is also based on sequential 'C'. 'C' is used as the host language and is extended to allow higher order functions, partial application of functions and polymorphism. The language uses *parallel abstract data types* (PADT) which control data access and allow regular and irregular problems to be dealt with equally. One example PADT is the array type which has a block-wise distribution. Method functions associated with the array data type include map, fold, permute and horizontal/vertical shifts.

1.2.3 SCL

Darlington *et al* propose a framework for parallel composition which "provides a natural integration of the compositional programming approach with the data parallel programming paradigm."[DGTJ95]. The authors define an upper-layer Structured Composition Language (SCL) to abstract a program's parallel behaviour, and use Fortran for the lower level base language to express sequential computation.

The functional skeletons of the SCL contain the details of the data parallel programming paradigm, so include partitioning, data placement, data movement and control flow aspects of the program. The authors separate the SCL skeletons into three categories: configurational, elementary and computational skeletons. Configuration skeletons control the distribution and alignment of nested distributed data structures, e.g. `distribution`, `partition`, `gather`, `split`, and `combine`. Elementary skeletons abstract the basic operations of the data parallel model of computation, e.g. `map`, `fold`, and `scan`. The elementary skeletons also include a set of 'bulk data movement functions', e.g. `rotate`, `brdcast`, `applybrdcast`, `send`, and `fetch`. Lastly, the computational skeletons abstract commonly used parallel computational patters such as `farm`, `SPMD`, and `iterateUntil`.

The framework captures the patterns of parallel computation which occur in data parallel programming, but the programmer is still required to distinguish between *sequential* data structures and nested *parallel* data structures of sequential structures. The majority of the functional skeletons in the SCL language are therefore provided to control the distribution and communication of the sequential data structures, although they do provide a greater level of abstraction than the point-to-point message passing of the underlying Fortran/MPI environment.

1.3 Developing a New Model of Parallel Programming

The skeleton programming models described in the previous section have the common characteristic that in each case the 'flesh' computations are expressed using a sequential base language, either functional or imperative. All the parallelism must therefore be expressed using only the skeletons available in the skeleton language.

Restricting the parallelism in this way has two main disadvantages. Firstly, it prevents the programmer from reusing existing parallel code written in a different parallel programming language, and in particular ignores the large body of specialised parallel libraries. Secondly, it restricts the patterns of parallelism available to the programmer. The patterns needed may be unstructured or have a regular structure which cannot be expressed cleanly using the existing skeletons.

In this thesis I propose the Hierarchical Skeleton Model (HSM) which allows the composition and nesting of algorithmic skeletons, and in which the 'flesh' computations can be expressed using a *parallel base language* instead of just a sequential base language. The leaf nodes of the hierarchy therefore use a parallel model of computation and language instead of a sequential model and language. Early work on the HSM model was published in [MC96], though the model and language have since been refined.

The *base model* allows the programmer to use ad hoc parallelism within the leaf nodes of the skeleton hierarchy while retaining the structured hierarchy of the nested parallelism. The programmer can use parallel library routines, or other existing parallel code, without having to rewrite the library using a skeleton programming language. Algorithms which have patterns of communication/ computation that cannot be expressed using the HSM language's skeletons can be expressed using the explicit parallelism of the base language.

1.3.1 The Skeleton Language

The HSM language proposed in Chapters 2, 3 and 4 is an imperative language with a style similar to that of the 'C' language. The language could just as easily have been based on a functional style, but the imperative style was chosen as the target base languages are also imperative.

Functions are defined using a 'C'-type syntax, and a *function-name* type exists to hold the name of a specific function. It is possible to pass the name of a function as an argument to another HSM function, and even to create arrays of function names and compound structures containing function names. It is not possible to define 'anonymous' functions, i.e. creating a new function as the argument where a function name is expected, so any function that is passed as an argument must be defined separately. The language requires that each function be defined fully, so it is also not possible to use partial function applications. This is simply a restriction of the HSM language proposed and not a fundamental restriction of the programming model.

Like the SKIL language, but unlike the P^3L language, the HSM language supports nested data structures. The distribution of nested data structures is more complicated than that of un-nested data structures, but the nesting forms an important tool for the programmer. The HSM language does not, however, support recursive data structures, e.g. trees of arbitrary depth, which are also excluded from the SKIL language. Again, this is a restriction of the HSM grammar and not a restriction of the programming model.

1.3.2 The Base Level Language

The computational model of the leaf nodes of the HSM hierarchy is a SPMD model or, more accurately, a *single-function-multiple-data* (SFMD) model in that a single function is called by each processor in the group. Data structures passed to the leaf nodes are then spread over the member processors of the leaf nodes. In a distributed memory architecture each processor is therefore likely to hold only a small part of the data structure.

In the programming models described in Section 1.1 data structures are either fully shared or distributed. In a *distributed data structure* each processor holds a standard sequential construct such as an array but knows nothing about where the local data fits in to the larger picture, i.e. the structure is *implicitly* part of the single structure spanning the node. For example, an array may be split into contiguous sections with one section per processor. It is left to the programmer to collate the other information, such as total size of the array and the index offset of the first element.

An alternative to distributed structures are *parallel data structures* where the structures held by each processor are *explicitly* part of a single structure spanning the node. This has the appearance of a partial-shared memory environment in that there is effectively only one data structure spread over the processors but there is no shared memory or virtual shared memory. The processors must still communicate with one another to access data held by other processors. The main difference between distributed and parallel data structures lies in how elements, e.g. of an array, are accessed. In a distributed structure the elements are indexed starting from zero on each processor. In a parallel structure the elements are indexed using the 'global' indices of the elements. This also allows for irregular parallel data distributions. The HSM language uses the latter 'parallel' data structures in the base language.

Within the leaf nodes the processors may communicate with one another using the communication features of the base language, e.g. message passing or shared variables. This allows ad hoc parallelism to be incorporated into the program as long as it can be expressed using the base language. The main disadvantage of including ad hoc code is that the portability of the program is restricted to architectures which support the specified base model of parallelism. For example, if a shared memory base model is chosen then the target architecture must be able to support a shared memory environment. The efficiency of the ported program will therefore depend on how efficiently the base models is supported.

1.4 Contributions of the Thesis

This thesis proposes a new model of programming for structured parallelism, called the Hierarchical Skeleton Model (HSM). As with the other models for structured parallelism described in Section 1.2 the HSM model is independent of any particular parallel architecture. The most significant difference between the HSM model and the alternative structured programming models is that the HSM model uses a *parallel* base model and language. This allows the programmer to combine the regular and structured properties of skeleton oriented programming with the flexible, though irregular, abilities of the base language.

Unlike the P^3L language, the HSM language permits the use of *nested* data structures. The SCL language also permits nested structures, but the structures are *distributed* so that the programmer must explicitly control the distribution and alignment of the individual *sequential* structures. The HSM language uses *parallel* data structures whose distribution and alignment is implicit in the program.

The HSM model provides a scalable and programmable environment which is portable over all architectures which can support the chosen base model of parallelism. The high level of abstraction makes the programs simpler to express and to understand, and the skeleton hierarchy encourages structured control flow using a programming methodology of stepwise refinement.

The following chapters define the HSM model in detail and present the grammar for a HSM language. The model and language are evaluated using a selection of software metrics such as length and clarity on solutions to three problems. The metrics are compared against the metrics for solutions written using the Fork95++ language in a shared memory environment, and the C++ language with MPI in a distributed memory environment. The results are discussed in Chapter 11 and the conclusions summarised in Chapter 12.

Part I

The Model

Chapter 2

The Hierarchical Skeleton Model

Contents

2.1 An Overview of the Hierarchical Skeleton Model . . 18
2.2 The Skeleton Nodes of the HSM 19
2.3 The Partitioning of a HSM node 22
2.4 The Leaf Nodes of the HSM 23
2.5 Summary 23

This chapter and the following four chapters present the Hierarchical Skeleton Model (HSM) of parallel computation, the programming language associated with it, and interfaces for two parallel base languages. This chapter introduces the concepts underlying the HSM model and is followed by a description of the HSM language's data structures and algorithmic skeletons in Chapters 3 and 4 respectively.

The HSM model allows ad-hoc parallel base models to be used as the leaf nodes of the skeleton hierarchy, requiring a well defined interface between the base models and the HSM model. A synchronous shared memory target architecture is considered in Chapter 5 using the Fork95++ language on a shared memory machine, and a distributed memory target architecture is considered in Chapter 6 using the C++ language with the MPI message passing library on a MIMD machine. The programmability of the HSM model and language are evaluated in Part II of the thesis, where three example problems are used to compare the HSM solutions with those of the two pure base models without using algorithmic skeletons.

Section 2.1 of this changer gives an overview of the Hierarchical Skeleton Model which is expanded in the later sections. Section 2.2 gives an explanation of what each node of the hierarchy represents; Section 2.3 explains how the hierarchy can be dynamically extended by partitioning the nodes; and Section 2.4 describes what the leaf nodes of the hierarchy represent. Lastly, Section 2.5 summarises the main points of the model. A more extensive discussion, which includes the results of the evaluation of the model, is given in Part II.

2.1 An Overview of the Hierarchical Skeleton Model

As suggested by its name, the Hierarchical Skeleton Model allows the composition and nesting of algorithmic skeletons. The architectural model is based loosely on the Hierarchical-PRAM model of computation where each node in the hierarchy represents a self contained parallel machine, consisting of processors and memory resources. The hierarchy specifies how the processors within a node are partitioned at each level, and how access to the data structures is assigned to each of the child nodes created by the partitioning.

The code for the non-leaf nodes of the HSM hierarchy is expressed using the skeleton based language associated with the HSM model which will be described in detail in Chapters 3 and 4. This differs from the H-PRAM which uses a generic PRAM language augmented with a partition{} construct. The code for the leaf nodes of the hierarchy can be expressed either using the HSM language without further partitioning or by using the language of the base model, as will be discussed in Chapters 5 and 6.

The merging of the skeleton hierarchy with an arbitrary base level language absorbs the flexibility of existing parallel languages into the HSM model while retaining the regular and structured properties of the skeleton hierarchy. The architecture-independence properties of the algorithmic skeletons remain intact by restricting the ad-hoc parallelism to the leaf nodes where the ad-hoc communications can be contained.

The use of a sequential base language at the leaf nodes of the hierarchy has been looked at before, but no work has yet been done in allowing parallel languages to be used at the base level. This requires a thorough investigation of the model of computation created, the interface between the skeleton and base levels and the sorts of parallel data structures used.

2.2 The Skeleton Nodes of the HSM

Each non-leaf node in the skeleton hierarchy, representing a self contained parallel machine, executes a single function written in the skeleton language of the Hierarchical Skeleton Model. The HSM language uses a *single-program-single-data* (SPSD) programming paradigm which gives just a single thread of control through the function code of each node. This single thread of flow avoids the possibility of implicit group partitioning which is inherent in the PRAM model and Fork95 language.

The role of individual processors is absorbed into the SPSD code, making the code fully independent of the size of the node, i.e. the number of processors. The code is also independent of the memory architecture of the underlying machine, although the HSM language compiler may have to implement some form of virtual shared memory environment if the underlying architecture uses distributed memory. This should not be difficult using the single threaded SPSD model.

The language of the HSM chosen for this thesis is an imperative language, but the language could just as easily have been a functional programming language. The imperative design was chosen to provide a familiar style for the base level programmer, as both base languages used in this thesis are also imperative.

The code for each node is contained in a single SPSD function, described using the following grammar. Each function consists of a sequence of statements, where a statement is either the assignment of the value of an expression to a data structure, or the returning of an expression holding the resulting data structure of the function. Data structure expressions can contain basic arithmetic on scalar variables, arrays and compound structures[1]; instances of algorithmic skeleton[2]; calls to other HSM functions; and calls to base level functions. There is also a special built-in function **node_size()** which returns the number of processors in the local node. The function syntax is described using the following grammar[3]:

function:

type id '(' '`void`' ')' '{' *var-declarations statement-list* '}'
| *type id* '(' *arg-list* ')' '{' *var-declarations statement-list* '}'

id:

`[a-zA-Z][a-zA-Z0-9_]*`

[1]described in Chapter 3
[2]described in Chapter 4
[3]the complete grammar of the language is given in Appendix A

arg-list:

type id | *type id* ',' *arg-list*

var-declarations:

/* *nothing* */ | *var-declaration var-declarations*

statement-list:

statement | *statement statement-list*

statement:

variable '=' *expr* ';'

| 'return' '(' *expr* ')' ';'

expr-list:

expr | *expr* ',' *expr-list*

expr:

variable | *id* '(' ')' | *id* '(' *expr-list* ')' | 'node_size' '(' ')' ';'

variable:

id

The definition of the non-terminal grammar symbol *expr* is only a partial definition and will be defined comprehensively over the next two chapters. As it stands, the function call syntax accommodates calls to HSM functions, base functions and algorithmic skeletons[4]. The non-terminals *type* and *var-declaration* will also be described later.

Every function takes zero or more arguments, where each argument represents one 'input' data structure. The function may also return an 'output' data structure. For example:

```
int add_fn( int x, int y ) {
   return( x+y );
}
```

A complete HSM program contains a series of type definitions, base function prototypes, HSM function prototypes, and HSM functions:

hsm-program:

type-definitions base-function-prototypes function-prototypes functions

[4]An instance of an algorithmic skeleton is simply expressed as a function call with the same name.

type-definitions:

/ nothing */* | *type-definition type-definitions*

base-function-prototypes:

/ nothing */* | *base-function-prototype base-function-prototypes*

function-prototypes:

/ nothing */* | *function-prototype function-prototypes*

functions:

/ nothing */* | *function functions*

For example:

```
typedef array< int > bag;
basefn sum_fn( bag x );
int main( array< string > argv );
```

A HSM function prototype can be described as follows:

function-prototype:

type id '(' '`void`' ')' '' ';'
| *type id* '(' *arg-list* ')' ';'

A base function prototype is very similar, but is explicitly declared as being a base function using the keyword **basefn**:

base-function-prototype:

'`basefn`' *type id* '(' '`void`' ')' ';'
| '`basefn`' *type id* '(' *arg-list* ')' ';'

The HSM program must contain at least one HSM function, namely the **main()** function. The HSM model starts by executing this function over the entire machine, e.g.:

```
int main( array< string > argv )
{
    return( 1 );
}
```

The **main()** function is no different from any other HSM function in that it takes one parameter, in this case the array of command line arguments, and uses the **return()** statement to return a data structure holding the result of the function, in this case the exit status of the program. The data structures passed as arguments to a function are distributed over the memory resources of the local node, which for the **main()** function happens to be the entire machine.

2.3 The Partitioning of a HSM node

Of the four classes of statement mentioned above: data, skeleton, HSM function and base function; only skeleton operations and base function calls can make true parallelism available to the programmer. For skeleton operations this is done by partitioning the processors of the node into subgroups, with each subgroup representing a new node of the hierarchy and adopting an independent thread of control as defined in Chapter 4. Section 2.4 explains how parallelism can be exploited using base functions with further details in Chapters 5 and 6.

When the processors are partitioned into subgroups the memory resources of the parent group are also partitioned, using a model similar to the private variant of the H-PRAM. This means that each child node inherits a disjoint portion of the parent node's data structures, and access to other data held by the parent node or to data held by another child node is not possible except, where permitted, through the functionality of the parent skeleton that caused the partitioning. Each child node inherits its data through the arguments passed to the function representing the program for that child node. On completion, data is passed back to the parent group by means of the return argument of the child node's function.

The input arguments to the child nodes are defined as being read-only so that the data structures cannot be modified by the function code. This avoids the problems of potential write conflicts between separate nodes in the hierarchy and makes reasoning about the behaviour of a program simpler. The data structures passed to the skeletons are also separated into a *global* argument and a *local* argument. The global argument contains the information which is required by all the body functions of the skeleton, while the local argument contains information which will be partitioned so that each body function is given a disjoint part of the structure. This ensures that data access is effectively structured *within* the program hierarchy, with irregular or ad-hoc data access only permitted inside the base language leaf nodes. Details of how each skeleton partitions its data are given in Chapter 4.

The structuring of data access within the program hierarchy is one of the main advantages of the HSM over both the shared and private variants of the H-PRAM, in that it allows limited communication between the nodes of the hierarchy while ensuring that all inter-node data access is kept regular.

2.4 The Leaf Nodes of the HSM

The leaf nodes of the hierarchy are also self contained parallel machines and the code for the nodes can be still be expressed using the HSM language. By definition the node is only a leaf node if no attempt is made to partition the node further. This means that the code for the node doesn't use any skeletons or base functions, or call any HSM functions which might. The leaf node HSM functions are therefore simply sequential data manipulation functions with no parallelism. This is the case for much of the current research into skeleton oriented programming.

To exploit the parallelism of the self-contained machines of the leaf nodes therefore requires adopting the base model directly by programming the code for the node in the language of the base model. The model requires that all the processors in the base node execute a single function using a *single-function-multiple-data* (SFMD) model. The base node calculation is complete when all the processors return from the function call. During the execution of the base function the processors can do whatever is permitted in the chosen base language, with the exception of taking part in any operation that involves interaction with processors outside the base node. Thus, processors can interact with processors inside the base node, but not outside the node. Interaction between nodes can only be performed by the algorithmic skeleton used in the parent node.

The HSM data structures passed as arguments to the base function are shared over the leaf node's processors, which means that in a distributed memory architecture the processors may have to explicitly communicate with one another to access parts of the structure stored on another processor. The return HSM data structure must be created by the base function, and will also be distributed over the processors of the node. Details of creating and manipulating the HSM data structures in the base language are given in Chapters 5 and 6 where the interfaces between the HSM model and the base models are discussed in greater detail.

2.5 Summary

The HSM model offers a high level model of parallel programming which allows the programmer to express the structured parallelism inherent in an algorithm in an architecture-independent form. Later chapters will show that not all parallelism can be expressed clearly using algorithmic skeletons, and the model acknowledges this by allowing the programmer to express some parts of the solution in an architecture-dependent base language.

The model encourages a stepwise refinement process of program design and the separation of tasks into nodes which execute concurrently with a well defined pattern of communication between the nodes. The level of abstraction in the skeleton superstructure enables the programmer to concentrate on the *structure* of the parallelism and removes the need to explicitly consider the lower level details of communication and synchronisation which are required to implement the parallel structure. When the parallelism cannot be expressed clearly using the skeletons, the model allows the programmer to adopt directly the base model of programming.

The model uses a basic set of data structures, namely scalar variables, arrays and compound structures. The ability to nest the structures leads to a potentially powerful data structure hierarchy which can match the skeleton hierarchy. Chapter 3 will define the data structures in detail, while Chapter 4 will describe the algorithmic skeletons. Chapters 5 and 6 will then discuss using shared memory and distributed memory base models respectively.

Chapter 3

Data Types in the HSM

Contents

3.1	**Data Structure Declaration and Creation**	**26**
3.2	**The HSM Scalar Types**	**27**
3.3	**The HSM Array Type**	**29**
3.4	**The HSM Struct Type**	**36**
3.5	**Defining New Types**	**37**
3.6	**Other Data Types**	**37**

This chapter documents the data types available in the language of the Hierarchical Skeleton Model, and describes the operations permitted on structures of those types.

The data structures and operations defined in this chapter are not novel as such, and can be found in existing parallel languages. High Performance Fortran (HPF), for example, contains a wider set of structures and operations which are utilised in a SIMD programming style. The HSM language differs, however, in that the structures are chosen to meet the requirements of the algorithmic skeletons and the hierarchical partitioning of data. The operations are also designed with the hierarchical partitioning of the data in mind.

Section 3.1 describes how data structures are declared, and what it means for a data structure to be declared and created.

Sections 3.2, 3.3 and 3.4 describe the scalar, array and compound structure types respectively, and Section 3.5 shows how new types can be defined using the existing types as building blocks.

Lastly, Section 3.6 discusses some additional data types which have not been included in the HSM language.

3.1 Data Structure Declaration and Creation

The handling of data structures is simplified quite considerably by the adoption of a single thread of control flow through each SPSD node of the skeleton hierarchy and the implicit abstraction away from the individual processors.

Data structures are *declared* through a declaration statement in the HSM code with a syntax similar to that of the 'C' language. Unlike 'C', it is not possible to initialise a HSM variable when it is declared. This may be a slight inconvenience to the programmer.

> ***var-declaration***:
>
> > *type id-list* ';'
>
> ***id-list***:
>
> > *id* | *id* ',' *id-list*

For example:

```
int x, y;
array< int > m;
```

The data structure is only *created* when storage has been assigned to the structure for all its members. The distinction between created and uncreated structures is necessary when investigating how the HSM structures will be implemented. For example, a multi-dimensional array may have been declared as being of a specific type but may not have been created with a specific number of dimensions or elements yet. All the structures are created implicitly the first time they are assigned a value.

The data structures declared and created within a node of the hierarchy *belong* to the local group of processors represented by that node. Since each node executes the code of a single function, data structures passed as input parameters to that function are considered to be owned by the local group. The output data structure used to hold the result of the function call must be declared and created inside the node, but is otherwise treated like all the other locally declared and created structures. The only difference is that the local structures will cease to exist when the code for the node is finished, while the output structure is passed back to the parent node. The partitioning and inheritance of data structures is determined by the skeletons being used, and is discussed in greater detail in Chapter 4.

The main data types offered in the HSM language are scalar variables, multi-dimensional arrays, and compound structures. These will be described in Sections 3.2, 3.3 and 3.4 respectively. Two additional data types are also required:

type:

  ‘`void`’ | ‘`fnname`’

The **void** type is used in a function's argument list if the function takes no arguments or if an argument will be ignored. The latter usage may occur in the definitions of the body functions of the skeletons described in the next chapter, where the global argument may not be required. The **fnname** type is used in the skeleton function prototypes to specify the name of a function as an argument. For the programmer's convenience new types can also be defined using the **typedef** command described in Section 3.5.

The following sections will now discuss each of the data types and the operations permitted on variables of those types.

3.2 The HSM Scalar Types

The simplest data structure offered by the HSM is a single instance of a variable such as a boolean, an integer, a real number or a string.

type:

  ‘`bool`’ | ‘`int`’ | ‘`real`’ | ‘`string`’

For example:

bool b;
int i, j, k;
real x, y, z;
string s;

There is no explicit creation operation as the scalar data structures are implicitly created when they are first assigned a value.

Expressions are built from constants, variable identifiers and operators all with a matching type. For the convenience of the following descriptions, expressions are classified into four categories, one for each of the basic types: *bool*, *int*, *real* and *string*. It is the responsibility of the compiler to ensure that expressions are of the correct types, rather than trying to enforce this in the grammar, but the following description tries to make the types clearer by using the terms *bool-expr*, *int-expr*, *real-expr* and *string-expr* to indicate an expression of a specific type. The term *num-expr* is also used to indicate the use of either an *int-expr* or *real-expr*.

The simplest expression syntax is simply the value of an existing variable or the result of a function call, as already described in Section 2.2. An expression can also be built using parenthesis to override or emphasise the operator precedence:

expr:

variable
| *id* '(' ')'
| *id* '(' *expr-list* ')'
| '(' *expr* ')'

A boolean expression can be constructed using the basic equality, inequality and boolean operators:

expr:

'TRUE' | 'FALSE'
| '!' *bool-expr*
| *bool-expr* '||' *bool-expr*
| *bool-expr* '&&' *bool-expr*
| *expr* '==' *expr*
| *expr* '!=' *expr*
| *num-expr* '<' *num-expr*
| *num-expr* '<=' *num-expr*
| *num-expr* '>' *num-expr*
| *num-expr* '>=' *num-expr*

Integer and real constants are composed as expected:

int-const:

[0-9][0-9]*

real-const:

int-const '.' *int-const*

Basic arithmetic is permitted for integer and real numbers, and expressions can be cast from one type to the other:

expr:

int-const

| *real-const*
| *num-expr* '+' *num-expr*
| *num-expr* '-' *num-expr*
| *num-expr* '*' *num-expr*
| *num-expr* '/' *num-expr*
| '(' 'int' ')' *real-expr*
| '(' 'real' ')' *int-expr*

For example:

```
b = ( i <= j );
i = j * k;
x = (real)( i * j * k );
s = "Hello World";
```

3.3 The HSM Array Type

A multi-dimensional array is a structure of several dimensions, say d dimensions, where each dimension $\mathcal{D}_i$ contains a finite set of n_i discrete indices, numbered $0, 1, \ldots, n_i - 1$. The whole array structure contains $n = \Pi(n_i)$ elements with each element referenced using a unique d-tuple, say $t =< t_1, t_2, \ldots, t_d >$ where $0 \leq t_i < n_i$ represents the index within dimension $\mathcal{D}_i$. The notation $a_{<t_1, t_2, \ldots, t_d>}$ will be used to refer to an element of array a.

3.3.1 Array Declaration

The elements of the array can be any of the four basic scalar types introduced above, or can be themselves arrays as will be shown in the evaluation Chapters 9 and 10. The type of element must be specified when the array is declared, but not the size or number of dimensions.

type:

'array' '<' *type* '>'

For example:

```
array< int > a;
array< array< real > > r;
```

The array r has, as yet, an unknown number of dimensions, but each element of r is an array of real numbers. The dimensions and sizes of the element arrays are also not specified, and can be different for each element.

3.3.2 Array Creation

The number of dimensions of the array and the size of each dimension is specified when the array is created. This can be done explicitly using the assignment operation and the built-in function **array_create()**.

> ***expr***:
>
> 'array_create(' *int-expr* ',' *int-expr-list* ')'

The **array_create()** function takes a variable number of arguments, all integer, the first of which specifies the number of dimensions. The remaining arguments specify the sizes of each of the dimensions, with exactly one argument for each of the dimensions. The function creates an array of the correct number and size of dimensions which is then assigned to the array variable.

```
a = array_create( 3, 10, 20, 5 );
r = array_create( 1, 10 );
```

Using the declaration examples given in the previous section, the first example creates a three-dimensional array of integers with dimensions of size 10, 20 and 5 respectively, i.e. holding a total of 1000 integer elements. The second example creates a one-dimensional array of ten elements, where each element is an array of real numbers of unknown size and number of dimensions as they have not been created yet.

If any of the dimension sizes are zero the array will not have any elements, i.e. it will be a *null* array. Specifying a zero number of dimensions is not permitted as this defines a *point* which is effectively a scalar value.

3.3.3 Array Assignment

The creation of arrays described above uses the '=' operator to assign the result of a function call to an uncreated array. In general the assignment operator can also be used to assign arrays, sub-arrays or array elements from one variable to another. If an array being assigned to has not been created yet then it will be created with the same number of dimensions and size of each dimension as the source array. If the array has been created then the number and size of dimensions must match up on both sizes of the assignment.

```
array < int > b;
b = a;
```

In this example array b is created with the same number of dimensions and size of each dimension as array a and each of the elements in a is assigned to the corresponding element in b:

$$b_{<t_1,t_2,\ldots,t_d>} = a_{<t_1,t_2,\ldots,t_d>} \quad \forall t_1, t_2, \ldots, t_d$$

Individual elements of an array can be accessed and assigned using the standard 'C' language square bracket index notation. This is included in the grammar of the HSM language by extending the definition of a *variable*:

variable:

variable ref-operator

ref-operator:

'[' *int-expr* ']'

A simple example of indexing an array to assign an element is therefore:

```
b[0][0][0] = a[4][5][6];
```

i.e.:

$$b_{<0,0,0>} = a_{<4,5,6>}$$

A sub-array can be accessed or assigned to using the *range* operator [i..j]. This reduces the size of one dimension of the array expression, but does not reduce the number of dimensions. A second rule is added to the *ref-operator* to describe the range operator:

ref-operator:

'[' *int-expr* '..' *int-expr* ']'

The following example will create a three dimensional array with each dimension of size two, and also assign that array to a sub-array of b:

```
array < int > c;
c = a[0..1][0..1][0..1];
b[2..3][2..3][2..3] = c;
```

i.e.:

$$\left.\begin{array}{rcl} c_{<t_1,t_2,t_3>} & = & a_{<t_1,t_2,t_3>} \quad ; \\ b_{<t_1+2,t_2+2,t_3+2>} & = & c_{<t_1,t_2,t_3>} \quad ; \end{array}\right\} \quad 0 \leq t_1, t_2, t_3 < 2$$

The assignment of values to the sub-array of b could have been done in a single step by assigning the sub-array of a directly to the sub-array of b:

```
b[2..3][2..3][2..3] = a[0..1][0..1][0..1];
```

$$b_{<t_1+2,t_2+2,t_3+2>} = a_{<t_1,t_2,t_3>} \quad 0 \leq t_1, t_2, t_3 < 2$$

The grammar of the HSM language makes it possible to mix the index and range operators in a single array reference, but requires that there is exactly one range or index operator for each dimension of the array. Using a combination of the index and range operators makes it possible to assign to and from arrays with different numbers of dimensions by reducing the effective number of dimensions or the size of one or more of the dimensions. If the array being assigned to has not been created yet, the array will be implicitly created with the minimum number of dimensions and size of each dimension needed. The following example creates a one-dimensional array d by extracting a single dimension of a:

```
array < int > d;
d = a[0][0..5][0];
```

$$d_{<t>} = a_{<0,t,0>} \quad 0 \leq t < 6$$

Using the range operator does not alter the number of dimensions of the array expression, but each use of the index operator decreases the number of dimensions by one. The earlier example of accessing an element of the array can therefore be viewed as returning a zero-dimensional array, i.e. a *point* or scalar value.

3.3.4 Array Concatenation

An array or an array expression can be appended with another array to create a larger array using the array concatenation operator ++', but only in the top-most dimension, i.e. $\mathcal{D}_1$. Both arrays must have the same number of dimensions and the size of all the dimensions except $\mathcal{D}_1$ must be the same for both arrays.

expr:

array-expr '++' *array-expr*

This has the effect of performing the following assignments:

$$\begin{aligned} c_{<t_1,t_2,\ldots,t_d>} &= a_{<t_1,t_2,\ldots,t_d>} & 0 \leq t_1 < a_{n_1}, & \quad \forall t_2,\ldots,t_d \\ c_{<t_1,t_2,\ldots,t_d>} &= b_{<t_1-a_{n_1},t_2,\ldots,t_d>} & a_{n_1} \leq t_1 < a_{n_1} + b_{n_1}, & \quad \forall t_2,\ldots,t_d \end{aligned}$$

where a_{n_1} and b_{n_1} are the sizes of dimension $\mathcal{D}_1$ for arrays a and b respectively and c has the following dimension sizes:

$$\begin{aligned} c_{n_1} &= a_{n_1} + b_{n_1} \\ c_{n_i} &= a_{n_i} = b_{n_i} \qquad 2 \leq i \leq d \end{aligned}$$

The concatenation operator changes the size of one dimension but not the number of dimensions.

The following example creates a three dimensional array with dimensions of size 25, 20 and 5 respectively.

```
array < int > a, b, c;
a = array_create( 3, 10, 20, 5);
b = array_create( 3, 15, 20, 5);
c = a++b;
```

i.e.

$$
\begin{array}{lcll}
c_{<t_1,t_2,t_3>} & = & a_{<t_1,t_2,t_3>} & 0 \le t_1 < 10, \quad 0 \le t_2 < 20, \quad 0 \le t_3 < 5 \\
c_{<t_1,t_2,t_3>} & = & b_{<t_1-10,t_2,t_3>} & 10 \le t_1 < 25, \quad 0 \le t_2 < 20, \quad 0 \le t_3 < 5
\end{array}
$$

3.3.5 Array Querying

The size of the dimensions of an array can be found by calling the built-in function **array_size()**. The function takes the array as an argument and returns a one-dimensional array of integers containing the sizes of each dimension. Continuing the examples given above, the following code returns a one dimensional array of three integers:

```
array < int > s;
s = array_size(a);
```

If the array hasn't been created yet, the **array_size()** function will return an empty one-dimensional array. This implies that an uncreated array has zero dimensions.

Since the **array_size()** function returns an array, a sub-array or element can be extracted using the index and range operations as before. The grammar described so far only permits the operators to be used on variables, so the definition of *expr* needs to be extended. The separate extension of the *variable* definition is still needed to allow for assignment to elements or sub-arrays of an array.

expr:

expr ref-operator

The following example will assign the size of the third dimension of a to x:

```
int x;
x = array_size(a)[2];
```

The total number of dimensions can be found by taking the size of the one-dimensional array returned by the **array_size()** function:

x = **array_size(array_size**(a))[0];

3.3.6 Array Initialisation

§3.3.2 explained how arrays could be created, but not how the values of the member elements could be initialised. The simplest form of initialising the array involves setting all the members to the same value, which can be done when the array is created using the built-in function **array_create_init()** instead of **array_create()**.

expr:

‘`array_create_init(`’ *int-expr* ‘`,`’ *int-expr-list* ‘`,`’ *expr* ‘`)`’

The **array_create_init()** function takes the same variable number of arguments as the **array_create()** function, and one extra initialisation value argument specifying a single value. An array of the appropriate number and size of dimensions is returned after the given value has been assigned to each element of the array. E.g.:

```
array< array< real > > r;
r = array_create_init( 1, 10, array_create( 2, 5, 5 ) );
```

This will create a one-dimensional array of ten arrays, where each of the ten arrays is a two-dimensional array of real numbers with 5 rows and 5 columns each. In this example all the element arrays have the same number and size of dimensions, though this does not need to be the case. The following example will create a one-dimensional array of ten arrays where the element arrays have different numbers of dimensions:

```
r = array_create_init( 1, 5, array_create( 1, 5 ) )
    ++ array_create_init( 1, 5, array_create( 2, 5, 5 ) );
```

To initialise an array with different values at each of the elements will normally require mapping a function over an array, but the array being mapped over must be able to distinguish its elements. This is done using the built-in function **array_create_indexed()**:

expr:

‘`array_create_indexed(`’ *int-expr* ‘`,`’ *int-expr-list* ‘`)`’

The **array_create_indexed()** function takes the same variable number of arguments as the basic **array_create()** function. An array of the appropriate number and size of dimensions is returned where each element of the array is an *index-tuple*. The index-tuples are one-dimensional arrays of n integers, where n is the number of dimensions in the 'parent' array. The index-tuples are initialised to hold the dimension-indices of the elements, with the counting starting from zero:

```
int x, y;
array< array< int > > ind_array;
array< int > index;
ind_array = array_create_indexed(2, 20, 10);
index = ind_array[19][9];
```

In this example the variable index will hold a one-dimensional array of two integers, with the values 19 and 9 respectively.

It is necessary for the **array_create_indexed()** function to return an array of arrays as multi-dimensional arrays will require more than one integer to index each element. For one-dimensional arrays this means that each index-tuple is a one-dimensional array containing exactly one integer. A second function is therefore provided which returns an array of integers instead of one-tuples, but this only works for one-dimensional arrays:

expr:

  'vector_create_indexed(' *int-expr* ')'

The **vector_create_indexed()** function only takes one argument, giving the number of elements in the one-dimensional array. An array of the required size is then returned with element i of the array initialised to the integer value of i.

```
array< int > z;
z = vector_create_indexed( 10 );
```

This will create a one-dimensional array of 10 integers numbered 0, 1, ..., 9 respectively.

3.3.7 Array Transforming/Casting

§3.3.3 showed how a single sub-array can be extracted from an array, but sometimes it is also necessary to divide the array into many sub-arrays. This can be done using a cast-type operator:

expr:

‘(’ ‘`array<`’ *int-expr-list* ‘`>`’ ‘)’ *array-expr*

The cast takes one or more distinct integer arguments representing dimension numbers, and divides the array along those dimensions creating an array of arrays with the number of dimensions in the ‘parent’ array equal to the number of integer arguments. For example:

```
array< int > a;
array< array< int > b;
a = array_create( 3, 10, 20, 5);
b = (array<2>) a;
```

This will create the one-dimensional array b with 20 elements, each of which is a two-dimensional array of integers with 10 rows and 5 columns:

$$b_{<i>} = a_{<0..9,i,0..4>} \quad 0 \leq i < 20$$

3.4 The HSM Struct Type

The last of the three main data types is the compound structure type **struct** {}, similar to the **struct** used in ‘C’. This allows several different data structures to be grouped together under a single type:

type:

‘`struct`’ ‘{’ *var-declarations* ‘}’

For example:

```
struct { int x, y; real z; array< int > m; } c;
```

This will create a single data structure with two integer members named ‘x’ and ‘y’, a real member named ‘z’ and an array named ‘m’. The components can be accessed in the same way as for the ‘C’ language using the ‘dot’ dereferencing symbol:

ref-operator:

‘`.`’ *id*

For example:

```
c.x = 5;
c.y = c.x;
c.z = (real) c.y;
c.m = array_create( 2, c.x, c.y );
c.m[0][0] = c.m[3][2];
```

3.5 Defining New Types

For the convenience of the programmer, new types can be defined using the same syntax as 'C':

> ***type-definition***:
>
> > '`typedef`' *type id* '`;`'

For example:

> **typedef struct** { **int** i, j; } intpair;

As will be shown in the evaluation Chapters 8, 9, and 10, this is particularly useful for passing several data structures as arguments to skeletons where only a fixed number of data structures are permitted.

3.6 Other Data Types

This chapter has introduced the data types available in the HSM language. The scalar types form the foundation of a rich range of data structures which can be built using the array and compound structures. The ability to nest arrays within arrays allows for the creation of hierarchical data structures which are more suited to the structure of the problem to be solved than simple flat data structures. Chapters 7 through 10 illustrate how the data structure types are sufficient to implement solutions for more complicated examples.

Some of the skeletons described in the next chapter are more commonly applied to *lists* of elements. It is not necessary to create a new list type as the list operations can all be performed using one-dimensional arrays. In particular, the head of a list l will be the element with index zero, and lists can be concatenated together using the $++$ operator. Inserting an element into a list is more complicated, but can be done by concatenating three arrays together, where the middle array is a singleton array and the other two form the front and end of the old array.

A tree structure can be implemented as a series of nested arrays, but the depth of the nesting must be known in advance and all the branches of the tree must have the same depth. This is because the language does not permit recursive data structures. This is unfortunate as it prevents the dynamic creation of arbitrary depth arrays, but is a necessary result of the type definitions of the structures. The language could be extended to allow recursive structures by allowing compound structures to be self referencing, e.g.:

```
typedef struct new_type {
    int x;
    array< new_type > m;
} new_type;
```

The distribution of the recursive structure will be the same as that for the non-recursive compound structure, i.e. the members will be shared over the same processors as the compound structure itself. This leads to interesting questions as to how recursive structures are partitioned, but since the compound structures cannot be partitioned the recursive structures will be the same. This is not a problem as none of the skeletons explicitly manipulate compound structures, but it opens the possibilities new skeletons designed to manipulate trees, etc..

The realisation of the data structure types is discussed for shared memory and distributed memory target architectures in Chapters 5 and 6. Those chapters show that the data structures can be implemented efficiently to keep the overhead costs of partitioning the data structures down to a minimum.

Chapter 4

Skeletons in the HSM

Contents

4.1 The Skeleton Function Syntax 40
4.2 Data Structure Partitioning 41
4.3 The if Skeleton 42
4.4 The while Skeleton 43
4.5 The map Skeleton 44
4.6 The zip Skeleton 45
4.7 The reduce Skeleton 46
4.8 The divide Skeleton 48
4.9 The scan Skeleton 49
4.10 The filter Skeleton 50
4.11 The pipe Skeleton 51
4.12 Other Skeletons 53

Chapter 2 introduced the concepts underlying the Hierarchical Skeleton Model and explained how each node of the skeleton hierarchy represents a self contained parallel machine executing a single function expressed in the HSM language. The SPSD function consists of a sequence of statements which either assign or return data structure expressions. The expressions are built from basic data structure arithmetic, skeleton instances and calls to HSM or base functions. The syntax of the data structure arithmetic and the syntax for calling functions has already been described in Chapters 2 and 3. This chapter will describe the syntax and semantics of using the algorithmic skeletons, and Chapters 5 and 6 will describe the use of base language functions.

The first two skeletons in this chapter, the if and while skeletons, are not normally considered to be algorithmic skeletons as they do not encapsulate any parallelism or pattern of communication. The constructs do form an essential component of sequential algorithms and are therefore required in the HSM's SPSD programming model. There are no control constructs in the HSM language so the if and while constructs are implemented as skeletons.

The other skeletons that are discussed in this chapter are the map, zip, reduce, divide, scan, filter, and pipe skeletons.

4.1 The Skeleton Function Syntax

Each skeleton function takes one or more arguments as input and returns one data structure. The input arguments are either the names of the body functions 'fleshing-out' the skeleton, or expressions whose values will be passed as arguments to the body functions. An instance of a skeleton therefore looks the same as a function call, e.g.:

z = **map** (body_fn, x, y);

The syntax for the skeleton is therefore expressed as a HSM function prototype, e.g.:

array< *type*-β > **map** (
    fnname body_fn, *type*-γ global, **array**< *type*-α > local
);

where *type*-α, *type*-β and *type*-γ represent arbitrary HSM data types.

The skeletons all take an exact number of arguments, for example one function name and two input structures, and returns one output structure. The programmer can pass two or more data structures as one argument by creating a compound structure, using the **struct** keyword as described in Chapter 3, and passing the compound structure as the argument.

As mentioned in Chapter 2, the HSM body functions cannot modify the input arguments. If necessary, the programmer has to make an explicit local copy of the data structures and make modifications to the local copy. This ensures that only the data structures being assigned to can have their values changed, although it is possible to assign the results of a function to a data structure that was used as an input argument to that function.

4.2 Data Structure Partitioning

With the exception of the if, while and divide skeletons, the skeletons all partition the input data structure so that each partitioned segment of the structure can be operated on by one of the child nodes of the skeleton node. Since there is no communication between the child nodes the segments of the input data structure must be independent. This is done by making the input structure an **array**<> of elements. The array is partitioned into disjoint portions by splitting the array into its individual elements. An $n \times n$ array of integers will therefore be partitioned into n^2 separate elements. This allows each child node to perform calculations on the local element without interfering with the local elements of other nodes.

In the following sections, the semantics of the skeletons are defined using an extension of the mathematical notation introduced in Chapter 3, e.g.:

$$\mathbf{map}\,(b, g, l)_t = b\,(g, l_t) \qquad \forall t$$

where i represents an index-tuple of the array, and l_t represents an element of the (multi-dimensional) array l. The notation $l_{s..t}$ represents a sub-array of l which contains elements with tuples s up to and including t.[1] The sub-array $l_{0..t}$ represents a sub-array of all the elements up to and including the element with tuple t.

Specific elements of the array are accessed as before using the tuple notation, e.g. $l_{<i>}$ and $l_{<i,j>}$ for one- and two-dimensional arrays respectively. The size of dimension j of array l is expressed as $|l|_j$, or just $|l|$ if the array is known to be one-dimensional. The number of dimensions is expressed as $\|l\|$, i.e. taking the size of the array holding the sizes of each of the dimensions.

Partitioning the data structures in this way guarantees consistency of the data within a skeleton node, but can cause extra work if the data does not need to be modified. For example, in the map skeleton the function calculation for each element of an array may rely on some unchanging *global* data structure. It is possible to create a second array of the same length as the data array and 'insert' a copy of the global data into each array element, but this seems unnecessarily complicated for the programmer. The solution chosen here is that the input data is separated into an array of 'local' data elements and a single 'global' data structure, such that the body functions will always be passed the same global data but different local data on each invocation.

[1]The tuples are ordered using a lexicographical ordering, which will be formalised in Chapter 6 for the `HSM_index` structure.

As mentioned previously the if and while skeletons differ from the other skeletons by not partitioning the skeleton hierarchy of the program so that there is no local structure to partition either. For the while skeleton a distinction is still made between the global and local arguments, so that the calculation is performed *on* the data of the local argument with the use of additional information in the global arguments. The local argument represents the *context* of the function which may change. There is no need to make a distinction for the if skeleton as there is no body function for the context or global data to be passed to. The arguments are simply expressions which can use any of the data structures local to the function.

4.3 The if Skeleton

The if skeleton lets a program perform a conditional branch in the calculation and calculate one expression instead of another. The skeleton takes three expressions as its arguments, where the first expression is of type **bool** and the remaining two expressions are both of type α. The skeleton returns a data structure of type α. The skeleton has the following function prototype in the HSM language:

```
type-α  if (
        bool test_expr,   type-α  then_expr,   type-α  else_expr
 );
```

The skeleton evaluates the *test-expr*, and if the result is **TRUE** the *then-expr* is evaluated and returned. Otherwise the *else-expr* is evaluated and returned:

$$\mathbf{if}(t, x, y) = \begin{cases} x & \text{iff} \quad t = \ \mathbf{TRUE} \\ y & \text{iff} \quad t = \ \mathbf{FALSE} \end{cases}$$

The following example will return the positive difference between two integers:

```
int x, y, z;
z = if ( x<y, y−x, x−y );
```

There is no parallelism inherent in the if skeleton itself, so any parallelism must be contained within the calculations of the expressions given as arguments to the skeleton. The compiler should be careful to use a form of lazy evaluation to prevent both type α input arguments being calculated before calling the if skeleton.

4.4 The while Skeleton

The while skeleton lets a program iterate through a function continuously for as long as a particular condition holds true. The skeleton takes as its arguments two function names, a global data structure of type γ and a local data structure of type α. Both the named functions take two arguments, of types γ and α respectively. The first function returns a value of type **bool**, while the second returns a value of type α. The skeleton returns a single data structure of type α.

```
type-α  while (
        fnname test_fn,   fnname body_fn,
        type-γ  global,   type-α  local
 );
 bool test_fn (
        type-γ  global,   type-α  local
 );
type-α  body_fn (
        type-γ  global,   type-α  local
 );
```

The skeleton calls *test-fn* with the global and local arguments, and if the result is **TRUE** it calls the *body-fn* with the same arguments, storing the result in a new local structure of type α. The process of calling the test function, and possibly the body function, is then repeated using the original global argument and the new local argument until the test function returns false. At that point the new local argument is returned as the result of the while skeleton.

$$\mathbf{while}(t,b,g,l) = \begin{cases} \mathbf{while}\,(t,b,g,b\,(g,l)) & \text{iff} \quad t(g,l) = \ \mathbf{TRUE} \\ l & \text{iff} \quad t(g,l) = \ \mathbf{FALSE} \end{cases}$$

The following example will call the *body-fn* exactly 10 times, and return the value '10':

```
bool test_fn( int a, int b ) {
    return ( b<a );
}
int body_fn( int a, int b ) {
    return ( b+1 );
}
int x;
x = while ( test_fn, body_fn, 10, 0 );
```

As with the if skeleton there is no parallelism inherent in the while skeleton. Any parallelism must be contained within the *test-fn* and *body-fn*s.

4.5 The map Skeleton

The map skeleton applies a function to every member of an array to create a new array. The skeleton takes one function name, a global data structure of type γ, and an array of local elements of type α. The named function takes two arguments, of types γ and α respectively, and return a value of type β. The skeleton returns an array of type β with the same number of dimensions and size as the input local array.

```
array< type-β > map (
        fnname body_fn,   type-γ global,   array< type-α > local
);
type-β body_fn (
        type-γ global,   type-α local
);
```

The skeleton applies the *body-fn* to each element of the array and stores the result in the corresponding element of the return array:

$$\mathbf{map}\,(b, g, l)_t = b\,(g, l_t) \qquad \forall t$$

The following example will generate a two-dimensional array of integers where the value of each element is the product of the row and column indices:[2]

```
int body_fn( void global, array< int > tuple ) {
    return ( tuple[0]*tuple[1] );
}
array< int > x;
array< array< int >> ind;
ind = array_create_indexed( 2, 10, 10 );
x = map ( body_fn, NULL, ind );
```

Since each function calculation is completely independent of the other calculations, the calculations can be performed concurrently. The local array is partitioned into its elements, and the HSM node is split into several child nodes. If

[2]The body_fn() ignores the global argument, so the map skeleton passes the NULL value for the global variable.

the number of elements in the array is greater than the number of processors in the node then each child node will contain exactly one processor and be assigned one or more elements of the array. The single process node will then execute the *body-fn* for each of the type α elements that it has been assigned and create the corresponding type β elements. The assignment of the local elements can be done with a simple static partition of the work or a task-farming approach can be taken to perform some load balancing.

If the number of elements in the array is less than the number of processors then the node will be partitioned into one child node for each element of the array. The child node will inherit one element of type α from the local array and execute the *body-fn* over the local node to create one new element of type β.

On completion of the work within each child node, the nodes rejoin to form the single parent node again. Each child node passes back the locally calculated β elements to form a single array of the same number of dimensions and size as the input array.

4.6 The zip Skeleton

The zip skeleton takes two arrays of equal size and applies a function to 'join' the two arrays together to create a new array. The skeleton takes one function name, one global data structure of type γ, and two arrays of local data structures of types α and β respectively. The named function takes three arguments, of types γ, α and β respectively, and returns a value of type δ. The skeleton returns an array of type δ with the same number of dimensions and size as the local input arrays.

array$<$ *type-*δ $>$ **zip** (
    fnname body_fn, *type-*γ global,
    array$<$ *type-*α $>$ local_a, **array**$<$ *type-*β $>$ local_b
);
type-δ body_fn (
    type-γ global, *type-*α local_a, *type-*β local_b
);

The skeleton applies the *body-fn* to each pair of elements from the local arrays, i.e. an element from the array ' local_a' and the corresponding element from the array ' local_b', and stores the result in the corresponding element of the return array:

$$\mathbf{zip}\,(b, g, la, lb)_t = b\,(g, la_t, lb_t) \qquad \forall t$$

The following example will generate a one-dimensional array of integers where the value of each element is the square of its index:

```
int body_fn( void global, int a, int b ) {
    return ( a*b );
}
array< int > x, y;
x = vector_create_indexed( 10 );
y = zip ( body_fn, NULL, x, x );
```

The parallelism of the zip skeleton is exactly the same as that of the map skeleton. The only difference is that two arrays need to be partitioned instead of just one. If a node is assigned an element of the first array it will also be assigned the corresponding element of the second array.

4.7 The reduce Skeleton

The reduce skeleton takes an array of elements and reduces all the elements to a single element using the given binary operation. The skeleton takes one function name, one global data structure of type γ, and an array of local elements of type α. The named function takes three arguments, of types γ, α and α respectively, and returns a single value of type α. The skeleton returns a single value of type α.

```
type-α  reduce (
        fnname body_fn,   type-γ  global,   array< type-α >  local
 );
type-α  body_fn (
        type-γ  global,   type-α  local_a,   type-α  local_b
 );
```

An array with just a single element cannot be reduced further, so the value of the element is simply returned. A one-dimensional array is reduced by splitting the array in half, reducing both halves and reducing the results of the two reduces. If the array has more than one dimension the lower dimensions are first reduced to single values, and then those values are reduced. This can be expressed using the array casting/transforming operation $(\textbf{array}{<}1{>})x$ which creates a one-dimensional array of arrays as explained in §3.3.7.

$$\textbf{reduce}(b,g,l) = \begin{cases} \textbf{reduce}\Big(b, g,\ (\textbf{array}{<}1{>})\ l\Big) \\ \qquad\qquad \text{iff} \quad \|l\| > 1 \\ \\ b\left(g,\ \textbf{reduce}\left(b, g, l_{<0..\lceil\frac{|l|}{2}\rceil-1>}\right),\ \textbf{reduce}\left(b, g, l_{<\lceil\frac{|l|}{2}\rceil..|l|-1>}\right)\right) \\ \qquad\qquad \text{iff} \quad \|l\| = 1 \quad \text{and} \quad |l| > 1 \\ \\ l_{<0>} \\ \qquad\qquad \text{iff} \quad \|l\| = 1 \quad \text{and} \quad |l| = 1 \end{cases}$$

The following example takes a one-dimensional array of integers and returns the element with the maximum value.

```
int max_fn( void global, int a, int b ) {
    return ( if ( a>b, a, b ) );
}
int x;
array< int > v;
v = vector_create_indexed( 10 );
x = reduce( max_fn, NULL, v );
```

The recursive definition given allows the reduction of an array to be computed by reducing both halves of the array concurrently and reducing the results of those reductions. This suggests a recursive halving of the array and the skeleton node. The base case is reached when either the array contains only a single element or the skeleton node contains only one processor. If the array contains more than one element the single processor nodes perform the reduction of the locally assigned elements sequentially.

Once the local elements have been reduced to a single element, the work of the node is finished. The node rejoins with the other child node back into the common parent node and passes back its reduced element. The two reduced elements, one from each child node, are then reduced over the larger parent node. The rejoining of child nodes and further reductions continue until the single node remains in which the skeleton was originally called.

The recursive halving of the array implies the creation of two arrays at each step. This is not necessary as the input array cannot be modified and the two sub-arrays will be no different from the original array except in size and displacement. Each node has to create a local element of type α to hold the reduced value which will be passed back to the parent node. The local elements are deleted once they have been reduced further with the special case that the base level elements, which are part of the original array, are not deleted.

4.8 The divide Skeleton

The divide skeleton takes a single element and returns a one-dimensional array of elements created by repeatedly dividing the original element using a specified body function for as long as the other specified body function indicates that division should continue. The skeleton takes two function names, a global data structure of type γ, and a local element of type α. The named functions take two arguments, of types γ and α respectively. The first function returns a value of type **bool**, and the second an array of values of type α. The skeleton returns a one-dimensional array of type α whose size is dependent on the number of divisions that were carried out.

```
array< type-α > divide (
      fnname test_fn,   fnname divide_fn,
      type-γ global,   type-α local
);
bool test_fn (
      type-γ global,   type-α local
);
array< type-α > divide_fn (
      type-γ global,   type-α local
);
```

The skeleton calls *test-fn* with the global and local arguments, and if the result is **TRUE** the local element is divided further by calling the *divide-fn* with the same arguments. The *divide-fn* returns an array of new local elements, over which the divide skeleton is again mapped. The array of arrays of type α is then reduced using the array concatenation operator $++$ into a single one-dimensional array of type α. The recursion stops when the *test-fn* returns **FALSE**, i.e. there should be no more divisions, and the skeleton returns the local element as the single element of a one-dimensional array.

$$\mathbf{divide}(t,b,g,l) \;=\; \begin{cases} \mathbf{reduce}\Big(\,{+}{+},\ \mathbf{NULL},\ \mathbf{map}\,(\mathit{map\text{-}fn}, g, b\,(g,l))\Big) \\ \qquad\quad \text{iff} \quad t(g,l) = \ \mathbf{TRUE} \\ \\ \mathbf{array\_create\_init}\,(1,1,l) \\ \qquad\quad \text{iff} \quad t(g,l) = \ \mathbf{FALSE} \end{cases}$$

$$\mathit{map\text{-}fn}(g,l) \;=\; \mathbf{divide}\,(t,b,g,l)$$

The following example takes an integer and divides it into three numbers, with the first number approximately twice the size of the other two. The three numbers generated are then also divided until each of the numbers is less than 16. The result is a one-dimensional array of numbers whose total is the original number.

```
bool test_fn( int a, int b ) {
    return ( b>a );
}
array< int > test_fn( int a, int b ) {
    array< int > ret;
    ret = array_create( 1, 3 );
    ret[0] = b/2;
    ret[1] = (b−ret[0])/2;
    ret[2] = b−(ret[0]+ret[1]);
    return ( ret );
}
array< int > z;
z = divide( test_fn, divide_fn, 16, 65535 );
```

The divide skeleton generates a recursive splitting of the hierarchy similar to that used in reverse by the reduce skeleton. The divide skeleton differs, however, in allowing each division to generate an arbitrary number of branches instead of just two. The parallelism inherent in the divide skeleton is similar in that each division partitions the node but the work is performed before partitioning further rather than after, as is the case for the reduce skeleton. The base case is only reached when the *test-fn* returns false. If the *test-fn* returns true when the skeleton node contains just one processor the node must perform the the divisions sequentially.

4.9 The scan Skeleton

The scan skeleton is similar to the reduce skeleton in that it involves reducing an array of results to a single value, but a whole array of reduced values is returned.

Element i of the array is the result of reducing the first i elements of the input array, i.e. it performs a *parallel prefix* computation.

The skeleton takes one function name, one global data structure of type γ, and an array of local elements of type α. The function named takes three arguments, of of types γ, α and α respectively, and returns a single value of type α. The skeleton returns an array of values of type α with the same number of dimensions and size as the input local array.

```
array< type-α > scan (
        fnname body_fn,   type-γ  global,   array< type-α > local
);
type-α  body_fn (
        type-γ  global,  type-α  local_a,  type-α  local_b
);
```

The result of the scan skeleton can be defined as follows:

$$\mathbf{scan}(b, g, l)_i = \ \mathbf{reduce}\,(b, g, l_{0..i}) \qquad \forall i$$

The following example takes a one-dimensional array of integers and returns a one-dimensional array holding the cumulative sum of the elements.

```
int sum_fn( void global, int a, int b ) {
    return ( a+b );
}
array< int > v, w;
v = vector_create_indexed( 10 );
w = scan( sum_fn, NULL, v );
```

4.10 The filter Skeleton

The filter skeleton takes an array and applies a test function to each of the elements. All the elements which pass the boolean test are returned in a one-dimensional output array. The skeleton takes one function name, a global data structure of type γ and a local array of type α as arguments. The function named takes two arguments, of types γ and α respectively, and returns a value of type **bool**. The skeleton returns a one-dimensional array of type α.

```
array< type-α > filter (
        fnname test_fn,   type-γ  global,   array< type-α > local
```

```
);
bool test_fn (
      type-γ global,  type-α local
);
```

The skeleton performs a map over the input array with the map function designed to return a singleton array holding the element if the *test-fn* is successful, and an empty array if the *test-fn* is not successful. This array of singleton and empty arrays is then reduced to one array of type α using the array concatenation operation to remove the empty elements.

$$
\begin{aligned}
\textbf{filter}(t,g,l) &= \textbf{reduce}\Big(\,{+}{+},\ \textbf{NULL},\ \textbf{map}\,(\textit{map-fn},g,l)\Big) \\
\textit{map-fn}(g,l) &= \begin{cases} \textbf{array\_create\_init}\,(1,1,l) \\ \qquad\quad \text{iff} \quad t(g,l) = \ \textbf{TRUE} \\ \\ \textbf{array\_create}\,(1,0) \\ \qquad\quad \text{iff} \quad t(g,l) = \ \textbf{FALSE} \end{cases}
\end{aligned}
$$

The following example takes an array of integers and filters out all the elements with odd values.

```
bool iseven_fn( void global, int a ) {
    return ( a==2*(a/2) );
}
array< int > x, y;
x = vector_create_indexed( 10 );
y = filter( iseven_fn, NULL, x );
```

4.11 The pipe Skeleton

The pipe skeleton takes an array of functions and an array of elements and successively applies each of the given functions to each element in turn. In so doing it produces an array of output elements with a one-to-one mapping with the input elements.

The skeleton takes an array of function names, an array of global data structures of type γ and an array of local elements of type α_0. The functions named take two arguments, of types γ and α_i respectively, and return an element of type α_{i+1}. The skeleton returns an array of type α_n, where n is the number of elements in the array of function names.

```
array< type-α_n > pipe (
        array< fnname > test_fns,
        array< type-γ > global,
        array< type-α_0 > local
);
type-α_{i+1} test_fns_i (
        type-γ global, type-α_i local
);
```

The skeleton operation can be pictured as a pipeline with each stage representing one function and the input elements flowing down the pipe with the output of one stage (function) being used as the input to the next. An array of global arguments is used instead of just a single value, so that each distinct stage in the pipeline can inherit a different global argument. This is useful to identify, for example, the index of each of the stages.

$$\mathbf{pipe}(b, g, l)_t = b_{n-1}\left(g_{n-1}, b_{n-2}\left(g_{n-2}, \ldots b_1\left(g_1, b_0\left(g_0, l_t\right)\right)\ldots\right)\right) \quad \forall t, \quad n = |b| = |g|$$

The following example takes an array of 10 integers and at each stage multiples the value by the index of that stage, with the exception of stage 0 which simply returns the local value.

```
int id_fn( void g, int l ) {
    return ( l );
}
int mult_fn( int g, int l ) {
    return ( g*l );
}
array< fnname > stage_fns;
array< int > g, y;
stage_fns = array_create_init( 1,1,id_fn )++array_create_init( 1,4,mult_fn );
g = vector_create_indexed( 5 );
x = vector_create_indexed( 10 );
y = pipe( stage_fns, g, x );
```

The parallelism inherent in the pipe skeleton can be exploited by overlapping the computations of each of the stages. The skeleton node is partitioned into one child node for each stage of the pipeline, with the arrays of function names and global data partitioned with one function name and one global element per child node. The input array of type α_0 is inherited one element at a time by the first stage of the pipeline. After each calculation, stage i passes its result of type α_{i+1} to stage $i+1$. The final stage, $n-1$, stores the results of type α_n and passes them back to the parent node on completion of the pipeline.

4.12 Other Skeletons

This chapter has described a range of algorithmic skeletons, but other researchers in the field of algorithmic skeletons have suggested different or additional skeletons such as farm, divide-and-conquer, and geometric.

The farm, or *task-farm* skeleton, has not been included as it is, in our opinion, identical to the map skeleton. There may be perceived differences to how a compiler would implement the two, but the functionality is the same.

The divide-and-conquer skeleton has also not been included as it can be composed more flexibly using the separate divide and reduce skeletons, with a possible map skeleton between the two to perform the base case of the recursion.

The geometric skeleton is a more complicated skeleton to consider than the other skeletons in that it provides a limited form of SIMD computation. The skeleton is normally used in image manipulation algorithms where a function is mapped over each pixel of the image and the calculation may depend on the values of the neighbouring pixels. A possible way to include this skeleton in the HSM model would be in a map-with-halo skeleton which, like the map skeleton, operates on a multi-dimensional array, but the body function takes a sub-array of the local array as input instead of just a single element:

array< *type-β* > **map** (
    fnname body_fn, *type-γ* global, **array**< *type-α* > local
);
type-β body_fn (
    type-γ global, **array**< *type-α* > local
);

The skeleton was not included in the HSM model due to the problems in specifying the size of the 'halo' required round each of the elements, and also how the input sub-array to the body function would need to be offset so that the origin of the sub-array is the pixel in the centre.

There is no reason why the map-with-halo skeleton could not be included in the HSM once these issues were resolved. Without the skeleton the same functionality can be achieved by:

(a) Passing the array as both the global parameter and the local parameter.
or

(b) Using the ad hoc parallelism of the base model to explicitly pass the halos between the processors.

Option (b) is likely to be the more efficient solution, but it requires the programmer to expend effort coding up the required base level parallelism.

Chapter 5

A Shared Memory Target Architecture for the HSM

Contents

5.1 The HSM Node Hierarchy 55
5.2 The HSM Data Structures 56
5.3 The HSM/Fork95++ Interface 57
5.4 A HSM/Fork95++ Example 59

This chapter will discuss the issues involved in realising the HSM model on target architectures which use a shared memory model, and describe the HSM interface for a shared memory base model programmed using the Fork95++ language.

An implementation of the model on a shared memory machine was beyond the scope of the thesis, but it is necessary to consider how the HSM skeleton hierarchy and data structures would be implemented so that the interface with the base language can be defined. Section 5.1 discusses how the machine can be partitioned to form the skeleton nodes of the HSM model, while Section 5.2 describes how the HSM data structures may be implemented. Lastly, Section 5.3 defines the HSM data structure operations available in the Fork95++ base code that forms the interface to the HSM model, and Section 5.4 gives a simple example of the use of the base language.

5.1 The HSM Node Hierarchy

The shared memory programming model and the Fork95++ language were introduced in Chapter 1. Each node of the HSM hierarchy is a group of processors and memory resources forming a self-contained shared memory machine. This allows a group of processors to create data structures to be shared over the local node.

In the Fork95++ language groups are identified by the shared variable ' `@`' which is common to the group, and the processors within the group each have a private variable ' `$`' the *rank* of the processor within the local group. The HSM nodes, and the processors within them, can therefore be distinguished.

A node is partitioned using the **fork**{} construct, which partitions the processors and allocates a *heap* of shared memory for the group. The data structures that exist in the shared memory of the parent node are not partitioned, and all the child nodes have potential read and write access. This leads to the possibility of write-conflicts between nodes which the HSM model tries to prevent by stipulating that structures must be passed *explicitly* to the child nodes and that these structures cannot be modified. The shared memory machine using the Fork95++ language is similar to the shared variant of the Hierarchical-PRAM, while the HSM model is more similar to the private H-PRAM variant. It is therefore the responsibility of the HSM compiler to enforce the restricted scope of the new skeleton nodes and the read-only restrictions on the data structures passed as arguments.

5.2 The HSM Data Structures

As was mentioned in Chapter 1, the HSM model uses parallel rather than distributed data structures. The Fork95++ language is therefore augmented with the 'parallel' data structure types used in the HSM superstructure, i.e. the scalar variables, multi-dimensional arrays and compound structures. Since the memory of the machine is shared, the HSM data structures can be located anywhere in the shared memory. For the scalar types, a block of shared memory is allocated to hold one element of the appropriate Fork95++ type. For the **array**<> type, the allocation must also include space to hold the number and sizes of the dimensions of the array. Lastly, for the **struct** type a block of shared memory is allocated to hold all the member elements of the compound structure.

If 'distributed' data structure types were used instead of the 'parallel' types the Fork95++ language would not need to be augmented. The base programmer would simply use the basic nested data structures available in the Fork95++ language. The disadvantages of using distributed structures are, however, that:

(a) it becomes unclear which of the data structures in the base model have been inherited from the HSM superstructure

(b) querying the size and number of dimensions of an array will have to be handled separately

Using the 'parallel' HSM data structures helps keep the interface between the base nodes and the skeleton superstructure clean, and provides a suitable class for the few attendant functions.

5.3 The HSM/Fork95++ Interface

The HSM/Fork95++ interface gives the Fork95++ programmer access to the HSM data structures created within the skeleton superstructure. Unlike the distributed memory interface described in the next chapter, the concept of element 'locality' is not important, so the elements are all considered to be 'local'.

As already mentioned, the HSM data types are given separate names despite the overlap between the HSM types and the Fork95++ types. This separates the HSM data structures from the base node structures and also allows the HSM/Fork95++ types to be implicitly specified as shared variables. The HSM **void** type is also defined, as it may be used as the type of a global argument in some of the base functions.

```
typedef HSM_bool sh int;
typedef HSM_int sh int;
typedef HSM_real sh double;
typedef HSM_string sh char*;
typedef HSM_void sh int;
```

E.g.:

```
HSM_int x;
HSM_real y;
```

The HSM multi-dimensional arrays are defined using the following C++ class:

```
template <class Type>
class HSM_array {
  public:
    HSM_array<Type> ();                          // Top level constructor
    sync create( sh int dims, ... );             // Create the array
    async int ndims( );                          // Get the number of dimensions
    async int size( int i );                     // Get the size of a dimension
    async Type& operator()( int i, ... );   // Get an element from the array
}
```

The value of an element in the array is returned using a round bracket notation where the comma separated arguments between the brackets specify the indices of the element in each of the dimensions. It would have been preferable to use the square bracket notation of the 'C' language, with one index per pair of square brackets, but the C++ language cannot handle arrays of arbitrary numbers of dimensions using this style. The round brackets were used instead of square brackets to make it clear that a different syntax is being employed.

```
sh int x;
HSM_array<HSM_int> a;
a.create( 2, 10, 10 );
a(5,5) = x;
x = a.size( 2 );
```

It should be noted that the ndims(), size() and array access functions are all asynchronous and that they return private values rather than shared values. Assignment of the private return value to a shared variable will therefore involve the local write-conflict resolution strategy. Defining the functions as being asynchronous allows the processors of the leaf node to operate independently from one another. The create() operation, on the other hand, is a synchronous function and must be called concurrently by every processor in the node.

Lastly, the HSM compound structures are created using the **struct** construct, and the elements can be accessed using the 'C' dereferencing style. New types which are defined in the HSM program are made available to the base programmer by prefixing the type name with HSM_. For example, with a HSM language **typedef** of:

```
typedef struct { int x; int y; } intpair;
```

the Fork95++ types will be HSM_intpair:

```
HSM_intpair p;
p.x = p.y;
```

These operations are all that are needed to create and manipulate the HSM arrays, as will be shown in Chapters 8 to 10.

5.4 A HSM/Fork95++ Example

The following HSM example code maps a base function over an array of arrays:

```
basefn int fork_body_fn( void global, array< int > localarray );
void main( void ) {
    array< array< int >> x;
    array< int > y;
    x = array_create_init( 1, 10, vector_create_indexed( 10 ) );
    y = map( fork_body_fn, NULL, x );
}
```

The base function, expressed in Fork95++, uses the PRAM multi-prefix instruction to sum the elements of the array:

```
HSM_int fork_body_fn(
    HSM_void global,
    HSM_array< HSM_int > localarray
) {
    pr int i;
    HSM_int sum;
    for ( i=$; i<localarray.size(0); i+=groupsize() ) {
        mpadd( &sum, localarray( i ) );
    }
    return( sum );
}
```

The HSM integer variable ‘ sum’ is used as if it were just another normal shared integer, and a private variable i is used to loop through all the elements of the array accepting that there are more array elements than there are processors. The **groupsize()** function returns the size of the local group, as it would in a normal Fork95++ program.

Chapter 6

A Distributed Memory Target Architecture for the HSM

Contents

6.1 The HSM Node Hierarchy 61
6.2 The HSM Data Structures 62
6.3 The HSM/C++/MPI Interface 70
6.4 A HSM/C++/MPI Example 75

This chapter will discuss the issues involved in realising the HSM model on target architectures which use a distributed memory model, and describe the HSM interface for a network of processors with local memory programmed with the C++ language and using the MPI message passing library to communicate between processors.

As with the shared memory target architecture discussed in the previous chapter, an implementation of the model on a distributed memory machine was beyond the scope of the thesis, but it is necessary to consider how the HSM skeleton hierarchy and data structures would be implemented so that the interface with the base language can be defined. Section 6.1 discusses how the network of processors can be partitioned into smaller networks of processors to form the skeleton nodes of the HSM model, while Section 6.2 describes how the HSM data structures may be implemented. Lastly, Section 6.3 defines the HSM data structure operations available in the C++/MPI base code that forms the interface to the HSM model, and Section 6.4 gives a simple example of the use of the base language.

6.1 The HSM Node Hierarchy

With the HSM model implemented on top of a distributed memory architecture, each skeleton node in the hierarchy consists of one or more processor/memory pairs. Each processor executes sequential code, written in the C++ language, and communicates with the other processors in the node using the MPI message passing library.

A group of processors in the C++/MPI model is represented by an MPI *communication group*, and every processor has a unique rank within each group. Communication between processors in a group is performed using *intra-group communicators*, such that a processor using a specific group communicator will only be able to communicate with other processors in the same group. The machine starts off as a single group with the single communicator MPI_COMM_WORLD.

The HSM model requires the base nodes to use a *single-function-multiple-data* model, which can be implemented by making all the processors in a single node execute the same function. Therefore with a leaf node of n processors there will be n instances of the same function being executed. The processors all share the same group communicator and so have a unique identifying rank within the local group.

Since the architecture uses distributed memory, the memory resources of each node in the hierarchy are the sum of the memory resources of each of the processors in the node. The data structures 'owned' by a node are shared over the member processors. Each processor has exclusive control over the data elements stored locally, and can communicate with the other processors in the node to access other elements of the HSM parallel data structures.

A skeleton node can be partitioned into smaller nodes by *splitting* the communicator. This is done using the MPI_Comm_split() function:

```
int MPI_Comm_split ( MPI_Comm parent_comm, int colour, int key,
                     MPI_Comm *comm_out )
```

There will be one *colour* for each child node, and the *key* is used to specify an ordering of the processors within that child node. The function takes the communicator for the parent node and creates separate communicators for each of the child nodes.

The new communicators partition the processors into the appropriate child nodes, but no partitioning of the memory resources is carried out. Each processor retains the data which was stored locally, and can communicate with the other processors in the child node to access other elements. For a child node to inherit a HSM data structure therefore requires ensuring that all the data structure elements are distributed over the processors of the child node. The communicator variable HSM_COMM_WORLD will be used to represent the communicator for the local leaf node.

6.2 The HSM Data Structures

As with the HSM/Fork95++ base model described in the previous chapter the HSM/C++/MPI base model uses parallel rather than distributed data structures. The C++/MPI environment is therefore augmented with the 'parallel' data structure types used in the HSM superstructure. As mentioned in Chapter 1 this has the appearance of a partial-shared memory environment in that there is effectively only one data structure spread over all the processors, but there is no shared memory or virtual shared memory. The processors must still communicate with one another to access data held by other processors.

Unlike the shared memory target architecture, the storage of a HSM data structure is no longer a simple case of just allocating storage space for each of the elements. For the distributed memory target architecture it is necessary to specify the *distribution* of the parallel HSM data structures so that every processor knows the size of the structure, over which processors it is distributed, and where every data element is stored. The data structures should be distributed as evenly as possible, and the distribution strategy should encourage low overheads when the data structures are partitioned and inherited by child nodes.

When a HSM structure is created the number and size of dimensions are specified. Each processor in the node needs to know this information and it is reasonable to replicate this information over every processor. All of the HSM data types will be distributed over the processors in the node, so the processors also need to know on which processors each data element will be located.

The HSM scalar structures cannot be partitioned further, so the distribution of a scalar structure over a group of processors will place the scalar element on a single processor, with the other processors in the group knowing where it is located. For the nested **struct** and **array<>** structures the elements will have their own distributions over a *range* of processors with the base case of the scalar structure distribution. The nested structures therefore only need to hold the distribution ranges of each of the member data structures, but indirectly they hold the location details of all the basic scalar elements.

There are three possible schemes which provide individual processors with the necessary location information:

(i.) Rigid distribution strategy — All data distributions use a single fixed distribution strategy. A processor can calculate the location of a specific data element without the cooperation of any other processor.

(ii.) Common ad hoc distribution strategy — Each processor knows the location of every other data element. A processor can find the location of any specific element without the cooperation of any other processor.

(iii.) Blind ad hoc distribution strategy — Each processor knows which data elements are stored locally but knows nothing about the location of any other data elements. A processor has to communicate with all the other processors to find the location of a specific data element not stored locally.

The Rigid distribution strategy has the advantage that two data structures of the same size and shape will always have the same distribution, but is inflexible as assignment between structures of different sizes will involve potentially expensive re-distributions.

The common ad hoc distribution strategy gets over this restriction by allowing *any* distribution of the data elements. This gives the programmer greater control over the data structure and hence over the efficiency of some algorithms. The disadvantage is that the information required to specify the data element locations can get quite large as, unlike the elements themselves, the location information must be duplicated over every processor.

In the blind ad hoc distribution strategy, the storage space required to hold the location information on each processor is proportional to the number of data elements stored locally, but there will be a communication overhead when finding out the location of non-local elements. The HSM/C++/MPI examples given in Chapters 8 and 10 show that this information is often needed in an algorithm.

The Common ad hoc Distribution Strategy was selected for this thesis to provide programming flexibility and potential performance efficiency, at the cost of increased storage requirements.

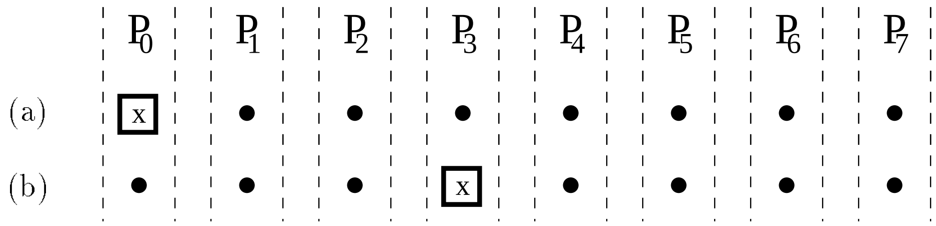

Figure 6.1: Example distributions of a scalar variable over a group of eight processors; with the scalar variable located (a) on the root processor, and (b) on the fourth processor.

6.2.1 The HSM Scalar Types

The HSM scalar type is the simplest of the HSM data structures to consider. Since the structure contains just a single element the element will be stored on exactly one of the processors in the group. This is still referred to as a *distribution*, however, as the scalar structure is still owned by the whole group.

Although the common ad hoc distribution strategy is used, it is still necessary to specify a default distribution which will be used when creating scalar variables from constants, e.g.:

```
int x;
x = 5;
```

The default distribution will place the element on the 'root' processor of the group, as illustrated in Figure 6.1(a). Other distributions can arise such as the one illustrated in Figure 6.1(b). How this happens will be explained in Section 6.2.4.

6.2.2 The HSM Array Type

A HSM array is a multi-dimensional array of HSM data structures, which can themselves be arrays or compound structures. As with the scalar types, the common ad hoc distribution strategy allows any distribution of the elements, but it is still necessary to specify a default distribution which will be used when creating arrays, e.g.:

```
array< int > a;
a = array_create( 1, 8 );
```

The distribution strategy that will be described appears rigid, and could in fact be used for a rigid distribution strategy, but is simply a default ad hoc distribution.

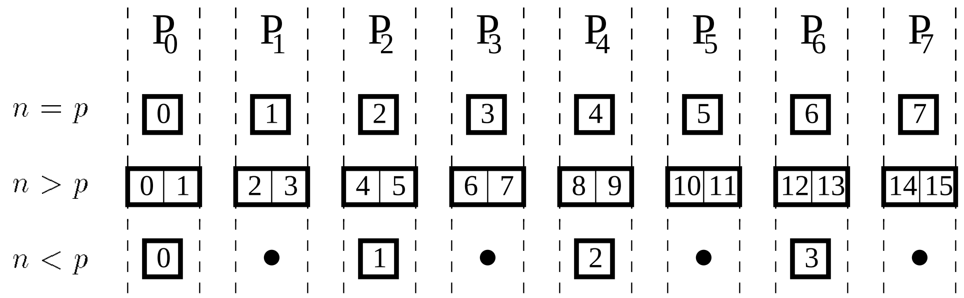

Figure 6.2: Simple examples of each of distribution cases.

The strategy is designed so that the array can be partitioned easily into sub-arrays, with one sub-array for each child node. Other distributions may arise after assigning arrays or sub-arrays, as will be discussed in Section 6.2.4.

A multi-dimensional array of elements is not the same as a one-dimensional array of arrays of elements, but §3.3.7 showed that it is possible to *transform* the one into the other. For this reason, the distribution of a multi-dimensional array will be made the same as a one-dimensional array of arrays with one fewer dimension. This gives a recursive distribution strategy where only the distribution of the one-dimensional array needs to be defined. Using a block distribution strategy of a one-dimensional array with n elements over a group of p processors, three cases can be identified:

- $n = p$ — Each processor will hold exactly one element.
- $n > p$ — Each processor will hold one or more elements.
- $n < p$ — Each element will be distributed over one or more processors.

Figure 6.2 illustrates one example for each of the cases on a node of eight processors:

```
array< int > a, b, c;
a = array_create( 1, 8 );     // n = p
b = array_create( 1, 16 );    // n > p
c = array_create( 1, 4 );     // n < p
```

For simplicity, the following discussion will assume that each of the elements are simple scalar structures. It will be shown later that the strategy extends easily to non-scalar structures.

In the balanced distribution of n scalar elements over p processors, processor P_i

will hold n_i elements, where:

$$\left\lfloor \frac{n}{p} \right\rfloor \leq n_i \leq \left\lceil \frac{n}{p} \right\rceil$$

This gives r processors holding $\left\lceil \frac{n}{p} \right\rceil$ elements and $p - r$ processors holding $\left\lfloor \frac{n}{p} \right\rfloor$ elements, where r is the remainder:

$$r = (n \bmod p) \quad \text{and} \quad 0 \leq r < p$$

If $r = 0$ the distribution is perfectly balanced and each processor will hold the same number of elements as every other processor:

$$n_i = \frac{n}{p}$$

If $r > 0$ the distribution will not be perfectly balanced. The selection of which r processors should be allocated the extra element is important as it can seriously affect the balance of nested compound data structures. To optimise the balance, the group of processors is partitioned implicitly into a number of sub-groups of approximately equal size, and the distribution of the elements is then handled separately for each sub-group. The size g_i of any sub-group will differ from any other sub-group by at most one processor:

$$g_i \leq g_j + 1$$

which assures a more balanced distribution of the elements. The allocation of storage for the elements within each sub-group is handled recursively with the base case reached when $r = 0$ for each sub-group.

The first non-base case to handle is where n and p (and hence r) have a common factor:

$$f = \gcd(n, p), \qquad f > 1$$

The group of processors can then be partitioned evenly into f sub-groups. The number of data elements and processors in each sub-group is therefore guaranteed to have been reduced:

$$g_i = \frac{p}{f}, \qquad n_{g_i} = \frac{n}{f}$$

The second case, when n and p have no common factor, i.e. are co-prime, forces an unbalanced distribution so the number and size of the sub-groups is chosen to minimise the unbalance. If $r > 1$, the processors are partitioned into r sub-groups

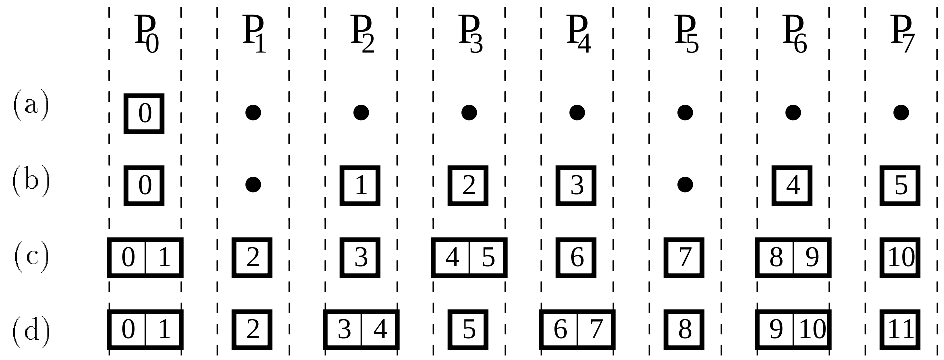

Figure 6.3: Example distributions of one-dimensional arrays over a group of eight processors with (a) one, (b) six, (c) eleven and (d) twelve elements.

of approximately equal size, with (p mod r) 'extra' processors, i.e. (p mod r) groups with one more processor than the other groups. The allocation of these extra processors can be decided by giving the first (p mod r) groups an extra processor:

$$g_i = \begin{cases} \left\lceil \frac{p}{r} \right\rceil & \text{if } i < (p \bmod r) \\ \left\lfloor \frac{p}{r} \right\rfloor & \text{otherwise} \end{cases} \qquad \sum_{i=0}^{i<r} g_i = p$$

Each group g_i is then allocated n_{g_i} elements:

$$n_{g_i} = g_i \left\lfloor \frac{n}{p} \right\rfloor + 1$$

$$\sum_{i=0}^{i<r} n_{g_i} \quad = \quad \left(\sum_{i=0}^{i<r} g_i \right) \left\lfloor \frac{n}{p} \right\rfloor + r \quad = \quad p \left\lfloor \frac{n}{p} \right\rfloor + (n \bmod p) \quad = \quad n$$

If $r = 1$, the above subdivisions do not reduce the problem so a special case is needed. Since there is exactly one 'remainder' element, it is allocated to the root processor with the rest of the elements distributed evenly. Each subgroup therefore has exactly one processor.

$$g_i = 1, \qquad \sum_{i=0}^{i<r} g_i = p$$

$$n_{g_i} = \begin{cases} \left\lceil \frac{n}{p} \right\rceil & \text{if } i = 0 \\ \left\lfloor \frac{n}{p} \right\rfloor & \text{otherwise} \end{cases}$$

$$\sum_{i=0}^{i<r} n_{g_i} \quad = \quad \left\lceil \frac{n}{p} \right\rceil + (p-1) \left\lfloor \frac{n}{p} \right\rfloor \quad = p \left\lfloor \frac{n}{p} \right\rfloor + 1 \quad = \quad n$$

Figure 6.3 shows the distributions of several one-dimensional arrays of different lengths over a node of eight processors.

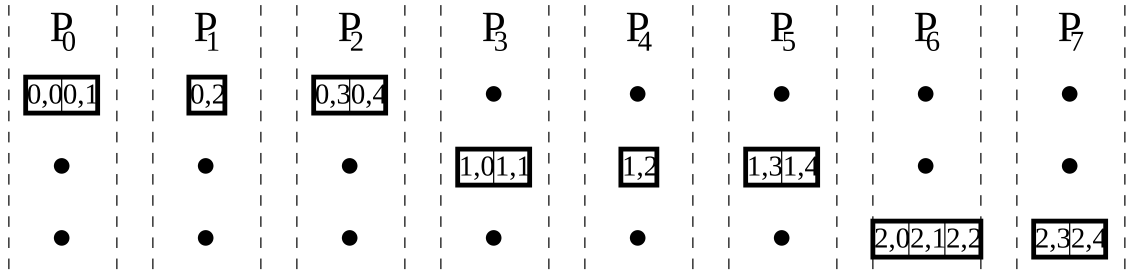

Figure 6.4: Example distribution of a two-dimensional array with three rows and five columns over a group of eight processors.

Non-scalar Array Elements

The above strategy explains how scalar array elements are distributed over the available processors. If the elements are non-scalar the distribution is handled almost identically, with only one case needing clarified. When a single element is to be distributed over several processors, i.e. $n = 1$, $p > 1$, $r = 1$, it will not be located entirely on the root processor of the group. Instead, the non-scalar element is itself distributed over all the p processors in the subgroup. Figure 6.4 shows the distribution of a two-dimensional array over a group of eight processors, which will be the same distribution as a one-dimensional array of one-dimensional arrays of scalars:

```
array< int > a;
a = array_create( 2, 3, 5 );
```

6.2.3 The HSM Compound Structure Type

The HSM compound structure is similar to the scalar structure in that it cannot be partitioned, and similar to the **array<>** structure in that each of its elements are further HSM structures. The member elements are therefore each distributed over the whole node. Figure 6.5 shows the distribution of a compound structure containing two scalars and a two-dimensional array of scalars:

```
struct { int x, y; array< int > a; } s;
s.a = array_create( 2, 2, 8 );
```

6.2.4 Data Structure Distribution after Assignment

The previous sections have described a default distribution strategy for creating and initialising the HSM data structures. Performing data structure arithmetic

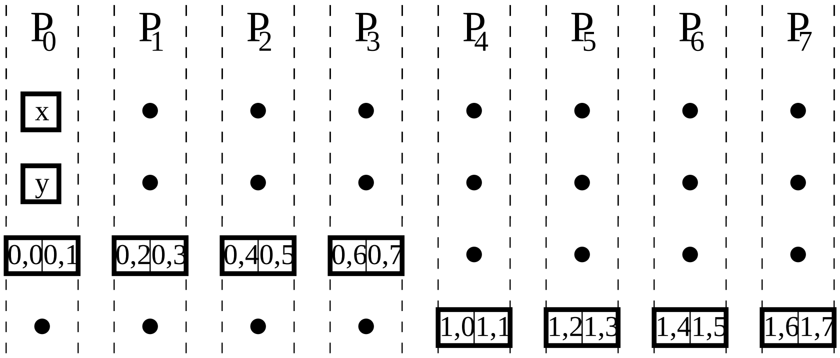

Figure 6.5: Example distribution of a compound structure containing two scalars and a two-dimensional array with two rows and eight columns over a group of eight processors.

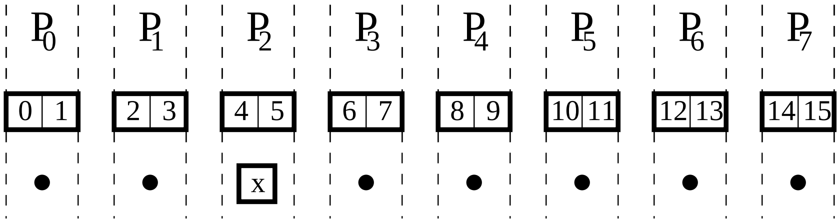

Figure 6.6: Example distribution of a scalar variable after assignment from the sixth element of a one-dimensional array.

on structures which do not have the same distribution will involve an implicit redistribution of the values to the required processors. Redistributions can be expensive in terms of the communications involved, so they are minimised by trying to keep the element distributions the same.

If a structure has not been created yet and it is being assigned to, the distribution of the elements of the structure will be the same as that of the operand, i.e. no element values will have to be communicated between processors. Figure 6.6 shows an example of assigning a value of an array to a scalar variable:

```
int x;
array< int > a;
a = array_create( 1, 16 );
x = a[5];
```

If the structure has already been created, the distribution is set and the value will have to be redistributed. Figure 6.7 shows an example of assigning a sub-array to a previously created array:

```
array< int > a, b;
a = array_create( 1, 16 );
b = a[0..7];
```

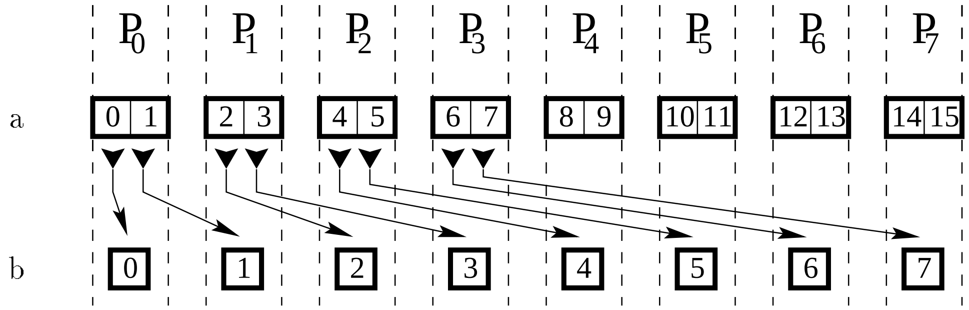

Figure 6.7: Example redistribution of a sub-array after assignment to an already created array: `b=a[0..7];`

6.3 The HSM/C++/MPI Interface

The HSM/C++/MPI interface gives the C++/MPI programmer access to the HSM data structures created within the skeleton superstructure. Each processor will have access to all the data structure elements that are stored on the local processor, but not to those stored on the other processors. No synchronisation of the processors in the leaf node is required, but the SFMD model required of the base node will guarantee that all the processors will have started executing the same base level function and will synchronise when all have finished. Each processor can manipulate the local data without synchronising with the other processors. A new HSM data structure can be created within the base node, but only if all the processors took part in the creation concurrently.

Each local instance of the HSM data types is defined as a C++ class, which embodies the structure access functions. The scalar, **array<>** and **struct** types will be described in §6.3.1, §6.3.2 and §6.3.4 respectively.

6.3.1 The HSM Scalar Types

As already explained, a HSM scalar data structure will store its value in the memory of only one processor in the group. Every processor will have an instance of the structure, but only one processor will store and have access to the value.

A HSM scalar variable is declared using the appropriate class, for example HSM_int, as shown in Figure 6.8. The classes for HSM_bool, HSM_real and HSM_string are similar.

Scalar structures passed as arguments to the base functions will have already been created by the skeleton hierarchy before the processors start executing the

```
class HSM_int {
  public:
    create( int distribution );  // create the structure
    operator int ();             // the actual value, if stored locally
    int is_local();              // true or false
    int where_is();              // the rank of the owning processor
    int add_local();             // add the element locally
    MPI_Type mpi_type();         // the MPI type of the element
}
```

Figure 6.8: C++ class for the HSM scalar integer type.

base function. The processors can use the is_local() member function to see if the element is stored locally, and the where_is() function to find out the rank of the processor on which the element is stored. If the element is stored locally the value can be accessed by simply accessing the structure:

```
int x;
HSM_int y;
if (y.is_local()) x=y;
```

This is accomplished by using the integer cast operator of the HSM_int class.

The owning processor can communicate the value of the scalar variable using any of the MPI message passing routines. The MPI type of the variable can be found using the mpi_type() member function:

```
MPI_Bcast( &x, 1, x.mpi_type(), x.where_is(), HSM_COMM_WORLD );
```

If the scalar structure has not been created yet it can be created by each processor calling the create() member function:

```
HSM_int x;
x.create( HSM_DISTRIBUTION_STANDARD );
```

The create() function takes a single argument which specifies the type of distribution to use. This is a choice between HSM_DISTRIBUTION_STANDARD and HSM_DISTRIBUTION_ADHOC.

The standard distribution will place the storage space for the data element on the root processor as described in §6.2.1.

The ad hoc distribution, on the other hand, will not allocate any storage for the element. It will be up to exactly one processor to call the add_local() member

```
template <class Type>
class HSM_array {
  public:
    create( int dimms, ..., int distribution );  // create the structure
    Type operator()( int i, ... );               // the value of an element
    Type element( HSM_index ind );               // the value of an element
    int ndims();                                 // number of dimensions
    int size( int i );                           // the size of a dimension
    HSM_index first_local();
    HSM_index next_local();
    HSM_index last_local();
    MPI_Type mpi_type();                         // the MPI type of the element
}
```

Figure 6.9: C++ class for the HSM array type.

function. Since none of the member functions entail any implicit communication, none of the other processors will know where the element is located. The validity of the scalar structure will be checked on exit from the base level functions before the processors join the HSM hierarchy again.

6.3.2 The HSM Array Type

The HSM **array**<> type is also represented by an instance of a C++ class on every processor. The class, shown in Figure 6.9, is a template class where the member elements are further HSM data structure types.

HSM arrays passed as arguments to the base functions will have already been created by the skeleton hierarchy before the base function code is reached. The processors can use the is_local() member functions of each of the array elements to see if the element is stored locally, as well as the where_is() member functions.

```
HSM_array< HSM_int > z;
if ( z(5,4).is_local() ) {
    z(5,4) = 0;
}
```

As with the scalar structures, the processor that owns an element can communicate the value of the element using any of the MPI message passing routines:

```
int rank;
MPI_Comm_rank( HSM_COMM_WORLD, &rank );
assert( z(rank,0).is_local() );
MPI_Bcast( &z(rank,0), 1, z.mpi_type(), z(0,0).where_is(), HSM_COMM_WORLD );
```

In this example the value of element $z_{<0,0>}$ is broadcast to each of the other processors and assigned to the locally stored element $z_{<i,0>}$.

By using the mpi_type() member function it is also possible to send several array elements in one go:

```
MPI_Bcast( &z(rank,0), z.size(1), z.mpi_type(),
           z(0,0).where_is(), HSM_COMM_WORLD );
```

This will broadcast the first row of the matrix from processor one to all the other processors. If the HSM array has not been created yet it can be created by each processor calling the create() member function concurrently. The first argument the function takes specifies the number of dimensions of the array, followed by one argument specifying the size of each dimension. The last argument specifies the type of distribution required, with a choice of HSM_DISTRIBUTION_STANDARD and HSM_DISTRIBUTION_ADHOC as there was for the scalar distribution.

```
HSM_array< HSM_int > z;
z.create( 2, 10, 10, HSM_DISTRIBUTION_STANDARD );
```

Each processor will create all the member elements of the array, but only some of the elements will be stored locally, as was described in §6.2.2.

As for the scalar data structures, an ad hoc distribution for the array will create all the member elements, but not assign any storage space. Storage will have to be assigned by each processor adding the local elements using the add_local() member function of the element. None of the processors will know where non-local elements are located, and the validity of the structure will again be checked on exit from the base level functions.

6.3.3 The HSM Index Type

As was shown above, the elements of an array can be referenced using simple round bracket notation, e.g. z(5,4). This requires one argument for each dimension, which doesn't allow for a single scalar to reference any element. For this reason the HSM_index type is provided, as shown in Figure 6.10.

```
class HSM_index {
  public:
    create( int dimms, ... );         // create the structure
    int ndims();                      // number of dimensions
    int operator()( int i, ... );     // the value of an element
    operator int();                   // cast to an integer value
}
```

Figure 6.10: C++ class for the HSM index type.

The HSM_index type is effectively a one-dimensional array of integers, where each integer represents the index in the corresponding dimension of the array. The HSM_index structure can be used to access elements of the array using the additional element() member function in the HSM_array<> class.

The HSM_array<> class also provides member functions for iterating through all the locally stored elements of the array:

```
HSM_index ind;
HSM_array< HSM_int > z;
for ( ind=z.first_local(); ind<=z.last_local(); ind=z.next_local() ) {
    z.element(ind) = 0;
}
```

The index can be cast to an integer value which represents the *local* index of the element, i.e. an index into the one-dimensional array of elements stored locally. This allows inequality operations to be defined to test the ordering of the elements locally, such that an index is less than another index if the former element is stored before the latter. The integer cast also allows indices to be subtracted from one another to give the relative displacements of locally stored elements, i.e. one plus the number of elements stored locally between the two. An integer value can also be added to an index to generate the index of a later element with the given displacement from the former, so that the next_local() member function becomes a simple integer increment operation.

The arithmetic on the index structures is necessary to calculate the number of elements stored locally so that some of the values can be sent to other processors in the group. The merge-sort problem discussed in Chapter 9 gives a good example of the need for HSM_index arithmetic.

```
class HSM_intpair {
  public:
    HSM_int x;
    HSM_int y;
    create( int distribution );  // create the structure
    MPI_Type mpi_type();         // the MPI type of the element
}
```

Figure 6.11: C++ class for a HSM compound structure of two integers.

6.3.4 The HSM Compound Structure Types

The HSM compound structures can only be used in the base level if there was an appropriate **typedef** command in the HSM program. This will create a new class in the base code with the HSM member structures as expected, and create() and mpi_type() member functions. Figure 6.11 shows the base level class for a pair of integers defined using the following type definition in the HSM program:

```
typedef struct { int x, y; } intpair;
```

As before, the distribution type needs to be specified when the structure is created, and the distribution type is inherited by each of the HSM member structures.

6.4 A HSM/C++/MPI Example

The following example, identical to the one used in the last chapter for the shared memory target architecture, maps a base function over an array of arrays:

```
basefn int fork_body_fn( void global, array< int > local );
void main( void ) {
    array< array< int >> x;
    array< int > y;
    x = array_create_init( 1, 10, vector_create_indexed( 10 ) );
    y = map( fork_body_fn, NULL, x );
}
```

The base function, expressed in C++/MPI, uses the built-in MPI_Reduce() function to sum one value from each processor:

```
HSM_int fork_body_fn(
    HSM_void global,
    HSM_array< HSM_int > local
) {
    int local_sum=0, full_sum;
    HSM_int sum;
    HSM_index i;
    for ( i=local.first_local(); i<=local.last_local(); i=local.next_local() ) {
        local_sum += local.element(i);
    }
    sum.create( HSM_DISTRIBUTION_STANDARD );
    MPI_Reduce( &local_sum, &full_sum, 1, MPI_INT, MPI_SUM,
                sum.where_is(), HSM_COMM_WORLD );
    if ( sum.is_local() ) sum=full_sum;
    return( sum );
}
```

A separate ' full_sum' variable is needed as the MPI_Reduce() function must use the same types for the send and receive buffers.

Part II

Evaluation

Chapter 7

Evaluation of the HSM Model

Contents

7.1	The Problems	78
7.2	The Languages	79
7.3	The Software Metrics	80
7.4	The Language Comparison	83

The previous chapters have described the HSM model and given simple examples of top level code and base level code, using synchronous shared memory and distributed memory target architectures. The next three chapters apply the HSM model to a selection of three problems to evaluate the programmability of the HSM model. The purpose of the evaluation is to see if the model can be used to express established parallel algorithms and compare the conceptual programming effort involved to that required when programming without the HSM top level code, i.e. using purely the base language.

7.1 The Problems

The three problems that will be looked at are all potentially parallelisable, and demonstrate non-trivial instances of the algorithmic skeletons described in Chapter 4. The problems manipulate a range of data types, including one- and two-dimensional arrays and compound structures as described in Chapter 3.

7.1.1 The Image Thresholding Problem

Problem Description: *Given a grey level image I and a target percentage p, construct a binary image B such that $B_{i,j}$ is set if no more* *than p percent of the pixels in I are lighter than $I_{i,j}$.*

The image thresholding problem, tackled in Chapter 8, manipulates two-dimensional arrays using the reduce and map skeletons. The instance of the reduce skeleton is interesting in that the reduce operation can itself be parallelised, which is done to illustrate the use of ad-hoc parallelism. The implementation can also use the scan and filter skeletons which require the manipulation of arrays of structures instead of just arrays of integers.

7.1.2 The Merge-Sort Problem

Problem Description: *Given an array of n integers, sort the integers into increasing order by dividing the array into two sub-arrays of size $\frac{n}{2}$; sorting both sub-arrays concurrently; and merging the two sorted sub-arrays into a sorted array of n integers.*

The merge-sort problem, tackled in Chapter 9, uses a *divide-and-conquer* paradigm which can be expressed using the divide and reduce skeletons with the possible use of map and zip skeletons as well. The instance of the divide skeleton turns out to be trivial, but as in the image thresholding problem the reduce skeleton can exploit parallelism within the reduce operation. The solution is extended to exploit a sequential library routine for the distributed memory target architecture.

7.1.3 The All-Pairs Shortest Paths Problem

Problem Description: *Given a directed graph of n vertices with weighted edges, calculate the weights of the shortest paths between all pairs of vertices.*

The all-pairs shortest paths problem, tackled in Chapter 10, again manipulates two-dimensional arrays, and uses map, reduce, zip, if and while skeletons. The algorithm has explicit nesting of parallelism with two distinct levels. Parallelism can be exploited in either one of the levels or in both levels.

7.2 The Languages

To evaluate the programmability of the HSM model, solutions to each of the problems are expressed using the HSM language for both shared and distributed memory target architectures. This required writing a *skeleton superstructure* for the problem in the HSM language, and then coding separate instances of the functions at the leaf nodes of the hierarchy in the languages of the base models. It is also possible to code the leaf node functions purely in the HSM language, which does not exploit the potential for ad hoc parallelism in the leaf node. This gives a total of three HSM based solutions:

(a) **Pure HSM** — a skeleton superstructure is expressed using the HSM language and the code for the leaf nodes is also expressed using the HSM language.

(b) **HSM/Fork95++** — a skeleton superstructure is expressed using the HSM language and the code for the leaf nodes is expressed using the Fork95++ language with the HSM/Fork95++ interface on a shared memory machine .

(c) **HSM/C++/MPI** — a skeleton superstructure is expressed using the HSM language and the code for the leaf nodes is expressed using the C++ language with the MPI message passing library and HSM/C++/MPI interface on a distributed memory machine.

Solutions to the problems were also expressed directly using the language of the target architectures, namely Fork95++ for the shared memory target and C++ with the MPI message passing library for the distributed memory target architecture. This gives a further two solutions to each of the problems:

(d) **Pure Fork95++** — the solution is expressed using just the Fork95++ language on a shared memory machine.

(e) **Pure C++/MPI** — the solution is expressed using just the C++ language with the MPI message passing library on a distributed memory machine.

The five solutions and their relation to one another are illustrated in Figure 7.1.

7.3 The Software Metrics

The programmability and usability of the HSM model are assessed by looking at how effectively the HSM language was used to implement the solutions for each of the three problems described above. The solutions are compared using four separate software metrics.

Various research into software metrics has been undertaken in the context of software engineering and project management. Software metrics can be separated

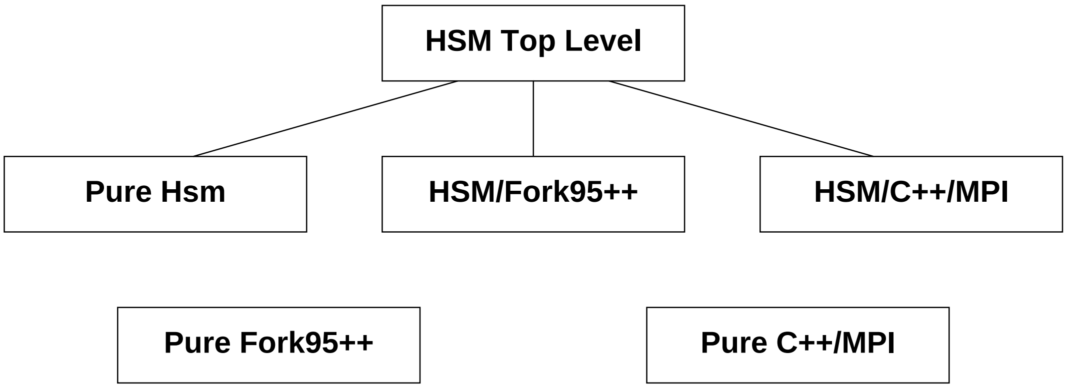

Figure 7.1: Language comparison summary.

into three categories [Pre92]:

(a) productivity metrics

(b) quality metrics

(c) technical metrics

Of the three categories only the technical metric category is suitable for the evaluation of the five language combinations, as it focuses on the 'character' of the software. The example problems are too small to be used to assess the productivity of the software engineering process or to assess the conformance to customer requirements, so all four metrics that are used are technical metrics.

A second categorisation of software metrics can also be made, into *size*-, *function*- and *human*-oriented metrics:

(a) size-oriented metrics — direct measures of software and the process by which it is developed

(b) function-oriented metrics — indirect measures of software and the process by which it is developed

(c) human-oriented metrics — measures of the manner in which people develop software and perceptions about the effectiveness of tools and methods.

Size-oriented metrics, such as the number of lines of code or the numbers of operators and operands, are easy to collect but don't necessarily mean that much on their own. It is particularly difficult to establish size-oriented metrics which may be applied to a range of languages and whose results are validly comparable. The first two software metrics used are objective size-oriented metrics which extract the salient features of the imperative languages and whose results are therefore comparable.

Function-oriented metrics, such as the *function point* method, give an empirical measure of software complexity by means of e.g. number of files, external interfaces, user inputs/outputs/inquiries etc.. Again, these metrics are more suitable for software projects rather than the simple problems used in this thesis.

Measuring the *perceptions* of the programmer is, however, reasonable as the problems exhibit a wide range of skeletons and data structures. The two remaining software metrics used are therefore human-oriented metrics.

A more detailed analysis of human perceptions would involve *field-tests* of the language and programming model on a range of programmers, but this is beyond the scope of the thesis. The two human-oriented metrics therefore involve only the subjective perceptions of one programmer.

7.3.1 Solution Size

The two size-oriented metrics measure the total length of the solution. The first metric, **lines** is a simple count of the number of lines of code for the solution. This does not include comments or blank lines.

The second metric measures the total number of **operations** in the solutions. More precisely, an operation is an assignment; a function call; a keyword; a binary operator or an array index operation. Unary operators and the simple addition/subtraction binary operators are not included. This metric turns out to be applicable to all three languages with only a minor modification for the Fork95++ language to consider the group and rank variables as operators.

The operations were chosen to give an indication of the *effort* that went into the solution by the programmer, so that the **operations** metric links program size and programmer effort more accurately than the **lines** metric.

7.3.2 Solution Complexity

One of the main goals of developing the HSM programming model was to make parallel programming less complicated than existing programming models. In particular, this requires that solutions are clear and concise.

The third metric, **clarity**, tries to measure the perception of clarity, i.e. whether or not it is clear what is going on in the code. Since the measurement is subjective, a three-state variable is used. The solution code is either clear, unremarkable, or unclear.

The fourth metric, **conciseness**, tries to measure the perception of succinctness, i.e. whether or not the solution is expressed briefly and to the point. This is normally measured relative to the algorithm complexity so that a solution which follows the algorithm closely and in similar length will be concise. As with the **clarity** metric, a three-state variable is used. The solution code is either concise, unremarkable, or long.

7.4 The Language Comparison

Solutions based on the HSM model were possible for all three problems, as were solutions programmed without the HSM model. For fairness, the time spent coding the solutions in each language were kept as similar as possible at roughtly one day per solution (not including the initial analysis of the problem). Given this restraint on keeping the time commitments equal, it was not always possible to implement the same degree of flexibility over all five language combinations. In particular, some of the solutions are restricted to a certain problem size or don't exploit all the parallelism available. This makes it difficult to make objective comparisons between the solutions, but the perceptual restrictions are real and do affect the usability of the language.

The following three chapters will now evaluate the model using the metrics on all five language combinations for each problem in turn. The results of applying the four software metrics are given for each of the solutions, and any restrictions of the implementations are also discussed.

Chapter 8

The Image Thresholding Problem

Contents

8.1	The HSM Skeleton Superstructure	85
8.2	The Pure HSM Solution	86
8.3	The HSM/Fork95++ Solution	88
8.4	The HSM/C++/MPI Solution	91
8.5	The Pure Fork95++ Solution	95
8.6	The Pure C++/MPI Solution	97
8.7	Summary	98

Problem Description: *Given a grey level image I and a target percentage p, construct a binary image B such that $B_{i,j}$ is set if no more than p percent of the pixels in I are lighter than $I_{i,j}$.*

The problem is one of the suite of Cowichan Problems suggested by Wilson *et al.* to assess and compare the usability of parallel programming systems:

> *The general idea is that an image's histogram should have 2 peaks, one centred around the average foreground intensity, and one centered around the average background intensity. This program attempts to set a threshold between the two peaks in the histogram and select the pixels above the threshold.* ([WIS94])

Ideally the percentage p should be calculated to lie at the trough between the two peaks but for simplicity the problem assumes that p is fixed.

The problem has two obvious potentials for parallelisation. Firstly in calculating the histogram and secondly in applying the threshold. There is also some possibility in parallelising the calculation of the threshold value. Following this breakdown, the solution takes the form of a sequential composition of three stages:

(a) **calculate histogram** — construct the image's histogram by summing together the 'histograms' of all the pixels.

(b) **calculate threshold** — scan through the histogram summing the pixel counts until p percent of the pixels are accounted for, setting the threshold t at the intensity found.

(c) **apply threshold** — map the threshold value over the pixels of image I, generating the corresponding pixel in B.

The image I can be represented as a two-dimension array of integers, and the image B as a two-dimension array of booleans. For simplicity image B will also be an array of integers but this need not be the case. A histogram is a one-dimension array of integers with the array index representing the pixel intensity.

Section 8.1 describes a top level skeleton formulation of the problem which is used in the first three solutions. Solutions are discussed using pure HSM in Section 8.2; HSM with a Fork95++ base level in Section 8.3; and HSM with a C++/MPI base level in Section 8.4. For comparison, solutions are coded without the HSM model using pure Fork95++ in Section 8.5; and pure C++/MPI in Section 8.6. The results are summarised and briefly discussed in Section 8.7. The code for all the solutions is given in Appendix B.

8.1 The HSM Skeleton Superstructure

Each of the three stages suggested above decomposes neatly into one or more skeleton instances. Stage (a) "**calculate histogram**" involves a reduction operation on an array of histograms to a single histogram, so before that can happen the image (2D-array of integers) must be converted to a 2D-array of histograms. This requires a map skeleton to map each pixel to a histogram.

The use of the reduce skeleton is an interesting part of the solution as it introduces the possibility of not only performing several reduction operations concurrently, but also of parallelising the reduction operation itself. The operation involves summing two histograms which may be done using a zip skeleton, or can be performed by a separate program written in the base language. This latter approach illustrates the potential for exploiting ad-hoc parallelism within the base model.

Stage (b) "**calculate threshold**" involves a prefix-sum operation on the histogram (1D-array of integers) creating another 1D-array of integers representing the cumulative counts at each intensity level. Selecting the appropriate threshold intensity involves searching for the first intensity level which has an accumulated count greater than the required number of pixels.

This can be done using a SIMD style base language function which compares every element with its left neighbour and returns the intensity level of the element that straddles the required threshold pixel count. The threshold level calculation can also be expressed purely in HSM code as will be described in Section 8.2.

Lastly, stage (c) "**apply threshold**" uses a map skeleton to apply the threshold intensity to each pixel of image I to create the image B.

Figure 8.1 shows the HSM code for the skeleton superstructure, with the full code listing given in Appendix B.1. The code is clear, but is slightly longer than expected due to the need to define the various simple attendant functions such as converting a pixel to a histogram.

Language	HSM
Lines	73
Operators	40
Clarity	clear
Conciseness	-
Comments	Simple attendant functions bloat the solution.

8.2 The Pure HSM Solution

The histsum_fn() function can be expressed using pure HSM code, as shown in Figure 8.2, but since the HSM language provides no facilities for performing file input/output a separate base level is still needed to code the load_image_fn() and save_image_fn() functions.

When the histsum_fn() is called from the top level **main()** function it is passed a **NULL** argument where the global parameter is expected. This is not ideal but the skeleton syntax must allow for a global parameter to be passed which is the same for all function evaluations. The histsum_fn() code ignores the argument by declaring it to be of type **void**.

The threshold intensity calculation of stage (b) "**calculate threshold**" can also be expressed using pure HSM code. The cumulative histogram can be filtered to remove all the intensity entries below the threshold count, so that the first pixel entry left, i.e. index zero of the array, will then be of the appropriate intensity value to set the threshold. This requires an extended histogram data structure as

```
void main(void)
{
    int threshold_count, threshold_intensity;
    array<int> image, binary_image;
    histogram_t full_histogram, cumulative_histogram;
    array<histogram_t> histogram_array;

    image = load_image_fn( "image.data" );
    threshold_count = (int)(0.7*(real)(array_size(image)[0]*array_size(image)[1]));

    histogram_array = map( inttohist_fn, NULL, image );
    full_histogram = reduce( histsum_fn, NULL, histogram_array );
    cumulative_histogram = scan( histelementsum_fn, NULL, full_histogram );
    threshold_intensity = find_threshold_fn( threshold_count, cumulative_histogram);
    binary_image = map( threshold_fn, threshold_intensity, image);

    save_image_fn( "thresholded_image.data", binary_image );
}
```

Figure 8.1: HSM code for the image thresholding problem.

```
    histogram_t histsum_fn(
        void global, histogram_t histogram_a, histogram_t histogram_b
    ) {
        return( zip( histelementsum_fn, NULL, histogram_a, histogram_b ) );
    }

    histogram_element_t histelementsum_fn(
        void global, histogram_element_t element_a, histogram_element_t element_b
    ) {
        histogram_element_t element_c;
        element_c.i = element_a.i;
        element_c.c = element_a.c + element_b.c;
        return( element_c );
    }
```

Figure 8.2: HSM code for summing two histograms.

```
int find_threshold_fn(
    int threshold_count, histogram_t cumulative_histogram
) {
    threshold_histogram = filter( histelementcmp_fn, threshold_count,
                                  cumulative_histogram);
    return( threshold_histogram[0].i );
}
```

Figure 8.3: HSM code for finding the threshold value.

the elements of the filtered array do not actually contain any information about their respective indices in the original array. It is therefore necessary to store the intensity level information in the histogram elements themselves. This can be done by replacing the integer elements with structures containing an integer for the pixel intensity and an integer for the pixel count, i.e.:

```
typedef struct { int i; int c; } histogram_element_t;
typedef array<histogram_element_t> histogram_t;
```

This is not a problem, but it does mean that the prefix-sum addition operation now adds two structures rather than just two integers. Figure 8.3 shows the top level function for finding the threshold value.

The code is concise and clearly shows where each skeleton is being used, but the use of the filter skeleton in the find_threshold_fn() algorithm does not reflect the natural expression of the search.

Language	pure HSM
Lines	90 (73+17)
Operators	46 (40+6)
Clarity	clear
Conciseness	concise
Comments	Filter skeleton does not reflect algorithm.

8.3 The HSM/Fork95++ Solution

The HSM skeleton superstructure described in Section 8.1 leaves four functions to be written in the base language: load_image_fn(), save_image_fn(), find_threshold_fn and histsum_fn().

Figure 8.4 gives an extract of the base language function load_image_fn() written using the Fork95++ base model with the HSM/Fork95++ interface. The full code

```
HSM_array<int>& image = *(new HSM_array<int>);
farm if ($==0) {  // File I/O is only sequential
    pr int rows, columns, i, j;
    ...
    image.create(2, rows, columns);
    for (i=0; i<rows; i++) {
        for (j=0; j<columns; j++) {
            fscanf(filehandle, "%d", &image(i,j) );
        }
    }
}
```

Figure 8.4: HSM/Fork95++ code for loading an image.

is given in Appendix B.3 along with the rest of the Fork95++ base level code. The code is concise, though the syntax differs from the pure Fork95++ language by indexing the histogram array using a round bracket notation array(i,j) as explained in Section 5.3. It is unfortunate that the HSM_array<> class cannot be written to use the standard square bracket array notation of C, e.g. array[i][j], but the round bracket notation still gives reasonable clarity.

As mentioned previously the histsum_fn(), shown in Figure 8.5 takes three HSM data structures as arguments, the first being the **NULL** global argument. The function is declared as a synchronous function using the **sync** qualifier and the arguments are implicitly shared as they are HSM structures. This is necessary as it ensures that all the processors in the leaf node of the hierarchy are synchronised and are passed the same arguments.

The code queries the histogram size and creates a new histogram using the array.size() and array.create() member functions respectively. One interesting detail is that the array.size() function is asynchronous and returns a private int, creating a write-conflict when assigned to the shared int n. n must be a shared int as the array.create() function takes only shared arguments to ensure that all the processors in the group are creating the same array. The write-conflict is not a problem, however, as all the processors will be trying to write the same value. For clarity, the private return values of array.size() are explicitly cast to a single shared integer in the program code, but this is not actually necessary.

To perform the local summation the program switches into asynchronous mode using the **farm** statement. Although not necessary, the array() access function

```
sync HSM_histogram_t histsum_fn(
    HSM_void& global,
    HSM_histogram_t& histogram_a,
    HSM_histogram_t& histogram_b
) {
    pr int i;
    sh int n;
    HSM_histogram_t& histogram_c=*(shnew HSM_histogram);

    n=(sh int)histogram_a.size(0);     // async function
    histogram_c.create(1, n);          // sync function
    farm for (i=$; i<n; i+=groupsize()) {
        histogram_c(i).i = histogram_a(i).i;
        histogram_c(i).c = histogram_a(i).c+histogram_b(i).c;
    }
    return histogram_c;
}
```

Figure 8.5: HSM/Fork95++ code for performing the local summation of the histogram.

is asynchronous and this makes it clear that each array access is independent of the others. The more usual style is to ignore the asynchrony and to keep the processors synchronised.

Section 8.2 explained how a pure HSM solution required each histogram to be a structure of two integers instead of simply one integer. This extended data structure is not necessary when using the Fork95++ base language, but the data structure has been adopted as firstly, it means only one instance of the skeleton superstructure code needs to be given, and secondly, it illustrates how the members of the histogram structure can be accessed using standard C notation, e.g. array(i).c.

The summation is performed by each processor iterating through selected elements of the array starting with the index set to the processor rank and proceeding through the array with a stride equal to the number of processors in the leaf node size. This is the cleanest way of evenly dividing up the calculations between the leaf node processors.

The threshold intensity can be found using the code for the find_threshold_fn() shown in Figure 8.6. The code relies on the fact that the cumulative histogram is a monotonically increasing function so there will only ever be one threshold in-

```
sync HSM_int& find_threshold_fn(
    HSM_int& threshold_count,
    HSM_array<int>& cumulative_histogram
) {
    pr int i;
    sh int n;
    HSM_int& threshold_intensity=*(shnew HSM_int);

    n=(sh int)histogram_a.size();
    threshold_intensity=0;
    for (i=$+1; i<n; i+=groupsize()) {
        if (cumulative_histogram(i−1).c<threshold_count
            && cumulative_histogram(i).c>=threshold_count) {
            threshold_intensity=i;
        }
    }
    return( threshold_intensity );
}
```

Figure 8.6: HSM/Fork95++ code for finding the threshold intensity.

tensity, but the Fork95++ implementation of the HSM_int data structure ensures that any write conflict would be resolved with either a priority or an arbitrary strategy.

The full code listing is concise, with each of the functions being expressed in a clear and natural manner.

Language	HSM/Fork95++
Lines	142 (73+69)
Operators	76 (40+36)
Clarity	clear
Conciseness	concise
Comments	Need to differentiate shared and private variables.

8.4 The HSM/C++/MPI Solution

The C++/MPI base code for load_image_fn() and save_image_fn() is very similar to that for the Fork95++ base level. Other than the absence of the **sh** and **pr** qualifiers, the only difference is that the programmer needs to specify the type of distribution when creating a new array. Figure 8.7 gives an extract of the base language code, with the full code listing given in Appendix B.4 along with the

```
HSM_array<int>& image = *(new HSM_array<int>);
image.create(2, rows, columns, HSM_DISTRIBUTION_STANDARD);
for (i=0; i<image.size(0); i++) {        // for each row
    for (j=0; j<image.size(1); j++) {    // for each column
        fscanf(filehandle, "%d", &image(i,j) );
    }
}
return image;
```

Figure 8.7: HSM/C++/MPI code for loading an image.

```
for (i=0; i<histogram_c.size(0); i++) {
     histogram_c(i).i = histogram_a(i).i;
     histogram_c(i).c = histogram_a(i).c + histogram_b(i).c;
}
```

Figure 8.8: HSM/C++/MPI code for sequentially summing two histograms.

rest of the C++/MPI base level code.

The histsum_fn() is more complicated than the input/output functions in that it exploits parallelism within the base model. The SPMD code first checks to see how many processors are taking part in the function execution by querying the MPI communicator HSM_COMM_WORLD. If there is only one processor the summation is done sequentially by the code shown in Figure 8.8.

If there is more than one processor the data has to be redistributed so that each processor has the data necessary to calculate the locally stored portion of the result histogram. Each processor can find out where to send the data by using the where_is() member function of the scalar element. This returns the rank of the processor where the element is stored.

Accessing one element of the array in the base language is straightforward, but the data redistribution requires accessing a block of several elements in one go. The first problem is how to calculate the number of elements that should be in the block. For a normal one-dimension array this may be done through arithmetic on the element indices as each index is simply an integer. For the multi-dimensional HSM_array<> the calculation is still done through simple arithmetic on the indices, as appropriate addition and subtraction operations have

```
HSM_index first;
...
first = histogram_a.first_local();
left_destination=histogram_c.element(first).where_is();
right_destination=histogram_c.element(first+blocksize).where_is();
MPI_Send( &histogram_a.element(first), block_size,
          histogram_a.mpi_type(), left_destination,
          0, HSM_COMM_WORLD);
MPI_Send( &histogram_a.element(first+block_size), block_size,
          histogram_a.mpi_type(), right_destination,
          0, HSM_COMM_WORLD);
...
a_source = histogram_a.element(first).where_is();
MPI_Recv( &histogram_newa.element(first), block_size,
          histogram_newa.mpi_type(), a_source,
          0, HSM_COMM_WORLD, &status);
...
```

Figure 8.9: HSM/C++/MPI code for sending and receiving parts of a histogram.

been defined for the HSM_index class. The difference between two indices was defined in Section 6.3.3 as one plus the number of locally stored elements between the two.

Once the block size has been calculated the next problem is how to pass the block as an argument to the MPI functions. Since the default distribution of arrays in the C++/MPI base level implementation guarantees that the local elements are stored consecutively and in order of increasing index, this can be done by returning the address of the first element in the block. To send the block of elements the MPI routines also need to know the type of the element which includes information as to how much memory each element takes up, which is found through the array.mpi_type() member function. The count argument to the MPI routines is then simply the number of elements to be sent.

Figure 8.9 shows an extract with the MPI send and receive instructions with Figure 8.10 illustrating the case with 4 processors and a histogram of length 8. The size of the machine is unfortunately restricted to being a power of two so that each processor only has to send two blocks. This could be overcome with a more sophisticated redistribution algorithm, but at significantly greater code complexity.

The new distributions of the two input arrays are still regular, in that the array

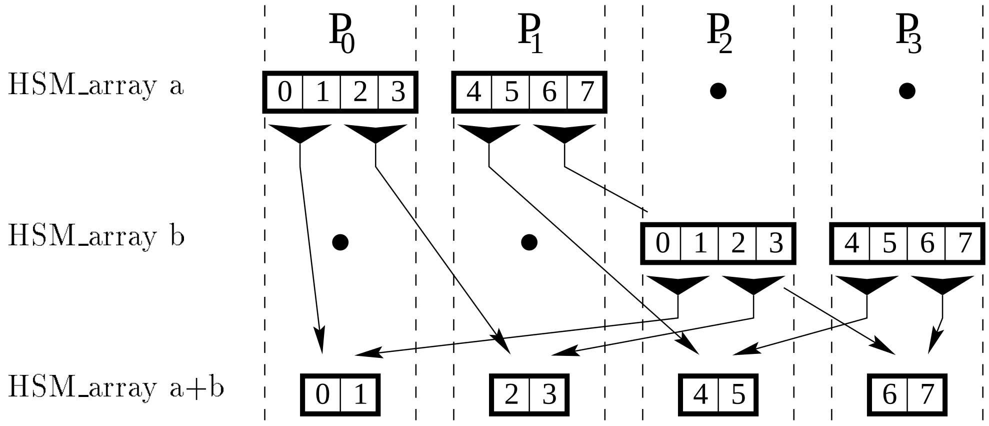

Figure 8.10: Sample distributions of histograms of size eight being summed over a group of size four.

```
HSM_index ind;
for (ind=histogram_c.first_local();
     ind<=histogram_c.last_local();
     ind=histogram_c.next_local()) {
    histogram_c.element(ind).i = histogram_newa.element(ind).i;
    histogram_c.element(ind).c = histogram_newa.element(ind).c
                                 + histogram_newb.element(ind).c;
}
```

Figure 8.11: HSM/C++/MPI code for performing the local summation of a histogram.

is split up unto contiguous blocks, so it would be reasonable to consider getting the HSM skeleton layer to perform the redistribution before calling the base function. This is not possible with the current language, but is discussed further in Chapter 11. Instead, it is left to the base level programmer to perform any communication or data passing necessary for the data redistribution.

Once the data has been redistributed the summation can be performed, as shown in Figure 8.11. The code uses the array.first_local(), array.next_local() and array.last_local() member functions to iterate through the locally stored elements. Since these functions return HSM_index structures, the array.element() function is used to access a particular element instead of the previously used array() member function.

Calculating the threshold intensity value is a relatively straight forward process in that each processor examines the locally stored elements of the cumulative his-

```
indices = new int[size];
MPI_Gather( local_index, 1, MPI_INT, indices, 1, MPI_INT,
            threshold_intensity.where_is(), MPI_COMM_WORLD );
if (threshold_intensity.is_local()) {
    for (i=0; i<size && indices[i]<0; i++) ;
    threshold_intensity = indices[i];
}
```

Figure 8.12: HSM/C++/MPI code for selecting the final threshold value.

togram and identifies a candidate threshold value. The HSM_int data structure only permits read or write access to the processor where the element is stored, so it is up to the programmer to select the correct value from those offered by the leaf group processors and send it to the owning processor. This is done by implementing a priority write resolution strategy using the MPI_Gather() operation. The 'root' processor of the node, where the default distribution places the value of the HSM_int, selects the first non-null element from the other processors, i.e. the processor with the lowest rank and valid threshold value succeeds. The code fragment is shown in Figure 8.12.

The base level code listing is quite long and not particularly clear, but the code is attempting a rather complicated data redistribution. Considering this, the solution is fairly straight forward. The code would be improved substantially by replacing the one and a half pages of data redistribution code with two calls to collective redistribution functions.

Language	HSM/C++/MPI
Lines	243 (73+170)
Operators	165 (40+125)
Clarity	-
Conciseness	long
Comments	Need redistribution functions in base model interface. Restricted machine size.

8.5 The Pure Fork95++ Solution

The structure of the HSM solution described in Section 8.1 can be applied even if skeletons are not being used explicitly. Appendix B.5 lists the code for a pure Fork95++ solution.

```
pr int i, row, col;

for (i=$; i<image.rows()*image.columns(); i+=groupsize()) {
    row=i/image.columns();
    col=i%image.columns();
    val=image(row,col);
    mpadd( &histogram[image(row,col)], 1 );
}
```

Figure 8.13: Pure Fork95++ code for calculating the image histogram.

The first point to note about the code is that it is now the Fork95++ programmer's responsibility to supply suitable classes for the various data structures involved. This leads to more work in designing and coding the classes, especially if the data structures are nested as they probably will be in more realistic examples. Allowing for the difference in classes, input and output is done in much the same way as for the Fork95++ base level code.

The biggest difference between the pure Fork95++ solution and the HSM/Fork95++ solution comes in the code for the main function where the functionality of each of the skeletons must now be implemented. Since the PRAM allows concurrent writes to the same memory location, only one histogram structure is needed. Figure 8.13 shows how the processors do not interfere when incrementing the intensity counts by using the built-in multi-prefix add instruction **mpadd()**. The code fragment also shows how the processors loop through the elements of the array as was done for the HSM/Fork95++ base level code.

The rest of the code is similar, and the **mpadd()** instruction is used once again to calculate the cumulative histogram. The full program listing is concise, but the pattern of parallelism embodied by each skeleton instance is obscured by the details of implementing the parallelism. The solution arrived at does exploit all the available parallelism, however.

Language	Pure Fork95++
Lines	115
Operators	102
Clarity	-
Conciseness	concise
Comments	High level concurrency is obscured.

8.6 The Pure C++/MPI Solution

MPI offers a set of collective communication routines which are put to good use in the pure C++/MPI code listed in Appendix B.6. In addition to the collective data distribution functions, MPI also offers a simple reduce skeleton which is used to calculate the full image histogram from the sum of each local histogram. Unlike the various HSM based solutions described above, each histogram summation is not parallelised. To do this would require code similar to that of the HSM C++/MPI base level code from Section 8.4.

The data distribution routines MPI_Scatter() and MPI_Gather() make the distribution of the image array almost trivial, but only with the restriction that the number of rows in the array must be divisible by the number of processors, i.e. the parallel data structure will have an even distribution. There are sister routines MPI_Scatterv() and MPI_Gatherv() which allow the scattering of different numbers of elements to each processor but it is the programmer's responsibility to calculate where each block starts and finishes and this requires considerable effort.

The image thresholding problem makes no use of the pixel positions so the image could be represented as one-dimension array, but it is reasonable to assume that some image manipulation algorithms require knowledge of the pixel coordinates. For this reason the array is sent and manipulated as a two-dimension array. Unlike the pure HSM solution where each processor can be given any number of elements by the map skeleton, the pure C++/MPI solution is restricted to giving each processor a block of complete rows. This is even more restrictive than the HSM/C++/MPI solution.

Even accepting these restrictions, the code is written purely looking at the local picture. The pixels of the image array do not map directly onto the pixels of the full_image array. Although the transformation can be calculated, this will lead to extra code complicating the solutions for problems requiring the 'global' pixel locations. For example, Figure 8.14 shows the code necessary for thresholding only every second row or column of the image.

The MPI_Bcast() function makes sending values to all the other processors a straightforward task, but has one minor drawback. Unless the programmer is willing to pack the required data into a single message, and unpack it at the other end, the programmer will end up using separate broadcast instructions for each parameter. If the parameters are all of the same type the packing is not too

```
for (i=0; i<image.rows(); i++) {
    for (j=0; j<image.columns(); j++) {
        image(i,j) = ((i+row_offset)%2 || (j+column_offset)%2)
                     && (image(i,j)>=threshold_intensity);
    }
}
```

Figure 8.14: Pure C++/MPI code for calculating a partial threshold operation.

complicated, but if they are of different types the programmer has the added task of deriving new MPI datatypes which is quite complicated.

The full program listing is reasonably concise but is disappointing in that it makes no attempt to exploit all the available parallelism. Not only is the calculation of the threshold value performed sequentially, the reduce operator is also only sequential. The C++/MPI model of programming makes it very difficult for the programmer to think of nesting the parallelism further.

Language	Pure C++/MPI
Lines	109
Operators	99
Clarity	-
Conciseness	concise
Comments	Restricted parallelism. Restricted problem size.

8.7 Summary

The image thresholding problem discussed in this chapter is a simple problem, but it has highlighted many of the programming issues surrounding the HSM model as well as touching on how the Fork95++ and C++/MPI programming models deal with such issues.

With the exception of the pure C++/MPI solution, each of the solutions managed to exploit all the available parallelism of the problem. The pure HSM solution and the two Fork95++ solutions also manage to handle arbitrary problem sizes which the HSM/C++/MPI and pure C++/MPI solutions were not able to do. This is a result of the difficulty of handling distributed data structures, though more flexible solutions are possible. Table 8.1 shows the results of the software metrics used on each solution, and Figure 8.15 gives a summary of the effectiveness of

Language	Lns	Ops	Clr	Cnc	Comments
pure HSM	90	46	clear	concise	Filter skeleton does not reflect algorithm.
HSM/Fork95++	142	76	clear	concise	Need to differentiate shared and private variables.
HSM/C++/MPI	243	165	-	long	Need redistribution functions in base model interface. Restricted machine size.
pure Fork95++	115	102	-	concise	High level concurrency is obscured.
pure C++/MPI	109	99	-	concise	Restricted parallelism. Restricted problem size.

Table 8.1: Summary of the software metrics for the image thresholding problem.

the languages.

The HSM superstructure uses a combination of map, reduce, and scan skeletons, and the pure HSM solution used additional zip, filter, and if skeletons. The pure HSM solution showed that it is possible to express the solution without using any ad hoc parallelism from the base models, but that this required using a filter skeleton to select the threshold value from the cumulative histogram. The selection more naturally requires the calculation for each element to query the values of the surrounding elements. This could have been done with a complicated reduce skeleton but would be more naturally coded using a map-with-halo skeleton as described in Chapter 4. The filter skeleton was not too unsuitable for this problem, but it highlights the difficulty in expressing parallelism when the available skeletons are unsuited to the task. The combination of the HSM language and the Fork95++ base language showed how the separation of the solution code into the upper skeleton superstructure and the lower base language functions can be used to code each part of the solution using the most appropriate language.

Using the HSM language with the C++/MPI base model was not as concise as we were expecting, but it does allow the programmer to exploit greater parallelism through the implementation of a complicated data redistribution which the pure C++/MPI code doesn't even attempt. The nested parallelism of the problem which was discouraged by the pure C++/MPI programming model is now intuitive and almost expected of the HSM/C++/MPI programmer. The provision of a set of collective routines to carry out standard data redistribution tasks would certainly make life easier for the base level programmer and would permit a more flexible solution allowing an arbitrary number of processors to take part in each reduce

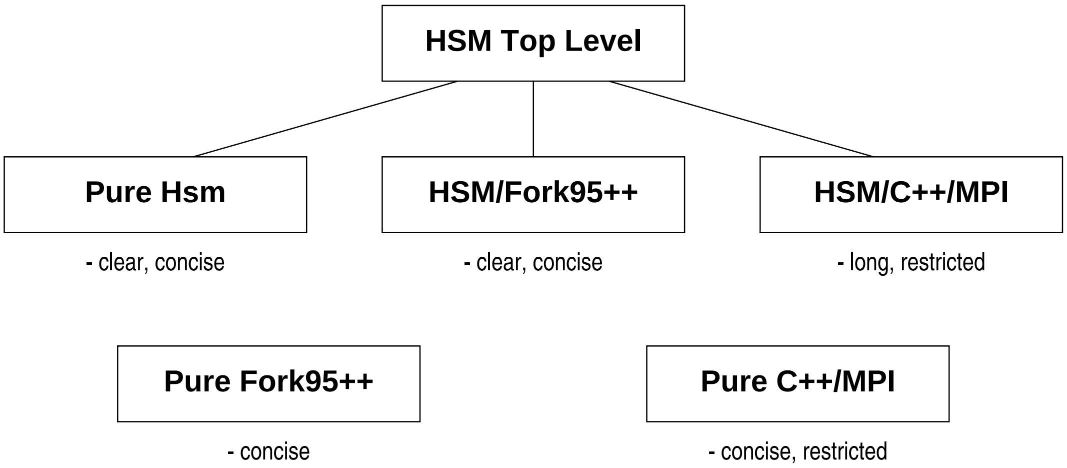

Figure 8.15: Language comparison summary for the image thresholding problem.

operation.

The pure Fork95++ programming model had no difficulty in exploiting all the parallelism available in the problem. The code is concise though it does rely on the availability of the **mpadd()** built-in function to perform the reduction of the histogram. This only works for the addition of integers, which is all that is needed for this particular problem, but if the reduction required a more elaborate operation the programmer would be required to write their own implementation of the reduce skeleton.

The pure C++/MPI solution was disappointing in that the restriction on the size of the problem is more significant than the restriction on the number of processors of the HSM/C++/MPI solution. The two solutions implement different functionality, though an equivalent HSM/C++/MPI solution would have been a fraction of the size, namely the half a dozen lines needed for the case of just one processor. The pure C++/MPI solution can, however, be compared with the HSM skeleton superstructure code against which the pure C++/MPI code comes off worse in terms of clarity and conciseness.

MPI does have the concept of group communicators to allow nested parallelism, but it is not possible to nest parallelism within the built-in skeletons such as MPI_Reduce(). To code a true solution would therefore involve implementing a true reduce skeleton as well as code equivalent to the HSM/C++/MPI base level functions of Appendix B.4. What is most revealing about this comparison is that the pure C++/MPI programming model effectively discourages the use of nested parallelism. A more sophisticated MPI_Reduce() skeleton would go some way to alleviating the difficulties, but this is simply introducing skeleton programming through another interface.

The last point to note is that all the above solutions have used rather simple data structures. One interesting comparison would be to see how the solutions would compare using a more compact data structure for the histogram. For example, the structure could be represented as a *list* of pixel intensity/count pairs instead of a sparse array. Writing the HSM base level C++/MPI code for this would be more complicated than for the array of integers, but it should be clear how it could be achieved. Writing the pure C++/MPI code would be significantly more challenging as the MPI_Reduce() operation would not be usable.

The problem solutions described in this chapter have shown that the HSM code is clear and well structured, and removes many of the more irritating details that the programmer has to take care of if programming purely in the base language. The combined skeleton/base language solutions show how more natural expressions of the solution can be used if an appropriate skeleton is not available and gives a taste of what could be done within the base node while still keeping a well structured skeleton hierarchy for the solution.

Chapter 9

The Merge-Sort Problem

Contents

9.1	The HSM Skeleton Superstructure	103
9.2	The Pure HSM Solution	105
9.3	The HSM/Fork95++ Solution	107
9.4	The HSM/C++/MPI Solution	107
9.5	The Pure Fork95++ Solution	111
9.6	The Pure C++/MPI Solution	113
9.7	Summary	115

Problem Description: *Given an array of n integers, sort the integers into increasing order by dividing the array into two sub-arrays of size $\frac{n}{2}$; sorting both sub-arrays concurrently; and merging the two sorted sub-arrays into a sorted array of n integers.*

The problems of merging and sorting are commonly discussed in parallel algorithm design and analysis research, and form a standard component of most recent text books on parallel computing, e.g. [Akl89], [KGGK94].

The merge-sort algorithm can be described using a divide and conquer paradigm, where the divide stage is trivial and the conquer stage does all the computation. The solution can therefore be broken down into two stages:

(a) **divide stage** — divide the input array into two sub-arrays of equal size and recursively sort each sub-array.

(b) **conquer stage** — take the two sub-arrays produced by the previous stage and merge them together into a single sorted array.

The divide stage performs no computation, but does partition the input array. The conquer stage is similar to the reduce operation of the image thresholding problem described in the previous chapter, where each reduce operation can be performed in parallel. The input array can be represented as a one-dimension array of integers.

Section 9.1 describes a top level skeleton formulation of the problem which is used in the first three solutions. Solutions are discussed using pure HSM in Section 9.2; HSM with a Fork95++ base level in Section 9.3; and HSM with a C++/MPI base level in Section 9.4. For comparison, solutions are coded without the HSM model using pure Fork95++ in Section 9.5; and pure C++/MPI in Section 9.6. The results are summarised and briefly discussed in Section 9.7. The code for all the solutions is given in Appendix C.

9.1 The HSM Skeleton Superstructure

The above solution outline for the problem uses a recursive formulation to break the original problem down into two smaller problems before merging the results together. The base case of the the recursion is reached when the array contains just a single element, i.e. already sorted.

This can be expressed naturally using a recursive function in the HSM language, as shown in Figure 9.1. The code requires two separate functions as the HSM `if` skeleton can only take function names or expressions for its arguments. Parallelism is introduced into the code through the use of the `map` skeleton to perform both recursive calls concurrently.

Although this recursion-based formulation is expressed entirely using the HSM top level language, it is not the only way to express the solution. An alternative approach is to consider the pattern of computation generated by the nested `map` skeletons of the divide-and-conquer paradigm. The computations can be separated into a single divide phase, which divides the input array into a list of the base cases, and a single reduce phase, which reduces all the base cases to a single array. Separating the dividing and conquering phases makes the pattern of computation and the inherent parallelism more explicit than that of the recursive formulation of Figure 9.1. Figure 9.2 shows the top level function using this more overtly skeleton-based formulation with the full listing given in Appendix C.1.

Figure 9.3 shows the function which performs the division of the array. The code

```
array<int> sort_fn( array<int> numbers )
{
    return(
        if ( array_size(numbers)[0]==1,
            numbers,                               /* base case */
            divide_and_merge_fn( numbers )         /* recursive case */
        )
    );
}

array<int> divide_and_merge_fn( array<int> numbers )
{
    int n;
    array<array<int>> list, new_list;
    array<int> new_numbers;

    n = array_size(numbers)[0];
    list = array_create( 1, 2 );
    list[0] = numbers[0..n/2-1];
    list[1] = numbers[n/2..n-1];
    new_list = map( sort_fn, NULL, list );
    new_numbers = merge_fn( NULL, new_list[0], new_list[1] );
    return( new_numbers );
}
```

Figure 9.1: HSM code for the merge-sort problem using a recursion-based formulation.

```
void main(void)
{
    array<int> numbers;
    array<array<int>> singletons;

    numbers = load_numbers_fn( "unsorted.data" );
    singletons = divide( divide_test_fn, divide_numbers_fn, NULL, numbers );
    numbers = reduce( merge_fn, NULL, singletons );
    save_numbers_fn( "sorted.data", numbers );
}
```

Figure 9.2: HSM code for the merge-sort problem using a skeleton-based formulation.

```
array<array<int>> divide_numbers_fn( void global, array<int> numbers )
{
    int n;
    array<array<int>> list;

    list = array_create( 1, 2 );
    n = array_size(numbers)[0];
    list[0] = numbers[0..n/2-1];
    list[1] = numbers[n/2.. n-1];
    return( list );
}
```

Figure 9.3: HSM code for dividing a one-dimensional array in half.

uses the array range operator to extract a sub-array containing the two halves of the original array.

The divide phase of the **main()** function is trivial in that it performs no computation or data redistribution. The instance of the divide skeleton may therefore be replaced by a map skeleton which maps each integer in the input array to a singleton array:

```
singletons = map( inttosingleton_fn, NULL, numbers );
```

This is likely to lead to a more efficient solution as it avoids the dynamic creation of multiple levels in the skeleton hierarchy. The divide skeleton is kept, however, as it can be used by the HSM/C++/MPI solution.

Language	HSM
Lines	86
Operators	12
Clarity	clear
Conciseness	concise
Comments	-

9.2 The Pure HSM Solution

The HSM superstructure leaves several functions to be expressed in either the pure HSM language or the chosen base language. The merge_fn() used in the reduce skeleton is required to take two sorted arrays and return a single sorted array created by merging the two input arrays. The sequential algorithm for performing this merge walks through both input arrays simultaneously, building up the output array by extracting the lower valued of the two head elements. This is explicitly sequential and doesn't lend itself to parallelisation.

One PRAM algorithm for merging two arrays takes every element of the first input array and finds the equivalent rank of the element in the second input array. The rank of the element in the output array will then be the sum of the ranks of the element in each input array. The complete output array is generated by performing the same calculations for the second input array.

Calculating the ranks of the elements can be done with a map skeleton on the one array using the other input array as a global argument, e.g.:

```
rank_a_in_a = array_create_indexed( 1, array_size( list_a )[0] );
rank_a_in_b = map( find_rank_fn, list_b, list_a );
rank_a_in_c = zip( int_add_fn, rank_a_in_a, rank_a_in_b );
```

All that is left to be done is the rearranging of the data using the calculated rank values, but this is a SIMD operation and not possible in the HSM language. Rearranging the data would require a complicated map skeleton which takes the entire array as a global argument and searches the array for the element of the required rank. This is neither efficient nor elegant.

A more suitable algorithm using skeletons employs a second divide-and-conquer design to divide the two input arrays such that the corresponding halves of the arrays can be merged together independently. The base case of the divide skeleton is reached when the two arrays either do not overlap or overlap on a single common element. Figure 9.4 shows an extract of the HSM top level code for merge_fn(), with the full listing including the definitions of divide_pair_test_fn() and divide_pair_fn() given in Appendix C.2. A pure HSM top level solution is therefore possible, but is significantly more complicated than the SIMD algorithm.

Language	pure HSM
Lines	177 (86+91)
Operators	76 (12+64)
Clarity	-
Conciseness	long
Comments	Skeletons cannot rearrange an array.

```
array<int> merge_fn( void global, array<int> a, array<int> b )
{
    original_pair.a = a;
    original_pair.b = b;
    pair_list = divide( divide_pair_test_fn, divide_pair_fn, NULL, original_pair );
    array_list = map( pairtoarray_fn, NULL, pair_list );
    final_array = reduce( arrayconcat_fn, NULL, array_list );
    return( final_array );
}
```

Figure 9.4: HSM code for merging two sorted arrays.

9.3 The HSM/Fork95++ Solution

The SIMD merging algorithm can be expressed quite naturally using the Fork95++ base level language with the HSM/Fork95++ interface, as shown in Figure 9.5. The rank value returned by find_rank_fn() needs to be tuned slightly to ensure a one-to-one mapping of the values when there are duplicates. Appendix C.3 shows the code for the asynchronous function find_rank_fn().

The HSM/Fork95++ solution is extremely concise, and follows the natural structure of the algorithms used.

Language	HSM/Fork95++
Lines	132 (86+46)
Operators	78 (12+66)
Clarity	clear
Conciseness	concise
Comments	-

9.4 The HSM/C++/MPI Solution

The pure HSM solution and the HSM/Fork95++ solution just described use a base case where each array contains only a single element. An alternative base case to consider is when each leaf group contains a single processor. For the HSM/C++/MPI combination in particular this would allow each processor to sort the local array using a standard sorting routine from a base language library, eg qsort() from stdlib. This is likely to lead to a more efficient implementation as it avoids the overhead of dynamically creating additional levels of the skeleton hierarchy when the reduce operator is sequential.

```
sync HSM_array<int>& merge_fn(
    HSM_void& global, HSM_array<int>& a, HSM_array<int>& b
) {
    pr int i, rank;
    HSM_array<int>& c=*(shnew HSM_array<int>);

    c.create( 1, a.size(0)+b.size(0) );
    for ( i=$, i<a.size(0); i+=groupsize() ) {
        rank = find_rank_fn( b, 0, b.size(0)-1, a(i) );
        while ( rank>0 && a(i)==b(rank-1) ) rank--;
        c(rank+i) = a(i);
    }
    for ( i=$, i<b.size(0); i+=groupsize() ) {
        rank = find_rank_fn( a, 0, a.size(0)-1, b(i) );
        while ( rank<a.size(0) && b(i)==a(rank) ) rank++;
        c(rank+i) = b(i);
    }
    return( c );
}
```

Figure 9.5: HSM/Fork95++ code for merging two sorted arrays.

To allow for this change in base case, the programmer needs to redefine the HSM function divide_test_fn(), as shown in Figure 9.6. The code uses the built-in HSM function node_size() to determine how many processors are grouped together in the current leaf node of the hierarchy.

The base case arrays produced by this amended divide test are no longer sorted, so each array in the list needs to be explicitly sorted before the list can be reduced. This is done by adding a `map` skeleton between the divide and reduce phases, as shown in Figure 9.7. The code for seq_sort_fn() is given in Appendix C.4.

```
bool divide_test_fn( void global, array<int> alpha )
{
    return( node_size()>1 && array_size(alpha)[0]>1 );
}
```

Figure 9.6: HSM code for deciding whether or not to divide the problem further, using a sequential sort for the base case.

```
void main(void)
{
    array<int> numbers;
    array<array<int>> base_cases;

    numbers = load_numbers_fn( "unsorted.data" );
    base_cases = divide( divide_test_fn, divide_numbers_fn, NULL, numbers );
    base_cases = map( seq_sort_fn, NULL, base_cases );
    numbers = reduce( merge_fn, NULL, singletons );
    save_numbers_fn( "sorted.data", numbers );
}
```

Figure 9.7: HSM code for the merge-sort problem using a sequential sort for the base case.

The Merge Function

The C++/MPI base level code cannot use the same algorithm as the Fork95++ solution as it is not possible to perform multiple binary searches concurrently. The binary search would be possible if each processor had access to the entire array, e.g. by broadcasting the array to every processor, but the solution worked on here avoids the data replication which would otherwise be necessary.

On entry to the merge_fn() the distribution of the input arrays is such that the first half of the processor group holds the first array and the second half holds the second array, as was the case with the image thresholding code. For each array, N elements are selected as potential pivots spaced $\lfloor \frac{n}{N} \rfloor$ apart, where n is the number of elements and N the number of processors[1]. The $2N$ potential pivots are then sorted and every second element is chosen to be a final pivot element. This ensures a reasonable partitioning of the output array.

To compute the output array the processors redistribute the two input arrays so that each processor holds all the elements less than that processor's chosen pivot element and greater than those that will be held by the neighbouring left hand processor. Figure 9.8 shows an example of merging two arrays of size eight over a group of four processors.

The base level C++/MPI code uses several collective MPI routines to carry out the merge operation. Firstly, it uses an MPI_Allgather() operation to collect and

[1] The algorithm in [Akl89] only selects $2(N-1)$ elements using a spacing of $\lceil \frac{n}{N} \rceil$, but the HSM/C++/MPI code is much simpler if $2N$ elements are considered.

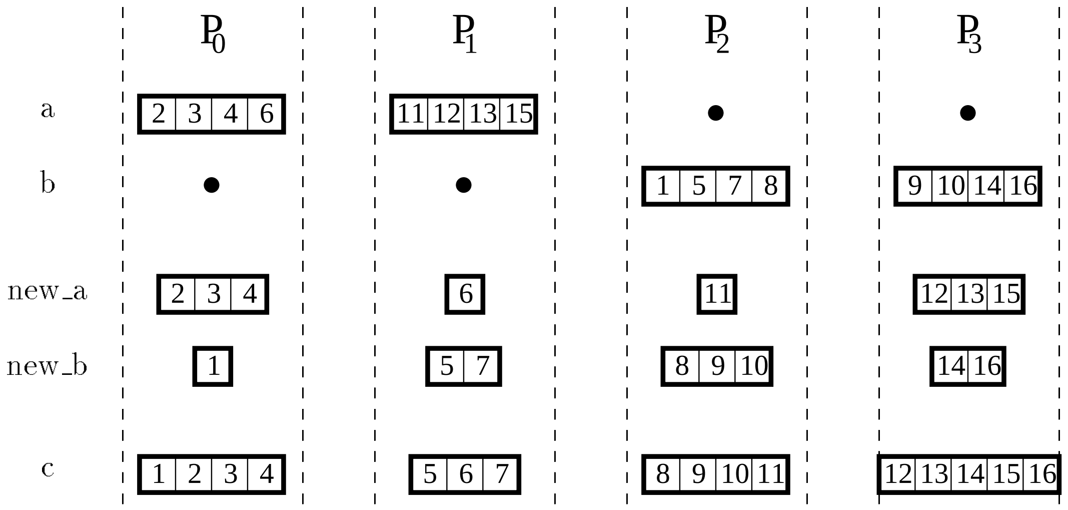

Figure 9.8: Example of two arrays of size eight being merged together. The initial pivots are the numbers 3, 6, 12, 15 from a and 5, 8, 10, 16 from b. The chosen pivots are 5, 8, 12.

broadcast the initially selected pivot elements which are then sorted by each processor using the standard library qsort() function. Once the final pivot elements are chosen each processor sends the appropriate array elements to the desired processors using a buffered send operation. An MPI_Alltoall() operation is used to inform each processor how many elements to expect from each other processor, followed by an MPI_Allgather() operation so that each processor knows how many elements are distributed on the processors to the left of it. This value indicates the index of the first local element of the newly distributed array.

Before the elements can be received the new array has to be created. This is done using the array.create() member function and specifying a distribution of type HSM_DISTRIBUTION_ADHOC. The array is created of the required size, but the location of the elements is not set. Once the programmer knows which elements will be stored on the local processor the indices are added to the data structure using the array.add_local() member function. The processor can then receive the elements from each of the other processors and merge the local portions of the two arrays. Figure 9.9 shows an extract of the code for creating and receiving an array, with the full code listing given in Appendix C.4.

The HSM/C++/MPI solution manages to exploit all the parallelism available in the algorithm although, as with the image thresholding problem, most of the code is needed to redistribute the data structures. The advantage is that the programmer has significant flexibility in passing the data among the processors

```
HSM_array<int> new_a;
new_a.create( 1, a.size(0), HSM_DISTRIBUTION_ADHOC );
for (i=0; i<count_a; i++) {
    new_a.add_local(offset_a+i);
}

for (i=0, offset=offset_a; i<size; i++) {
    if (recv_block_size[i]>0) {
        MPI_Recv( &new_a(offset), recv_block_size[i], MPI_INT,
                 i, 0, HSM_COMM_WORLD, &status );
        offset+=recv_block_size[i];
    }
}
```

Figure 9.9: HSM/C++/MPI code for creating an ad-hoc array and receiving blocks of elements from the other processors.

and can exploit established libraries of sequential algorithms.

Language	HSM/C++/MPI
Lines	247 (86+161)
Operators	101 (12+89)
Clarity	clear
Conciseness	long
Comments	Need redistribution functions in base model interface.

9.5 The Pure Fork95++ Solution

The pure Fork95++ solution can be coded using just the conquer stage of the outline solution given at the start of the chapter. The implementation is simplified by restricting the algorithm to the case where the number of elements to be sorted is equal to the number of processors in the shared memory machine. This makes the actual merging of the two arrays more straightforward and avoids the necessity to perform a sequential sort in the base case. The implementation can be extended to cover the general case but the code becomes significantly more complicated.

The main() function performs a **while**{} loop to iterate through the $\log n$ steps, where n is the total number of processors in the machine. Within each step the machine is **fork**{}ed into $\frac{n}{s}$ separate groups, where s is the size of each group

```
void main(void) {
    pr int rank;
    start {
        sh int local_size=1;
        while (local_size<=groupsize()) {
            fork( numbers.n/local_size; @=$%(numbers.n/local_size); ) {
                if ($%local_size < local_size/2) {
                    /* processor is in left half of block */
                    block_start=($/local_size)*local_size+local_size/2;
                    rank = find_rank_fn( ... );
                    rank += $;
                }
                else { /* processor is in right half of block */ }
                numbers[rank]=numbers[$];
            }
            local_size*=2;
        }
    }
}
```

Figure 9.10: Pure Fork95++ code for forking the machine into separate groups and performing the merge reduction.

which starts off with a value of 1 and doubles after each step. The processors within each group remain synchronised, but the separate groups execute asynchronously. On completion of the **fork**{} all the processors synchronise before executing the next step of the **while**{}. Figure 9.10 shows an extract of the code.

The pure Fork95++ listing, given in Appendix C.5, is fairly concise, but is restricted to the rather simple case when the number of elements to be sorted is the same as the number of processors. Even with this simplification the calculations for which block a process should be working on are extremely complicated and difficult to follow. The use of a reduce-style algorithm is also totally obscured by the processor rank and group manipulations.

Language	pure Fork95++
Lines	86
Operators	140
Clarity	unclear
Conciseness	concise
Comments	High level concurrency is obscured. Restricted problem size.

```
void main(int argc, char *argv[]) {
    MPI_Comm local_comm;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(HSM_COMM_WORLD, &rank);
    MPI_Comm_size(HSM_COMM_WORLD, &size);

    for ( local_size=1; local_size<size; local_size*=2 ) {
        MPI_Comm_split( MPI_COMM_WORLD, rank/local_size, rank,
                        &local_comm );
        merge_fn( ... , local_comm );
        MPI_Comm_free( &local_comm );
    }
    MPI_Finalize();
}
```

Figure 9.11: Pure C++/MPI code for creating separate communicators for each leaf group in the reduce skeleton and performing the merge operation within each group.

9.6 The Pure C++/MPI Solution

The pure C++/MPI solution also implements just the conquer phase of the original outline solution, but allows for there to be more elements than processors. The number of elements is restricted to being an exact multiple of the number of processors so that the difficulties of calculating the block sizes and offsets required by MPI_Scatterv() function are avoided.

Performing the reductions in parallel is straightforward, and the reduce operations may also be internally parallelised following the breakdown used by the HSM/C++/MPI solution. Figure 9.11 gives an extract of code which shows how separate communicators can be created for each leaf group so that the merge function can be carried out independently within each group. Message passing within each group would then be similar to that illustrated in Figures 9.8 and 9.9 for the HSM/C++/MPI solution.

Since the pure C++/MPI code for this approach will be very similar to the HSM/C++/MPI base level code already described, an alternative solution is implemented which performs each reduce operation sequentially on only one processor in the group. This requires one other processor in the leaf group to send its entire array to the local root processor, with the resulting sorted array ending up

```
for ( jump=1; jump<size && rank%jump==0; jump*=2 ) {
    /* find out who the neighbour is */
    if (rank%(jump*2)==0) neighbour=rank+jump;
    else neighbour=rank-jump;

    if (neighbour<rank) {
        /* just send the entire array */
        MPI_Send( &local_numbers[0], nelems/size*jump, MPI_INT,
                  neighbour, 0, MPI_COMM_WORLD );
    }
    else {
        /* receive the neighbours array */
        MPI_Recv( &tmp_numbers_a[0], nelems/size*jump, MPI_INT,
                  neighbour, 0, MPI_COMM_WORLD, &status );
        /* merge the two arrays */
    }
}
```

Figure 9.12: Pure C++/MPI code for performing each reduce operation sequentially.

entirely on the root processor of the group. All the other local group processors processors lie idle while the reduction is carried out. Figure 9.12 shows an extract of the code necessary to calculate to which processor the data should be sent.

The pure C++/MPI solution is concise, but the use of the reduce-style algorithm is not visible in the code. The calculation of the rank of the neighbour is also an added complication for the programmer. Although the solution given does not exploit all the available parallelism it can be seen that this would be possible using the code fragment of Figure 9.11 and a merge_fn() similar to that from the HSM/C++/MPI solution.

Language	pure C++/MPI
Lines	97
Operators	168
Clarity	-
Conciseness	concise
Comments	High level concurrency is obscured. Restricted parallelism. Restricted problem size.

Language	Lns	Ops	Clr	Cnc	Comments
pure HSM	177	76	-	long	Skeletons cannot rearrange an array.
HSM/Fork95++	132	78	clear	concise	-
HSM/C++/MPI	247	101	clear	long	Need redistribution functions in base model interface.
pure Fork95++	86	140	unclear	concise	High level concurrency is obscured. Restricted problem size.
pure C++/MPI	97	168	-	concise	High level concurrency is obscured. Restricted parallelism. Restricted problem size.

Table 9.1: Summary of the software metrics for the merge-sort problem.

9.7 Summary

The merge-sort algorithm is most naturally expressed as a recursive function where the two recursive calls can be executed concurrently. As was shown in the solutions presented in this chapter the algorithm can also be expressed by separating the dividing and conquering phases, and also by flattening out the recursion.

The parallelism of the recursion-based solution is contained within the single `map` skeleton instance in each level of recursion, but it is not immediately apparent that this results in a branching tree of concurrency. The complete separation of the divide and conquering phases in the skeleton-based solution makes it quite clear what parallelism is inherent in the algorithm but sacrifices the link between the dividing and the conquering steps of each level of recursion. The third approach of flattening out the recursion shows that parallelism is being exploited, but the solution lacks the clarity of *how* it is being exploited, i.e. which skeletons; or *why* it is being exploited, i.e. the recursion. Table 9.1 shows a summary of the software metrics for each of the languages/combinations.

A fully parallel solution using pure C++/MPI code is little different from the HSM/C++/MPI solution. The pure C++/MPI programmer has the extra work of manipulating the local communicator and implementing the iterations of the reduce skeleton, but otherwise the code is quite similar. One difference, however, is that the pure C++/MPI code is restricted to the case where the number of elements is an exact multiple of the number of processors. Allowing for arbitrary sizes of

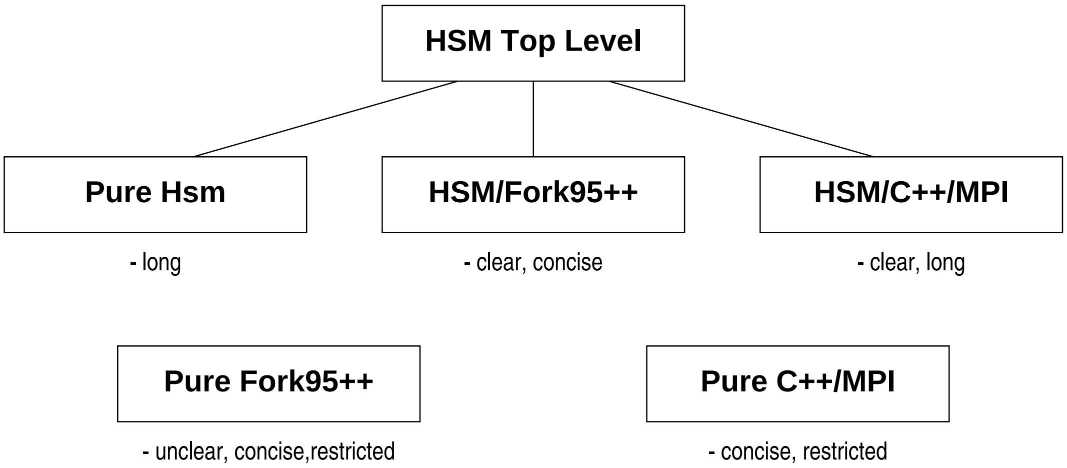

Figure 9.13: Language comparison summary for the merge-sort problem.

arrays would introduce various additional calculations which the HSM/C++/MPI code already performs. The HSM/C++/MPI code also makes it clear where skeletons are being employed and what functions are used to instantiate the skeleton. Replacing the trivial divide phase with something more complicated is therefore clearly possible in the HSM/C++/MPI code, but would require greater work when programming in pure C++/MPI.

The pure Fork95++ code flattens the parallelism of the recursion and requires complicated expressions to calculate which block of numbers each processor is working on. Even after restricting the problem to match the size of the shared memory machine the manipulations of processor rank and group identifier are not very intuitive and it is difficult to observe that a reduce skeleton has been implemented. This compares unfavourably with the HSM/Fork95++ code which expresses the solution clearly and concisely, although the pure Fork95++ solution could be made more clear by using separate sync functions and the recursive formulation of the problem.

The HSM/Fork95++ code also compares favourably against the pure HSM code, due mainly by the lack of an appropriate use of skeletons for the merge operation. The pure HSM solution would benefit from a form of `permute` skeleton which takes an array of elements and an array of indices, returning the permuted array of elements. The `permute` skeleton performs no computation, but encapsulates an irregular communication pattern over the array.

The HSM/C++/MPI solution makes an interesting contribution to the evaluation of the HSM model by making the base case of the `divide` skeleton dependent on the parameters of the base machine at that point in the hierarchy. This is a departure from the purely functional code of the pure HSM and HSM/Fork95++ solution but allows the programmer to exploit a sequential library function. Suitable parallel library functions can also be exploited by stopping the division while the local group still contains several processors. The programmer would simply have to ensure that the data existed in the format and distribution required by the library before calling the function.

As was found in the HSM/C++/MPI solution for the image thresholding problem, the redistribution of data is a non-trivial task. In the merge-sort problem it would be possible to broadcast the entire array to everyone from which each processor could access the required elements. Broadcasting the distributed arrays in this manner is much simpler than calculating where individual blocks should be sent to or received from, and may not necessarily lead to a less efficient solution. Broadcasting two arrays may well take less time than performing the three collective and n^2 point-to-point communications required by the chosen solution, at the cost of increased storage requirements.

The above HSM solutions have shown that skeletons can be used to code clear and well structured algorithms in the HSM top level language, although the top level language may not be the most suitable way of expressing some of the component stages of the solution. Combining the top level with a separate base language allowed a concise implementation of the component stages without sacrificing flexibility. In comparison to the two pure solutions the HSM code retains its structure and avoids unnecessary base level details.

Chapter 10

The All-Pairs Shortest Paths Problem

Contents

10.1 The HSM Skeleton Superstructure 119
10.2 The Pure HSM Solution 121
10.3 The HSM/Fork95++ Solution 124
10.4 The HSM/C++/MPI Solution 126
10.5 The Pure Fork95++ Solution 129
10.6 The Pure C++/MPI Solution 129
10.7 Summary 132

Problem Description: *Given a directed graph of n vertices with weighted edges, calculate the weights of the shortest paths between all pairs of vertices.*

The all-pairs shortest paths problem is interesting in that the solution can be naturally expressed in terms of concurrently solving the single-source shortest paths problem. The single-source problem is also potentially parallelisable which leads to a two level hierarchy of parallel decomposition using Dijkstra's algorithm [KGGK94] at the lower level. Although the actual shortest paths are calculated during the course of the algorithm, only the weights of those paths are stored and returned. This leads to a much simpler solution while retaining the computational characteristics of the original and without giving rise to additional complications due to manipulating further data structures.

The solution can be considered separately for the two levels:

(a) **all-sources** — decompose the problem into n instances of the single source problem.

(b) **single-source** — find the weights of the shortest paths from the given source to all the other vertices.

The weighted graph can be represented as an adjacency matrix, i.e. two-dimensional array, with element $a_{<i,j>}$ giving the weight of the edge connecting vertices v_i and v_j, or ∞ if no edge exists. An undirected graph can be handled by setting $a_{<j,i>}$ to hold the same weight as $a_{<i,j>}$, i.e. making the adjacency matrix symmetrical across the centre diagonal.

The weights of the shortest paths can also be represented as a two-dimensional array, with element $a_{i,j}$ giving the weight of the shortest path connecting vertices v_i and v_j. For simplicity the graph is assumed to be connected so that shortest paths exist for all the pairs of vertices.

Section 10.1 describes a top level skeleton formulation of the problem which is used in the first three solutions. Solutions are discussed using pure HSM in Section 10.2; HSM with a Fork95++ base level in Section 10.3; and HSM with a C++/MPI base level in Section 10.4. For comparison, solutions are coded without the HSM model using pure Fork95++ in Section 10.5; and pure C++/MPI in Section 10.6. The results are summarised and briefly discussed in Section 10.7. The code for all the solutions is given in Appendix D.

10.1 The HSM Skeleton Superstructure

The top level "**all-sources**" problem can be expressed as a `map` skeleton of single-source problems as no interactions are needed between any of the single-source calculations. The `map` body function is applied to each vertex in the graph with the graph's adjacency matrix being passed as a global parameter, common to all the calculations. A vertex can be represented by the index of its adjacency entries in the adjacency matrix, so a one-dimensional array containing all the vertices can be built by creating an indexed vector[1] with the same number of elements as the adjacency matrix has rows (or columns). The `map` skeleton applies the single-source shortest paths calculation to each element of the indexed vector, generating a one-dimensional array of weight matrices. Each weight matrix is a two-dimensional array, though it only has one row.

The single-rowed two-dimensional arrays are then concatenated together using a

[1]An indexed vector has integer elements which, in this case, is more suitable than the one-tuple elements that an indexed one-dimensional array would have.

```
void main(void)
{
    array<int> graph, vertices, weights;
    array<array<int>> adjacency_rows;

    graph = load_graph_fn( "graph.data" );
    adjacency_rows = map( single_source_fn, graph,
                          vector_create_indexed( array_size(graph)[0] ) );
    weights = reduce( arrayconcat_fn, NULL, adjacency_rows );
    save_graph_fn( "weights.data", weights );
}
```

Figure 10.1: HSM code for the all-pairs shortest paths problem.

reduce skeleton to form the final two-dimensional array of shortest path weights[2]. Figure 10.1 shows the HSM code for the skeleton superstructure.

The lower level, "**single-source**" problem uses Dijkstra's algorithm to build a vertex tree V_T, starting with the source node and adding one vertex at a time. The algorithm keeps track of the minimum costs required to reach each vertex from the source vertex by passing through only vertices in V_T and selects the vertex which can be reached with minimum cost. When this vertex is added to V_T the minimum costs are updated to reflect the possibility that it may be possible to reach a vertex more cheaply through the added node than it was before.

Using the skeleton superstructure of the **all-sources** algorithm, the **single-source** algorithm can be expressed using the ad hoc parallelism of the base models, as will be described in Sections 10.3 and 10.4, or using purely the HSM language as described next. The full listing of the superstructure code is given in Appendix D.1.

Language	HSM
Lines	25
Operators	12
Clarity	clear
Conciseness	concise
Comments	Need separate function to 'wrap' the simple concatenation operator.

[2]If the single rowed matrices were one-dimensional the reduce skeleton would concatenate them all into one, very long, one-dimensional array.

```
array<int> single_source_fn( array<int> graph, int vertex )
{
    int n;
    array<vertex_t> work;
    array<int> weights;

    n = array_size(graph)[0];
    work = zip( costtovertex_fn, NULL,
                vector_create_indexed( n ), graph[vertex][0..n−1] );
    work[vertex].used = TRUE;
    work = while( more_paths_fn, add_vertex_fn, graph, work );
    weights = array_create( 2, 1, n );
    weights[0][0..n−1] = map( vertextocost_fn, NULL, work );
    return( weights );
}
```

Figure 10.2: HSM code for the single-source shortest paths problem.

10.2 The Pure HSM Solution

Expressed in the pure HSM language, the single_source_fn() uses a while skeleton to add one vertex to the tree V_T on each iteration, as shown in the code extract in Figure 10.2. The vertex tree V_T and the minimum cost array are represented together as an array of vertex_t compound structures. Each vertex_t contains one boolean and two integer members representing: whether the vertex is in V_T; the index of the vertex; and the minimum cost of reaching the vertex using only vertices already in V_T. This one-dimensional array of vertices is created by zipping an indexed vector together with the relevant row of the adjacency matrix.

The while skeleton takes a boolean test function to decide when to stop looping, and a body function to be executed on each iteration. The body function, add_vertex_fn(), takes as arguments the graph's adjacency matrix and the array of vertices and returns the updated array of vertices. The test function, more_paths_fn(), takes the same two arguments as the body function and returns true while there are still more paths to be found, although the adjacency matrix is not actually used in the calculation. Lastly, the shortest path weights in the minimum cost array are extracted using a map skeleton with the vertextocost_fn() to return the array of integers. The values are assigned to the first row of the array weights so that a single-rowed two-dimensional array is returned, as required by the top level HSM **main()** function.

```
array<vertex_t> add_vertex_fn( array<int> graph, array<vertex_t> work )
{
    extended_graph_t current;
    array<vertex_t> new_work;
    current.graph = graph;
    current.vertex = reduce( min_vertex_fn, NULL, work );
    new_work = map( update_vertex_fn, current, work );
    new_work[current.vertex.index].used = TRUE;
    return( new_work );
}
```

Figure 10.3: HSM code for adding the next vertex to V_T and updating the data structures, taking the graph's whole adjacency matrix for its global argument.

```
vertex_t update_vertex_fn( extended_graph_t current, vertex_t old_vertex )
{
    updated_vertex = old_vertex;
    updated_vertex.cost = updated_vertex.cost
                        + current.graph[current.last_vertex.index][old_vertex.index];
    return( if( old_vertex.used || old_vertex.cost<=updated_vertex.cost,
                old_vertex, updated_vertex ) );
}
```

Figure 10.4: HSM code extract for updating a vertex using the graph's adjacency matrix as a global argument.

The body of the while loop

The work of the while skeleton body is performed by the add_vertex_fn() function, shown in Figure 10.3. The function uses a reduce skeleton on the array of vertices to find the single vertex with lowest cost which has not yet been added to V_T. Each vertex is then updated using a map skeleton with the function update_vertex_fn() which takes the adjacency matrix and the latest added vertex together as a single global argument.

The more_paths_fn() also uses a reduce skeleton on the array of vertices, but this time to find any element that is still unused. This can be done by the same reduction used to find the vertex with lowest cost, with the result that there is a duplication of work between the termination test function and the body of the while loop. Unfortunately this cannot be avoided without passing data between

```
array<vertex_t> add_vertex_fn( array<int> graph, array<vertex_t> work )
{
    array<vertex_edge_t> augmented_work;
    new_vertex = reduce( min_vertex_fn, NULL, work );
    augmented_work = zip( augmented_work_join_fn, NULL,
                          graph[new_vertex.index][0..n-1], work );
    augmented_work[new_vertex.index].vertex.used = TRUE;
    new_work = map( update_work_fn, new_vertex, augmented_work );
    return( new_work );
}
```

Figure 10.5: HSM code extract for adding the next vertex to V_T and updating the data structures, using no global argument.

```
vertex_t update_vertex_fn( vertex_t new_vertex, vertex_edge_t local )
{
    updated_vertex = local.vertex;
    updated_vertex.cost = updated_vertex.cost + local.edge_cost;
    return( if( local.vertex.used || local.vertex.cost<=updated_vertex.cost,
                local.vertex, updated_vertex ) );
}
```

Figure 10.6: HSM code extract for updating a vertex using only the latest added vertex as a global argument.

the loop's test function and the loop's body function.

The vertex array update function

The update_vertex_fn() calculation, shown in Figure 10.4, only uses one entry of the adjacency matrix, the entry being the cost of reaching the desired node from the recently added node. It is therefore unnecessary, and possibly inefficient, to pass the whole adjacency matrix as a global parameter. The algorithm can be recoded to pass each calculation only the required information, as shown in the code fragments of Figures 10.5 and 10.6. A zip skeleton is used to join each vertex of the work array with the cost of the edge connecting the new vertex with it.

The first solution shows how it is much easier to pass all the relevant data as a single global argument, but that this may lead to a less efficient solution. Separating out the required information from the global information is more complicated, but it leads to a more natural update_work_fn() as well as giving the potential of a more efficient solution. The complexity arises from the requirement to define a new data type, vertex_edge_t, and having to work out the arguments for an additional skeleton instance, the zip skeleton to join the relevant data elements together.

The complete code for the pure HSM solution, listed in Appendix D.1, uses the latter formulation to avoid the passing of unnecessary global data. The code is clear and straightforward, but is not very concise. The main disadvantage of the code is that there are duplicated calculations between the termination test and the body of the while loop.

Language	pure HSM
Lines	140 (25+105)
Operators	76 (12+64)
Clarity	clear
Conciseness	long
Comments	Duplication of code in test and body of while skeleton.

10.3 The HSM/Fork95++ Solution

Dijkstra's **single-source** algorithm can also be expressed in the Fork95++ language using the HSM/Fork95++ interface , as shown in the code extract of Figure 10.7. One minor complication encountered in the code is that Fork95++ does not have a multi-prefix minimum operator, so **mpmax()** is used instead with the arguments negated. Unlike the previous use of multi-prefix operators, the location of the minimum element is important. This is calculated by comparing the local cost with the calculated minimum cost and setting the location appropriately. If several elements have the same minimum value a write-conflict occurs which may be resolved arbitrarily by the underlying machine.

The HSM/Fork95++ code differs from the pure HSM code in that the reduce operator is now a simple **mpmax()** operation which cannot check if the vertex has already been used or not. The check is performed in an **if**{} statement resulting in an undefined number of processors taking part in the reduction, but this is not a problem for the Fork95++ language or the shared memory machine.

The updating of the minimum cost array is performed as expected, and finally the multi-prefix bitwise AND operation is used to see if there are any unused vertices left. Since the graph is assumed to be connected the check could have

```
sync HSM_array<int>& single_source_fn(
    HSM_array<int>& graph,      /* global argument */
    HSM_int& vertex
) {
    pr int i;
    sh int *used, finished=0, mincost, minpos;
    HSM_array<int>& weights=*(shnew HSM_array<int>);

    weights.create( 2, 1, graph.size(0) );
    used = shnew int[graph.size(0)];
    for ( i=$; i<graph.size(0); i+=groupsize() ) {
        used[i]=0;    weights(i)=graph(vertex,i);
    }

    while ( !finished ) {
        mincost=INT_MIN;
        for ( i=$; i<graph.size(0); i+=groupsize() ) {
            if ( !used[i] ) {
                mpmax( &mincost, -weights(i) );
                if (mincost==-weights(i)) minpos=i;
            }
        }
        used[minpos]=TRUE;

        for ( i=$; i<graph.size(0); i+=groupsize() ) {
            if (!used[i] && weights(i)>=weights(minpos)+graph(minpos,i)) {
                weights(i)=weights(minpos)+graph(minpos,i);
            }
        }

        for ( i=$; i<graph.size(0); i+=groupsize() ) {
            mpand( &finished, used[i] );
        }
    }
    return( weights );
}
```

Figure 10.7: HSM/Fork95++ code for Dijkstra's **single-source** algorithm.

been performed using a simple count of the number of vertices used so far, but the code illustrates how the **mpand()** operator can be used instead.

The complete code for the HSM/Fork95++ solution, listed in Appendix D.3, is clear and concise, although the use of the multi-prefix operations is more complicated than would be the case if there were the same number of data elements as processors.

Language	HSM/Fork95++
Lines	68 (25+43)
Operators	78 (12+66)
Clarity	clear
Conciseness	concise
Comments	Use of multi-prefix operator is more complicated when handling arbitrary array sizes.

10.4 The HSM/C++/MPI Solution

It is also possible to express Dijkstra's **single-source** algorithm in C++/MPI using the HSM/C++/MPI interface although, as with both the image thresholding problem and the merge-sort problem, the base level programmer has to be concerned with the distribution of the data structures. Figure 10.8 shows an extract of the loop code of the single_source_fn() function.

Calculating the minimum cost vertex is straightforward, and is done using the MPI reduce function with the MINLOC operator. This operator keeps track of the location, or index, of the successful element.

Updating the minimum cost array should also be straightforward, but unfortunately the standard distributions of the minimum cost array and the adjacency matrix are not compatible. Figure 10.9 illustrates the standard distributions of an eight node graph's adjacency matrix and minimum cost array over four processors. As can be seen the adjacency matrix uses a row-block distribution while the minimum cost array uses what is effectively a column-block distribution.

This incompatibility of the distributions could be solved by performing a single redistribution of the adjacency matrix into a column-block format but, as will be shown for the pure C++/MPI solution in Section 10.6, the calculations involved are long winded. Instead, the adjacency matrix is scattered one row at a time and only when needed. No data is sent more than once but the overhead of the multiple scatter operations is likely to be greater than that of a single all-to-all

```
for ( iteration=n-1; iteration>0; iteration-- ) {
    valloc[0] = INT_MAX;
    for ( ind = used.first_local(); ind <= used.last_local();
          ind = used.next_local() ) {
        if (!used.element(ind) && mincost.element(ind)<valloc[0]) {
            valloc[0]=mincost.element(ind);
            valloc[1]=ind(0);
        }
    }
    MPI_Allreduce( valloc, minvalloc, 1, MPI_2INT,
                   MPI_MINLOC, HSM_Comm_world );
    vertex = minvalloc[1];

    /* add the minimum vertex to V_T */
    if ( used(vertex).is_local() ) used(vertex)=1;

    /* scatter the appropriate row of the adjacency matrix */
    MPI_Scatter( &graph( vertex, 0 ), n/size, MPI_INT,
                 &adjmatrow.first_local(), n/size, MPI_INT,
                 graph( vertex, 0 ).where_is(), HSM_Comm_world );

    /* update the minimum cost matrix */
    for ( ind = used.first_local(); ind <= used.last_local();
          ind = used.next_local() ) {
        if (!used.element(ind)
            && mincost.element(ind)>mincost(vertex)+adjmatrow(ind)) {
            mincost.element(ind)=mincost(vertex)+adjmatrow(ind);
        }
    }
}
```

Figure 10.8: HSM/C++/MPI code extract for Dijkstra's **single-source** algorithm.

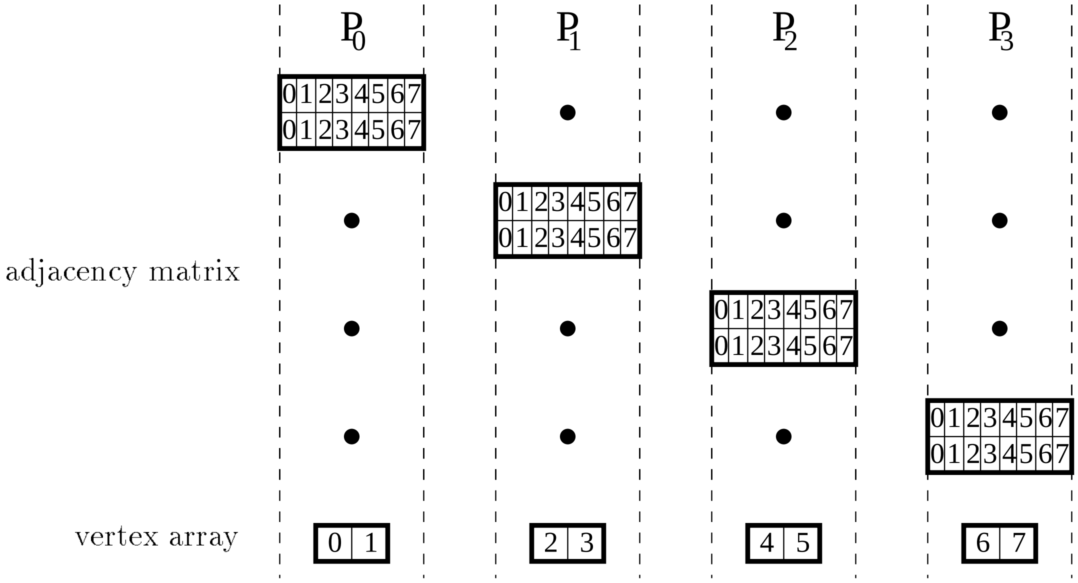

Figure 10.9: Example of a single-source calculation of an eight node graph over four processors. The adjacency matrix uses the standard row-block distribution.

data distribution.

Another restriction in the code extract is that, by using the simpler MPI_Scatter() operation instead of MPI_Scatterv(), the number of vertices in the graph is restricted to an exact multiple of the number of processors. Both of these problems would be removed if the base level programmer had access to collective functions for redistributing the HSM_array<> structures and if a column-block distribution was one of the standard patterns.

Despite these problems the base level C++/MPI code, listed in Appendix D.4, is clear and well structured. One minor hindrance to the clarity is that the arrays are sometimes accessed using HSM_index structures, e.g. array.element(ind), and sometimes just with the integer indices, e.g. array(i). This is unfortunate but difficult to avoid without replacing all the integer indices with HSM_index structures, which may also obscure the algorithm. On balance the code is clear and concise.

Language	HSM/C++/MPI
Lines	81 (25+56)
Operators	101 (12+89)
Clarity	clear
Conciseness	concise
Comments	Uses two different methods to index arrays. Restricted number of vertices.

10.5 The Pure Fork95++ Solution

The pure Fork95++ solution can be coded along similar lines to the HSM/Fork95++ solution. An additional outermost loop is needed to implement the top level `map` skeleton to solve the n single-source problems concurrently, as shown in Figure 10.10.

This is done with a **fork**{} structure, setting the group identifier to the index of the respective single source vertices, and resetting the processor identifiers to number consecutively from zero within each local group.

To avoid confusion an array of shared parameters is created over the whole machine with one entry for each group. It is possible to use the *scope* of the variable declaration to define the locality of a shared variable but this requires significant care in understanding the synchronicity of the processors and groups of processors at each level of the program. Declaring distinct variables for each group is therefore less ambiguous, but it means that each time a group variable is accessed extra array indexing notation is needed. This leads to a profusion of array`[@]` type references throughout the program.

The code, listed in Appendix D.5, is not much longer than the HSM/Fork95++ base level, but the concurrency of the multiple single-source problems is lost amid the code and the use of the one and two dimensional arrays makes the algorithm look more complicated than it actually is.

Language	pure Fork95++
Lines	85
Operators	140
Clarity	unclear
Conciseness	concise
Comments	High level concurrency is obscured. The use of multiple arrays to enforce scope is confusing.

10.6 The Pure C++/MPI Solution

The pure C++/MPI solution follows the solution of the pure Fork95++ code, but is significantly longer due to the MPI communicator manipulations and data distributions. To allow for a single redistribution of the adjacency matrix, the solution creates new MPI data types to represent one row of a column-striped

```
void main(void)
{
    pr int i;
    start {
        sh array2d_t graph, mincost, used;
        sh array1d_t minval, minpos;
        ...
        fork ( graph.rows(); @=$%graph.rows(); $=$/graph.rows() ) {
            for ( i=$; i<graph.rows(); i+=groupsize() ) {
                used[@,i]=0;
                mincost[@,i]=graph[@,i];
            }
            used[@,@]=1;

            finished[@]=0;
            while ( !finished[@] ) {
                minval[@]=INT_MIN;
                for ( i=$; i<graph.rows(); i+=groupsize() ) {
                    if ( !used[@,i] ) {
                        mpmax( &minval[@], -mincost[@,i] );
                        if ( minval[@]==-mincost[@,i] ) {
                            minpos[@]=i;
                        }
                    }
                }
                used[@,minpos[@]]=1;

                for ( i=$; i<graph.rows(); i+=groupsize() ) {
                    if ( !used[@,i]
                         && mincost[@,i]>=mincost[@,minpos[@]]
                                          +graph[minpos[@],i] ) {
                        mincost[@,i]=mincost[@,minpos[@]]
                                          +graph[minpos[@],i];
                    }
                }

                for ( i=$; i<graph.rows(); i+=groupsize() ) {
                    mpand( &finished[@], used[@,i] );
                }
            }
        }
    }
}
```

Figure 10.10: Pure Fork95++ code extract for the All-Pairs Shortest Paths problem.

```
/* create a data-type for one row of a a column block */
MPI_Type_vector( nvertices, local_width, nvertices, MPI_INT,
                 &mpi_columnblock_row_type );
MPI_Type_commit( &mpi_columnblock_row_type );

/* create a data-type for an entire column block */
mpi_columnblock_blocklengths[0] = 1;
mpi_columnblock_blocklengths[1] = 1;
mpi_columnblock_displacements[0] = 0;
mpi_columnblock_displacements[1] = local_width*sizeof(int);
mpi_columnblock_types[0] = mpi_columnblock_row_type;
mpi_columnblock_types[1] = MPI_UB;
MPI_Type_struct( 2, mpi_columnblock_blocklengths,
                 mpi_columnblock_displacements,
                 mpi_columnblock_types, &mpi_columnblock_type );
MPI_Type_commit( &mpi_columnblock_type );

/* distribute the adjacency matrix */
MPI_Scatter( &graph[0,0], 1, mpi_columnblock_type,
             &graph_columnblock[0,0], nvertices*local_width, MPI_INT,
             0, single_source_comm );
```

Figure 10.11: Pure C++/MPI code extract for redistributing a row-striped matrix as a column-striped matrix.

matrix and to represent an entire column block of the matrix. Figure 10.11 shows an extract of the code including the call to scatter the matrix over the local group.

Creating the new data type for one row is straightforward, but to create the new data type for an entire column-block requires setting the end of the block to point to the start of the next column block. This ensures that the column blocks are consecutive so that the scatter operation can be used as normal. The code extract shows that complex new MPI data types can be created, but it requires a great deal of effort by the programmer.

The **single-source** problem is solved in much the same way as the HSM/C++/MPI base level solution, with the **all-pairs** algorithm implemented by splitting the MPI_COMM_WORLD communicator into separate communicators for each local group and using the collective routines over the local communicator instead of MPI_COMM_WORLD.

At the start of the program, one processor loads in the adjacency matrix and broadcasts it to the root processors of the local groups. The groups then operate independently and calculate the shortest path weights for the required source vertex. Before completing the single-source problem the local root processor must therefore gather in the minimum cost array from the local group. On completion of all the single-source problems the global root processor gathers the minimum cost arrays from the local root processors.

The pure C++/MPI solution, listed in Appendix D.6, is long but the single-source algorithm stays quite clear. The top level `map` skeleton is, however, lost in the details of which MPI communicator is used in which MPI collective operation, and which communicator each processor belongs to. The code is also disadvantaged by restricting the number of vertices to an exact multiple of the number of processors, although the restriction could be removed by using the more complex MPI routines.

Language	pure C++/MPI
Lines	175
Operators	168
Clarity	-
Conciseness	long
Comments	High level concurrency is obscured. Restricted number of vertices.

10.7 Summary

The all-pairs shortest paths problem differs from the image thresholding and merge-sort problems by explicitly requiring a nested parallel solution. The top level parallelism is contained within a simple `map` skeleton but this is enough to give the implementor some choice over how to exploit the available parallelism. On a distributed memory architecture with relatively few processors it may be reasonable to implement the single-source calculations sequentially while on a PRAM machine up to n^2 processor may be employed using parallelism at both the all-pairs level and the single-source level. Using only the pure base languages, this choice has to be coded into the algorithm in such a way that the code performs a predicted decomposition of the problem, although this can be parameterised as it was in the pure C++/MPI solution.

The HSM model, on the other hand, allows the programmer to express both levels of parallelism but leaves it up to the HSM implementation to exploit the parallel resources of the base machine. This avoids complicating the top level solution code with the base machine parameters while allowing the implementation to

Language	Lns	Ops	Clr	Cnc	Comments
pure HSM	140	76	clear	long	Duplication of code in test and body of while skeleton.
HSM/Fork95++	68	78	clear	concise	Slight complications in use of multi-prefix operator.
HSM/C++/MPI	81	101	clear	concise	Uses two different methods to index arrays. Restricted number of vertices.
pure Fork95++	85	140	unclear	concise	High level concurrency is obscured. The use of multiple arrays to enforce scope is confusing.
pure C++/MPI	175	168	-	long	High level concurrency is obscured. Restricted number of vertices.

Table 10.1: Summary of the software metrics for the all-pairs shortest paths problem.

exploit the available parallelism.

The HSM solutions use a combination of map, reduce, zip, if and while skeletons, although only the map and reduce skeletons were integral to the algorithm. The functionality of the skeletons was implemented with reasonable ease in both the pure Fork95++ and pure C++/MPI solutions, though the nesting of the parallelism led to some complicated code which obscured the existence of the skeleton.

The pure HSM language had no problem expressing the problem, although the use of the while skeleton was not as intuitive as expected. Separating the body function away from the loop has the disadvantage of hiding the iterative nature of the code which does not happen when the loop and the body are expressed together. There is also the problem of duplicating work between the termination test function and the body function. That aside, the solution is clear and straightforward.

Using Fork95++ as a base language with the HSM code allowed a standard imperative **while**{} construct to be used but resulted in the functionality of the map and reduce skeletons having to be implemented in the base code. Catering for the difference between the number of data elements and processors is not difficult but it leads to code that is more complicated than would otherwise be necessary. The skeletons of the pure HSM solution avoid these details but incur the conceptual

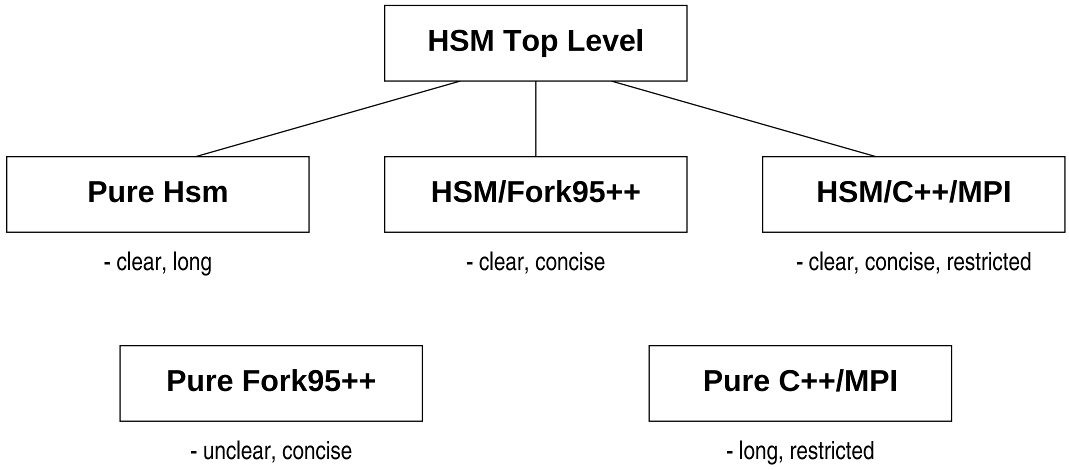

Figure 10.12: Language comparison summary for the all-pairs shortest paths problem.

cost of more data structure manipulations.

Acknowledging the restrictions of the data redistributions, the combination of the HSM code with a C++/MPI base language gives remarkably clear and concise code. The code highlights again the need for standard data redistribution routines which would reduce the work required by the base level programmer and lead to an even clearer solution.

The pure Fork95++ solution is concise but quite confusing, due mostly to the problems of sharing variables over a local group but not the whole machine. The Fork95 language does define precise semantics for the scope of shared variables but the extent of the sharing is only implicit in the program structure. Debugging the extent of the sharing is complicated and it is easier to make the distinction of the shared variables explicit through a single shared array of multiple instances, with one array element for each local group. The explicit distinction of the multiple instances makes it clear that there are multiple local groups executing concurrently though the pattern of the computation of the `map` skeleton is lost. This contrasts with the HSM/Fork95++ solution where the implicit scope of the shared variable over the local group, and only the local group, is naturally defined.

Lastly, the pure C++/MPI solution shows how nested parallelism is possible through the splitting of communicators leading to a single-source implementation similar to that of the HSM/C++/MPI solution. The code for redistributing an array with a row-block distribution to one with a column-block distribution is applicable to both solutions, and illustrates quite well the difficulties in manipulating complex distributed data structures in C++/MPI.

The effectiveness of each of the languages at implementing a solution, summarised in Table 10.1 and Figure 10.12, reinforces the ability of the HSM model to abstract out the common patterns of computation and communication inherent in the algorithm. The pure HSM solution proved relatively effective at describing the solution clearly although it highlighted difficulties in the clarity and efficiency of the `while` skeleton.

Part III

Discussion

Chapter 11

Discussion

Contents

11.1 Structured Parallelism **138**
11.2 The HSM Language **140**
11.3 The HSM Base Models **147**
11.4 An HSM Cost Model **152**
11.5 An HSM Compiler **153**
11.6 Extending the HSM Model **155**

This chapter takes a retrospective look at the HSM model and language with the results of the evaluation from Chapters 8, 9, and 10. The effectiveness of the programming model is discussed, as well the weaknesses and how the model may be improved. In particular the chapter addresses the various issues that have been raised in the preceding chapters.

Section 11.1 discusses the general use of structured parallelism and in particular the requirement for exploiting parallelism within a separate base model.

Section 11.2 concentrates on the HSM language described in Chapters 2, 3 and 4 and considers changes and extensions to the syntax, data types and skeletons.

Section 11.3 looks at the two base models described in Chapters 5 and 6 and considers two additional base models.

Section 11.4 looks at the adoption of a computational cost model and Section 11.5 looks at the potential for building HSM language compilers for the different base models.

Lastly, Section 11.6 looks at how the concepts of the HSM programming model can be extended beyond the existing HSM model.

11.1 Structured Parallelism

The top level problem formulations used in the previous three chapters each mapped neatly into instances of algorithmic skeletons, leading to clear and concise solutions. The sequential composition of skeletons was most obvious, but each of the problems also exploited the *nesting* of skeletons within one another.

11.1.1 Nested Parallelism

The nesting of the skeletons allowed the programmer to concentrate on one level of parallelism at a time, enabling an abstraction away from the parallelism details of lower levels. This high level of abstraction enhanced the clarity of the HSM superstructure solutions for each of the three problems.

For example, the clear nesting of parallelism in the HSM solutions for the all-pairs shortest paths problem contrasted with the solutions using just the pure base languages. In both the pure Fork95++ and pure C++/MPI solutions the nested parallelism was 'flattened' out, obscuring the pattern of parallelism from the upper level.

The difference was even more marked in the pure C++/MPI solutions for the image thresholding and merge-sort problems where the solutions do not exploit the full potential for parallelism in the reduce operators. The pure C++/MPI language doesn't prevent the exploitation of full parallelism, but the programming effort required is significantly greater than that required by the other languages. This is due mostly to having to implement the functionality of skeleton programming in a distributed memory environment.

11.1.2 Flexible Problem Sizes

Another advantage of the use of skeletons and nested parallelism is that the HSM code is fully independent of the number of processors available or the size of the problems. This contrasts with the restrictions of the pure C++/MPI solutions, and one HSM/C++/MPI solution. The restriction of the solutions is due to the complexities in handling irregular data distributions in the distributed memory environment. The HSM/C++/MPI solutions were not quite as restricted as the pure C++/MPI solutions, however, since the skeleton superstructure removed one level of detail.

The pure Fork95++ solution for the merge-sort problem also suffered a restriction in the problem size, requiring an equal number of processors and array elements. This could have been overcome by nesting a sequential level into the solution, but at the cost of greater complexity and even less clarity. The HSM/Fork95++ solution, on the other hand, had no problems in handling arbitrary problem sizes.

11.1.3 Flexible Machine Sizes

By abstracting away the size of the machine, the HSM solutions manage to be largely independent of the number of processors. The one exception was the HSM/C++/MPI solution for the image thresholding problem which required that the size of each node performing a reduce operation doubled at each level, i.e. the sizes are powers of 2. The restriction is brought about by trying to simplify the redistribution of the HSM data structures. This is unfortunate but, as with the C++/MPI restrictions already described, flexible solutions are possible.

Another advantage of the independence of machine size was highlighted for the all-pairs shortest paths problem. The solution can exploit parallelism on n^2 processors on a problem of size n, so when there are fewer than n^2 processors some parallelism cannot be exploited. In the HSM model the skeleton compiler can decide which skeletons to parallelise, and which to implement sequentially. The programmer expresses all the available parallelism but leaves the choice up to the compiler. This differs from the pure Fork95++ and C++/MPI languages where the programmer, rather than the compiler, queries the machine size before deciding on which parallelism to exploit. The code required to perform this querying will decrease the clarity of the pure language solutions, as illustrated in the proliferation of **for**{} loops in the Fork95++ code.

11.1.4 Use of Ad Hoc Parallelism

The final issue to arise from the previous three chapters was the use of the ad hoc parallelism in the base model. Solutions using purely the HSM language were all possible, so the ad hoc parallelism was not found to be essential, but the skeleton formulations were not always the clearest or most intuitive ones.

The merge-sort problem in particular highlighted the difficulty of expressing a SIMD algorithm using a skeleton formulation. The most obvious solution to this problem is to add a SIMD skeleton to the language, but this simply encompasses a separate base model in a different manner to that used by the HSM/Fork95++ interface. The alternative solution discussed at the end of Chapter 9 is to add a permute skeleton which will be discussed in Section 11.2.3.

The image thresholding problem also had a slight problem in using a skeleton formulation to select the appropriate thresholding value. The use of the filter skeleton was successful, but only because it removed the dependency on the use of neighbouring values. The obvious solution is therefore to add the map-with-halo geometric style skeleton to the language which was described at the end of Chapter 4.

In both these cases, the solution was to propose additional skeletons to the language. While the above skeletons have been investigated by other researchers, it remains an open question as to whether a small finite set of skeletons exists which can be used to express all the parallelism in any problem. The use of a separate base model in which ad hoc parallelism can be exploited, such as the SIMD code of the merge-sort problem, avoids this issue by allowing the programmer to express any parallelism that can be expressed in the chosen base language. This allows a programmer to use the structured parallelism where suitable skeletons exist, and alternative means where there are no suitable skeletons.

11.2 The HSM Language

The HSM language defined in Chapters 2, 3 and 4 is one language for the HSM model, but other languages are also possible. In particular an imperative style was chosen for the language, though other skeleton languages exist based on alternative styles as introduced in Chapter 1. This section looks at the syntax and data types of the HSM language presented in this thesis and considers various changes and extensions.

11.2.1 Syntax

The HSM language was able to express solutions to all the problems. The code was not always concise, however, due mostly to the need for a range of attendant functions. Most of these functions are of the order of three or four operators long, and perform assignment to compound structures or simple structure arithmetic operations.

Since functions can only be passed to the skeletons by name, this requires separate functions for each of the attendant tasks. One example from the pure HSM solution to the merge-sort problem concatenates two arrays together:

```
return ( a++b );
```

The separate function leads to a longer program listing, and decreases slightly the readability of the code by separating the calculation from its use in the reduce skeleton.

The calculation is simply an instance of an operator, so one improvement in the language would be to 'define' default function names for each of the operators. The SKIL language, for example, allows the conversion of operators to functions by passing them as arguments between brackets. The HSM language could be extended in a similar manner by quoting the operator, e.g.:

```
reduce( '++' , NULL, array_list );
```

Anonymous Functions. The HSM language could also be extended to allow the use of *anonymous* functions in the language. For example, the pure HSM solution to the image thresholding problem requires the comparison of an element of the histogram with the threshold count:

```
return ( element.c >= threshold_count );
```

This requires referencing one element of the compound structure and comparing it with the integer count, so it is not simply an instance of an inequality operator. The filter skeleton could use an anonymous function defining the same calculation, e.g.:

```
filter( (int tc, histogram_element_t e){return(e.c>=tc);},
        threshold_count, cumulative_histogram);
```

This is not particularly clear, but it removes the need for a separate function and keeps the calculation together with the skeleton. It also facilitates the use of expressions and sequences of statements, for example to negate the result of a boolean test or to perform several assignments in the body function of a skeleton. This would be particularly useful for the if skeleton as described in Section 11.2.3.

Partial Function Applications. If the HSM language was moved to a functional style, for example based on ML, it would also be possible to define partial function applications. This would be a particularly interesting change as there would no longer be the need to separate the global and local arguments to the body functions. Instead, the body function could be partially defined by already including the global component. The syntax for this, however, would be highly dependent on the rest of the functional syntax for the new language.

11.2.2 Data Types

The data types available in the HSM language were also sufficient to express solutions to each of the problems. The ability to nest the data types was particularly useful, for example in the image thresholding problem to declare an array of histograms, where each histogram is an array of compound structures containing integers.

The following paragraphs discuss some extensions to the data types and operations which would make the language more usable, but were not required by the problems.

Multi-Dimensional Arrays. One slight confusion was inherent in the use of the multi-dimensional array type as it is not immediately obvious how many dimensions an array has. The type of an array does not specify the size or the number of dimensions, so that an array of arrays can contain one- and two-dimensional member arrays with different sizes.

The confusion between one- and two-dimensional arrays was particularly noticeable in the all-pairs shortest paths problem when dealing with single-rowed two-dimensional arrays. This is inherent in the use of the multi-dimensional array type, so it is left to the programmer to use meaningful names to improve the clarity of the algorithm.

Partial Array Indexing. The semantics of the language defined in Chapter 3 require that when dereferencing an array there is exactly one range or index operator for each dimension. This raises the question of what would happen if *partial* array dereferences were permitted.

The language could be extended to permit partial array dereferences by implicitly replacing each 'missing' operator with a range operator spanning the full width of that dimension. For example, the following two sub-array assignment operations would be identical:

```
array< int > a;
a = array_create(2, 10, 10);
a[1][0..array_size(a)[0]−1] = a[0][0..array_size(a)[0]−1];
a[1] = a[0];
```

This would not require a change in the grammar, merely the semantics, and would simplify code such as the pure HSM solution of the all-pairs shortest paths problem.

Since the [0..array_size(a)[0]−1] expression is commonly used to span the width of one dimension, it may also be reasonable to provide a shorthand representation by allowing 'missing' start and end points to be implicitly replaced, e.g.:

```
a[..][1] = a[..][0];
```

Array Ranges and Concatenation. As described in Chapter 3, the array concatenation operator can only concatenate arrays in the top most dimension. It therefore seems reasonable to extend the language to permit concatenation in any dimension. This could be done by passing a single integer argument to the concatenation operator as is done for transforming arrays, e.g.:

```
array< int > b;
b = a[..][0] ++<1> a[..][9] ;
```

It should even be possible to extend the language to allow 'implicit' concatenation on the left hand side of an equation, e.g.:

```
b = create( 2, 10, 2 );
a[..][0] ++<1> a[..][9] = b;
```

Languages such as High Performance Fortran (HPF) provide powerful operations for selecting specific elements from an array. HPF's flexibility in manipulating arrays has proved a significant benefit to HPF programmers, and there should be few problems in extending the HSM language to include similar functionality by extending the range operator. The only extra issue of concern is that of the resulting data distributions on distributed memory architectures, but these issues are already addressed by the sub-array assignment and concatenation described in Chapter 6. The implementation will perform the minimum data redistribution necessary to undertake the array assignments, and the default distribution can be enforced by creating the array before assigning anything to it.

Array Splitting The divide-and-conquer paradigm used by the merge-sort problem also highlighted the need to be able to split an array into contiguous chunks. This was done in the HSM superstructure by creating a one-dimensional array of arrays with two elements and then assigning each element sequentially.

Dividing up an array into n blocks is a relatively common operation, so it seems reasonable to extend the language to add this functionality. This could be done by adding the following function to the HSM language:

```
array< array< type-α > > array_split (
        int nblocks,   array< type-α > local
);
```

The function would take an integer expression and an array, with the integer expression specifying the number of blocks that the array should be broken into.

Array Casting. The use of array range and index operators allows a sub-array to be expressed with fewer dimensions, as is also possible using the array transformation/cast operator, but the language doesn't offer any facilities to *increase* the number of dimensions, for example to convert a one-dimensional array into a single-rowed two-dimensional array. This would have been particularly useful for the HSM superstructure to the all-pairs shortest paths problem. This could be done by extending the array transformation/cast operator using a 'null' placement, e.g.:

```
array< int> c;
c = array_create( 1, 10 );
a = (array<*,1>)c;
```

This would create a two-dimensional array with the first dimension of size one.

11.2.3 Skeletons

As already mentioned in Section 11.2.1, some limitations to the skeletons were found and several further skeletons could be added to the language. The main limitations were in the if and while skeletons which sometimes decreased the clarity of the code.

Other problems were due mostly to the lack of a suitable skeleton such as the map-with-halo skeleton for the image thresholding problem, and a permute skeleton for the merge-sort problem. The absence of this latter facility was particularly noticeable as it led to long code which lacked clarity.

The following paragraphs discuss some of the issues involved.

The if Skeleton. The if skeleton was put to good use in the pure HSM solution to the all-pairs shortest paths problem when finding the minimum vertex:

```
return( if( b.used, a, if( a.used, b, if(a.cost<=b.cost, a, b ) ) ) );
```

The code is clear and concise, and benefits from the use of expressions instead of function names. The use of expressions has the disadvantage, however, in that only one of the last two expressions in each if skeleton should be evaluated. When expressions are used as the arguments to any other function or skeleton the expressions are always be evaluated, so the if skeleton proves to be an anomaly in this respect. A more accurate if skeleton would use the same style of function name with global and local arguments[1] which appear in the other skeletons, e.g.:

```
type-β  divide (
        fnname test_fn,   fnname then_fn,   fnname else_fn
 );
 bool test_fn ( type-γ  global,   type-α  local  );
type-β  then_fn ( type-γ  global,   type-α  local  );
type-β  else_fn ( type-γ  global,   type-α  local  );
```

This is significantly more complicated, however, and would lead to particularly long code listings even with the use of anonymous functions.

Another alternative would be to recognise that the if skeleton is not an algorithmic skeleton and to replace it with a typical if-then-else construct in the SISD language of the HSM.

The while Skeleton. Another problem using the extended if skeleton outlined above is in the duplication of calculations which already happen to the while skeleton. In the pure HSM solution to the all-pairs shortest paths problem both the test function and the body function of the while skeleton perform the same reduce calculation. It would therefore be more efficient if the reduction was performed once for the test function, and the results retained for use in the body function.

[1]Only one argument is needed, but it's better to conform with the other skeleton body functions.

It may be possible to get the HSM compiler to identify this pattern and optimise it, but the programmer must still explicitly write the duplicate instances.

A related problem is that the test and body functions are integral parts of the while loop, and forcing them to be separated may decrease the clarity of the construct. This would be avoided by replacing the while skeleton with a typical while construct in the SISD language of the HSM, as was suggested for the if-then-else construct.

The permute Skeleton. The need for a permute-type skeleton was made particularly clear by the pure HSM solution to the merge-sort problem. Without the permute facility the HSM code was significantly more complicated and much less clear. The permute skeleton differs from the other algorithmic skeletons presented in this thesis as it performs no computation, but the skeleton does encapsulate an irregular communication pattern over the array.

The skeleton effectively facilitates a limited form of SIMD computing in a shared memory environment. Because the communication is restricted, the skeleton can be implemented on a distributed memory architecture. The efficiency of this operation is unclear, so an accurate cost model will be needed to compare different algorithms as will be described in Section 11.4.

11.2.4 Efficiency

All the algorithmic skeletons described in Chapter 4 have natural implementations which utilise the hierarchical partitioning of the underlying machine. The ability to map computations efficiently onto a range of parallel architectures is, in fact, one of the design criteria of algorithmic skeletons. The efficiency of the implementations discussed in this thesis therefore comes down to the overhead of partitioning and de-partitioning the processors and memory resources.

In a shared memory machine no partitioning of the memory is required, and the partitioning of the processors into groups is a simple function on the processor index. The overhead in shared memory machines is therefore minimal.

In a distributed memory machine the processors can be partitioned with a similar function on the processor index, but partitioning the memory can be a potentially expensive operation. At worst, this will involve n^2 communications to send a block of data from each processor to every other processor. The default hierarchical distribution of the datastrutures tries to reduce the number of redistributions required during the execution of a program, but the ability to have ad-hoc data distributions means that it is not possible to eliminate all data redistribution. It is therefore up to the programmer or the compiler to minimise the number of redistributions. Section 11.5 discusses some of the optimisations that the compiler can perform.

11.3 The HSM Base Models

The HSM solutions described in the previous chapters showed how the HSM model integrated successfully with both the shared and distributed memory base models, whose interfaces were defined in Chapters 5 and 6. The common use of a SPMD leaf node, or more accurately a *single-function-multiple-data* leaf node, was particularly natural to the problems and fitted in well with the hierarchy of nodes created by the skeleton superstructure.

The use of the skeleton superstructure allowed the base model programmer to concentrate on smaller problems and therefore made it easier to exploit the full potential for parallelism. This was particularly noticeable in the HSM/C++/MPI solutions to both the image thresholding and merge-sort problems, especially when compared with the pure C++/MPI solutions that were developed.

The biggest difficulties encountered in the base model were due mainly to manipulating the distributions of the parallel HSM data structures. The use of a single encompassing parallel structure instead of separate distributed structures made the solutions similar for the shared and distributed memory targets, but also led to a false level of expectation to the distributed memory programmer as will be described below.

The following sections discuss the issues for both the base models, and also look at the possibilities of using further base models.

11.3.1 The Shared Memory Base Model

The combination of the HSM superstructure with the Fork95++ base language and HSM/Fork95++ interface was the most successful of the three HSM environments used in the previous three chapters. The HSM/Fork95++ combination generated clear and concise solutions to all the problems, unlike the pure HSM language which was sometimes unable to express the solution clearly, and unlike the pure Fork95++ language which obscured the upper levels of parallelism in the problem.

The use of the parallel HSM data structures was particularly intuitive and abstracted out the complexities of the extent of scope of shared variables. The difference to the pure Fork95++ language was particularly marked for the all-pairs shortest paths problem where the scope of the shared variables in the pure Fork95++ language was unclear.

11.3.2 The Distributed Memory Base Model

The distributed memory base model also benefited from the use of parallel data structures instead of distributed data structures by encapsulating the distribution information in a common class. This allowed the base model programmer to use the is_local() and where_is() operations to determine communication patterns.

Array Indexing. The use of the first_local(), next_local(), and last_local() operations was also invaluable to iterate through all the locally stored elements. The one side effect of these operations is that the functions return HSM_index structures. This leads to the algorithm using a combination of direct array access using integers and element access using the index type:

```
int i;
HSM_index ind;
HSM_array< HSM_int > a;
ind = a.first_local();
i = ind[0];
assert( a(i)==a.element(ind) );
```

For one-dimensional arrays in particular, this leads to slight confusion in the code. It would be preferable to adopt a single reference function, but overloading it to accept either integers or an index would also be confusing. The confusion is unfortunately unavoidable when permitting arrays of arbitrary numbers of dimensions.

Data Structure Distributions. The greatest difficulty encountered in the various HSM/C++/MPI solutions was the redistribution of the data contained in the parallel HSM structures to other processors. The redistribution is a necessary part of the solutions, but the details required to calculate what and where to send are complicated and prevent the programmer from implementing fully flexible solutions. This was particularly noticeable in the image thresholding problem, which only needed the default standard distributions of arrays over the whole node. Unfortunately the function is passed the default distributions over each half of the node, so the base model programmer is required to redistribute the arrays from one standard distribution to another.

The all-pairs shortest paths problem also requires the redistribution of an array with a standard distribution, but the required distribution of the resulting data structure uses a column-block format instead of the standard row-block format. It should be possible to add this distribution strategy to the HSM/C++/MPI interface, as well as any other common strategies such as column and row striping.

The redistribution required by the HSM/C++/MPI solution to the merge-sort problem is quite different in that the resulting distribution is irregular and data dependent. It is therefore necessary to be able to manipulate ad hoc distributions to match the ad hoc nature of the partitioning of the input structures. This was easily possible using the create(AD_HOC) and add_local() functions described in Chapter 6.

There are two approaches to redistributing data structures which would have required less effort from the base model programmer. The first would be to specify a *required* distribution of the input structures. This specification could be included in the top level HSM code, but the HSM code is supposed to be independent of the underlying model. The advantage of this approach is that the compiler could perform some optimisations to make sure that the data is kept in the required distribution for as long as is possible.

The second approach would be to provide data redistribution routines for the HSM data structures. This is in keeping with the concept of the parallel HSM structures, but the current routines require no synchronisation or communication between the member processors. The only link between the HSM interface and the MPI interface is the need for the HSM_COMM_WORLD communication scope and mpi_type() member function used in the ad hoc communications. Adding data redistribution functions to the HSM classes is possible, but would require each processor to call the same routine at the same time, i.e. the routines are collective. The function prototype would be:

> *HSM-type-α* redistribute (
>   *HSM-type-α* old, **int** distribution
>);

A further collective routine to add to the class would be a synchronisation routine for structures with ad hoc distributions. After synchronisation the processors would be able to use the is_local() and where_is() routines as can already be used for standard distributions.

void sync (**void**);

MPI already has collective routines which were used extensively in all three of the HSM/C++/MPI solutions, but the availability of the collective redistribution routines would extend the partial-shared memory environment feel of the base model. If a collective redistribution routine was added, the next step would be to provide collective read and write functions on the structures. Again, this is possible but it is changing the C++/MPI base model into a pseudo-shared memory model rather than simply augmenting the base model with the HSM/C++/MPI interface.

Location Information. One problem with the common ad hoc distribution scheme described in Chapter 6 is that it requires each processor to know the location of every element. The rigid distribution scheme avoided this problem by making the calculation deterministic, but the information is required if ad hoc distributions are to be allowed.

The one solution would be to include a form of compression in how the location information is stored in the parallel structure. For example, ranges of elements stored on the same processor can be specified by their endpoints instead of listing each element individually. The advantages of including the information were clearly demonstrated in the HSM/C++/MPI solutions to the image thresholding and merge-sort problems, however, so the cost is considered worth paying.

Overlapping Distributions. As mentioned above, the standard row-block distribution described in Chapter 6 is only one of a family of standard distributions, which also includes the column-block distribution. A particularly interesting distribution strategy would be an overlapping distribution designed to facilitate the map-with-halo skeleton. This would replicate selected elements on several processors depending on where the halos overlapped. Since the HSM structures passed as arguments cannot be modified, keeping the structure coherent over the node is possible by synchronising the overlaps before each instance of the map-with-halo skeleton.

11.3.3 A BSP Base Model

An alternative base model to the Fork95++ and C++/MPI base models would be the Bulk Synchronous Parallel (BSP) model introduced in Chapter 1. The BSP model also uses a SPMD style which can be fitted into the SFMD style used in the leaf nodes of the hierarchy.

The BSP model doesn't allow the machine to be partitioned into smaller machines, but this only affects the building of a suitable HSM/BSP compiler. The HSM/BSP interface would not be affected by the lack of locality.

Since the BSP model uses a two-level memory hierarchy with local memory unique to each processor, the interface and parallel HSM data structures will be similar to the HSM/C++/MPI interface. The only differences will be in the omission of the HSM_COMM_WORLD and the mpi_type() member function. The collective redistribution routines described above can also be implemented in the BSP model, so no change to the interface is required.

11.3.4 A Threaded Base Model

Another possible base model could be a threaded base model, using a library of routines such as POSIX-Threads to create, synchronise and destroy lightweight threads.

Assuming the machine consists of a fixed number of loosely synchronised threads, the hierarchy would partition in a similar manner to the shared memory/PRAM machine. The leaf nodes would therefore contain a number of threads which could communicate using shared memory and synchronise using the thread synchronisation primitives. This would require an HSM interface and parallel data structures similar to those of the HSM/C++/MPI interface, as well as a threaded equivalent to the HSM_COMM_WORLD communicator.

An alternative model could be implemented using a single thread for each node, especially as the HSM code is SISD, i.e. sequential. To 'partition' the node the required number of new threads could be created for each body function of an algorithmic skeleton. The base model of this latter hierarchy would be a sequential function, requiring no synchronisation or communication with other threads.

One issue that this second model raises is that the HSM **node_size()** function will always return a size of one thread. If the programmer uses this value to decide when to stop creating more parallelism, for example in the HSM/C++/MPI solution to the merge-sort problem, this will prevent *any* parallelism from being exploited in the divide skeleton.

11.4 An HSM Cost Model

The HSM programming model described in this thesis does not define a cost model:

> *Augmenting programming models with performance metrics is not as simple and accurate as in a computational model. Combining performance analysis with programming models seems a messy task with inherent tradeoffs between accuracy and architecture independence. Ideally, a programming model should stick to its role ... and rely on mappings down to an underlying computational model which has the responsibility for performance analysis.* ([HL93])

An existing model of computation therefore needs to be adopted and a mapping should be created from the algorithmic skeletons down to the computational cost model.

Although this thesis does not describe a cost model, the HSM programming model is designed to enable the adoption of a suitable computational cost model. In particular, the composition of the skeletons should map cleanly to the composition of function costs, and the nesting of skeletons should be calculable from nested function costs.

Creating cost functions for each algorithmic skeleton is not a simple matter, as the performance will rely heavily on:

(a) the performance characteristics of the underlying machine

(b) the algorithms chosen to implement the skeleton

One suitable cost model would be the CLUMPS computational cost model[Cam94]. The CLUMPS cost model encapsulates the performance of the underlying machine and is particularly suited to architectures which can be recursively partitioned.

Alternatively, the BSP cost model could be adopted[McC92]. The BSP cost model does not permit the use of localised communication/synchronisation costs, nor does it permit the partitioning of the machine, but only the BSP language prevents sub-group synchronisation. The BSP cost model could therefore be adopted using the sub-group synchronisation of the HSM language, and the global synchronisation and bandwidth parameters of the whole machine. This does require, however, that the global synchronisation costs are at least as great as the sub-group synchronisation costs.

With either cost model, it then becomes necessary to calculate the cost functions for each of the skeletons. The cost of the skeletons will be dependent on the implementation of the skeleton, and also on the costs of the body functions. These costs may not always be regular, which will lead to more complex cost analysis of HSM code.

A useful technique for profiling the cost of algorithmic skeletons would be to adapt the work by Howell in predicting parallel performance[How96]. Howell uses a technique of *reverse profiling* of MPI libraries which allows the execution of a parallel program on one architecture using the cost functions for MPI routines on another architecture. A similar technique could be used by crafting algorithm-specific cost functions for each implementation of a skeleton, and simulating the execution. For example, there could be one cost function for implementing the `map` skeleton using a task farm, and another cost function for using a static partition of the work. Both of these will be mapped onto the underlying cost model, such as the BSP cost model or the MPI model developed by Howell.

11.5 An HSM Compiler

During the course of research towards this thesis a parser was built for the HSM language using the grammar described in Appendix A. This parser was used to check the syntax of the HSM solutions given in the previous three chapters.

Although the parser exists and various work was done in implementing several of the skeletons in both Fork95 and C++/MPI, a compiler for the HSM language does not exist. Building a compiler would involve the following work:

(a) implementing the nested data structures and operations described in Chapter 3.

(b) implementing the polymorphic skeletons described in Chapter 4.

The existing skeleton implementations only use 'flat' arrays of integers, and extending the implementation to nested data structures will prove a challenge. This is particularly true for the shared memory target architecture as the Fork95++ language does not exist yet! For the distributed memory target architecture the issue of concern is the difficulty in calculating MPI data types for the arbitrary HSM data structures. Given enough time, however, an MPI implementation would be possible.

An HSM/BSP compiler could be written, but since the existing BSP languages don't support sub-group synchronisation only the Bulk Synchronous Message Passing routines would be available. This gives little advantage over the MPI implementation, though it is possible that some form of machine partitioning may appear in the BSP languages of the future.

An HSM/Threads compiler would be much more interesting, and should also be easier as it avoids the difficulties of passing messages containing the nested data structure elements. There are, however, still issues to be resolved as to the exact HSM/Threads interface.

11.5.1 Compiler Design

The HSM compiler must deal with both the top level skeleton code and the base level code. The design adopted by the preliminary work took the form of a preprocessor for the skeleton code which generated code in the base level language. The output would then be compiled using the base language compiler, as would the base language code. The compiled outputs of both tiers would then be linked together to form the final executable.

11.5.2 Compiler Optimisations

The ad-hoc nature of the base language makes it difficult to perform parallel optimisations to the base code, but there is significant scope for optimisation within the top level skeleton code. The elimination of redundant variables would offer particular improvements to both space and time usage.

The distribution of the parallel data structures may also be optimised by identifying the data distribution required ahead in time and minimising the need to re-distribute data. The structured nature of the hierarchical skeletons also provides potential optimisations through program transformation from one hierarchy to another while keeping the semantics the same.

The performance of the base level code can be optimised by ensuring that the HSM data structures passed down are already in the required data distribution.

Obtaining this information is complicated, however, by the fact that the HSM compiler looks only at the HSM source code when compiling. One way to supply this information to the HSM compiler would be to specify the required data distribution in the top level code at the same time as the base function prototypes are given. This is not clean as it puts architecture specific information in the architecture independent top level code. The alternative is to specify the required distribution in the base level code, in the form of compiler directives, and to process the base files as part of the compilation of the skeleton code. This would also enable the compiler to check consistency between the HSM prototype of the base function and the base language prototype.

11.6 Extending the HSM Model

The programming model described in this thesis uses the HSM language to express the skeleton superstructure of a problem and uses separate base languages with an HSM interface to program the leaf nodes of the superstructure hierarchy.

The SPMD style of the HSM/C++/MPI solutions give the base level code the look and feel of a structured partial-shared memory environment. The addition of the data distribution operations discussed in Section 11.3.2 would reinforce this feel, and could be extended further. For example, the HSM skeleton operations could be provided as member functions of the HSM_array<> class. This would obviate the need for learning a new language or for building a new compiler, but the appearance of structure in the base level code is only superficial and is not enforced by the language.

This raises a second, related, issue concerning the use of structured parallelism *within* an ad hoc hierarchy, as opposed to ad hoc parallelism within a structured hierarchy. If the algorithmic skeletons are simply member functions of the HSM_array<> class, it would be possible for the processors to perform structured HSM operations from within an unstructured and ad hoc framework. This would clearly provide greater flexibility to the programmer, but the lack of a structured framework would make it more difficult to follow and reason about the behaviour of an algorithm. The purpose of the HSM model is to improve the descriptive simplicity of parallel programs by separating the structured parallelism from the unstructured parallelism. Replacing the HSM base language interface with a self-contained class library of parallel data structures and operations would lead to a powerful base model but at the cost of losing the clear superstructure of structured skeletons.

Since the program would be written in the language of the base model, the above approach would also make it more difficult to translate the program to a different base model. Using the HSM model, only the base model leaves need to be translated from one base model to another.

The previous three chapters have shown how the HSM model is suitable for tackling the three problems, and the discussions in this chapter have suggested various improvements to the HSM language which would make the model even more flexible. For this reason the adoption of the separate HSM language is the preferred option.

Chapter 12

Conclusions

Contents

12.1 Sequential and Parallel Base Languages 157
12.2 The HSM Language 159
12.3 The HSM Base Model Interface 159
12.4 Further Work 160
12.5 The Future of Parallel Computing? 160
12.6 Summary 161

This thesis has proposed a new model of programming for general purpose parallel computing which is designed to produce clear, portable, and efficient code.

The Hierarchical Skeleton Model builds on the use of algorithmic skeletons to encapsulate recurring patterns of parallel computation and interaction in parallel algorithms. The skeletons in the HSM language give the programmer a high level of abstraction which avoids the complex details of inter-processor communications needed to implement the patterns of interaction and computation. The skeletons are parameterised by parallel functions which can also be expressed using the skeletons, or can be expressed using the ad hoc parallel facilities of the base language. This generates a dynamic hierarchy of parallel functions with an instance of an algorithmic skeleton at each node, and leaf nodes containing potential ad hoc parallelism.

12.1 Sequential and Parallel Base Languages

Existing models of structured parallel programming have required the programmer to express all the parallelism of a problem in terms of algorithmic skeletons, so that the leaf nodes of the hierarchy are sequential calculations. A special skeleton language is used to express the upper layers of the hierarchy, i.e. the skeleton superstructure, and the leaf node calculations are expressed using an established sequential base language such as C++ or Fortran. The skeleton superstructure interfaces with the sequential base languages by passing standard sequential data structures which will be updated by the sequential function.

Programs written using a skeleton language are potentially clearer and more concise than their non-structured language counterparts, though the programmer must be able to express all the parallelism of the algorithm using the available skeletons. If suitable skeletons do not exist a solution may still be possible but may not be as clear or concise as the original algorithm. These languages must therefore have an exhaustive set of skeletons for all potential patterns of computation and interaction.

The HSM model also defines a special skeleton language, the HSM language, but the parallelism is not restricted to the skeleton superstructure. The base language may therefore be parallel, such as C++ with the MPI message passing library for a distributed memory base model, or Fork95 for a shared memory base model. The superstructure interfaces with the base language by passing *parallel* data structures. The base language is augmented with the appropriate HSM interface which provides the new parallel data types. The parallel types allow individual processors to manipulate elements of the structures independently from the other processors, though the nature of the memory of the base model may restrict access to the data structure elements. The processors execute in a *single-function-multiple-data* style, accepting the parallel data structures as arguments to the common function and returning a single parallel data structure. During execution of the function the processors may communicate with one another using the interaction facilities of the base language.

By allowing parallelism within both the skeleton hierarchy and the leaf nodes the programmer has the choice of expressing the patterns of computation and interaction using either the skeletons or the facilities of the parallel base language. This enables the programmer to use the structured skeletons where appropriate but also provides for the greater flexibility possible in the ad hoc use of the base language facilities.

12.2 The HSM Language

The HSM language proved to be sufficient for expressing the skeleton superstructures of the three problems tackled, generating clear and concise solutions. The HSM language was not always ideal, however, for expressing the entire solutions without the use of a separate base language. In particular, the language lacked map-with-halo and permute skeletons which would have enabled clearer solutions. Using only the available skeletons to express the solution was possible but led to complex and unclear solutions. The combination of the HSM language with the separate parallel base language was therefore the preferred choice for two of the three problems.

Although the skeleton superstructures were expressed concisely using the HSM language, the language could be improved to enable even shorter solutions. In particular, the inclusion of anonymous functions and partial function applications might improve the simplicity of the HSM code even further. This would be difficult to do with the existing imperative HSM language, but the HSM model does not require the language to be imperative. A functional HSM language would be more suited to the use of second- and higher-order functions in the skeleton superstructure.

12.3 The HSM Base Model Interface

The adoption of a *single-function-multiple-data* model for the base model fitted naturally into the partitioning hierarchy of the skeleton superstructure. The use of parallel data structures to augment the base languages also made the interface between the superstructure and the base models explicit, generating clear solutions for the base language functions. The solutions made frequent use of the collective information of the parallel structures, which would have led to longer and less clear solutions had distributed sequential structures been used instead of the parallel structures.

The solutions for the distributed memory base model were significantly longer than those for the shared memory base model for all three problems. This was due primarily to the need to redistribute the parallel data structures within the base functions. The ability to perform non-standard distributions was necessary, but the HSM/C++/MPI interface would benefit greatly by the inclusion of standard data redistribution functions.

12.4 Further Work

A natural next step for the HSM model would be to develop a cost model for the language. This will require the adoption of a suitable computational cost model, such as CLUMPS or BSP, and the creation of a mapping from the algorithmic skeletons to the underlying costs. The composition and nesting of skeletons should map cleanly to the composition and nesting of cost functions.

An HSM compiler should also be developed, initially for the C++/MPI target environment and then for a shared memory environment. A threaded target environment would be particularly interesting. The performance of programs written using the HSM programming model could then be compared against the performance predicted by the cost model. The use of reverse profiling techniques could be used to experiment with different architectures and skeleton implementations.

12.5 The Future of Parallel Computing?

Parallel computing is spreading beyond the massively parallel machines used for large physics and meteorological simulations, and is already appearing on some 'desktop' machines in the form of symmetric multi-processor workstations. Since they offer a single shared memory, there is less need for explicitly distributed message passing environments such as MPI. Thread-based programming offers a more appropriate environment especially with the standard POSIX thread library and the inclusion of threads in emerging languages such as Java.

Controlling multiple threads is still a complicated task, however, and there will need to be some way to simplify the task. Algorithmic skeletons offer a suitable high level programming model for threaded programming, and therefore have a significant contribution to make to general purpose parallel programming. The HSM model presented in this thesis which accepts ad hoc parallelism in the leaf nodes should also provide a suitable programming model, though other models are likely to emerge. In particular, skeletons may appear in the form of library functions and in the longer term may become embodied in standard programming languages such as the C++ STL algorithms.

12.6 Summary

The HSM programming model has been used to generate clear and concise code which takes advantage of the ad hoc parallel interaction facilities of the base languages. The high level of abstraction in the HSM language allows the programmer to concentrate on the nature of the parallelism rather than the details of interaction which are required by the lower level base languages. This leads to greater simplicity in the description of the algorithm, as well as greater programmability. The decomposition of a parallel algorithm into skeleton instances also follows naturally from existing development methodologies. Lastly, solutions are portable over architectures supporting the same base model, and only minimal translation is required to translate the base language functions programs to alternative base models.

Appendix A

The HSM Language Grammar

This appendix gives the full grammar of the HSM language, as described in Chapters 2, 3 and 4.

id:

```
[a-zA-Z][a-zA-Z0-9_]*
```

id-list:

id | *id* ‘`,`’ *id-list*

type:

‘`void`’
| ‘`bool`’
| ‘`int`’
| ‘`real`’
| ‘`string`’
| ‘`fnname`’
| ‘`array`’ ‘<’ *type* ‘>’
| ‘`struct`’ ‘{’ *var-declarations* ‘}’

type-definition:

‘`typedef`’ *type id* ‘;’

type-definitions:

/ nothing */*
| *type-definition type-definitions*

var-declaration:

type id-list ‘;’

var-declarations:

/ nothing */*

| *var-declaration var-declarations*

int-const:

`[0-9][0-9]*`

real-const:

int-const '`.`' *int-const*

expr:

variable

| *expr ref-operator*

| *id* '`(`' '`)`'

| *id* '`(`' *expr-list* '`)`'

| '`(`' *expr* '`)`'

| '`TRUE`'

| '`FALSE`'

| *int-const*

| *real-const*

| '`(`' '`int`' '`)`' *real-expr*

| '`(`' '`real`' '`)`' *int-expr*

| '`(`' '`array<`' *int-expr-list* '`>`' '`)`' *array-expr*

| '`!`' *bool-expr*

| *bool-expr* '`||`' *bool-expr*

| *bool-expr* '`&&`' *bool-expr*

| *expr* '$==$' *expr*

| *expr* '$!=$' *expr*

| *num-expr* '$<$' *num-expr*

| *num-expr* '$<=$' *num-expr*

| *num-expr* '$>$' *num-expr*

| *num-expr* '$>=$' *num-expr*

| *num-expr* '`+`' *num-expr*

| *num-expr* '`-`' *num-expr*

| *num-expr* '`*`' *num-expr*

| *num-expr* '`/`' *num-expr*

| *array-expr* '`++`'*array-expr*

| '`array_create(`' *int-expr* '`,`' *int-expr-list* '`)`'

| '`array_create_init(`' *int-expr* '`,`' *int-expr-list* '`,`' *expr* '`)`'

| ‘`array_create_indexed(`’ *int-expr* ‘`,`’ *int-expr-list* ‘`)`’
| ‘`vector_create_indexed(`’ *int-expr* ‘`)`’

expr-list:
expr
| *expr* ‘`,`’ *expr-list*

arg-list:
type id
| *type id* ‘`,`’ *arg-list*

ref-operator:
‘`[`’ *int-expr* ‘`]`’
| ‘`[`’ *int-expr* ‘`..`’ *int-expr* ‘`]`’
| ‘`.`’ *id*

variable:
id
| *variable ref-operator*

base-function-prototype:
‘`basefn`’ *type id* ‘`(`’ ‘`void`’ ‘`)`’ ‘`;`’
| ‘`basefn`’ *type id* ‘`(`’ *arg-list* ‘`)`’ ‘`;`’

base-function-prototypes:
/ nothing */*
| *base-function-prototype base-function-prototypes*

function-prototype:
type id ‘`(`’ ‘`void`’ ‘`)`’ ‘’ ‘`;`’
| *type id* ‘`(`’ *arg-list* ‘`)`’ ‘`;`’

function-prototypes:
/ nothing */*
| *function-prototype function-prototypes*

statement:
variable ‘`=`’ *expr* ‘`;`’ | ‘`node_size`’ ‘`(`’ ‘`)`’ | ‘`return`’ ‘`(`’ *expr* ‘`)`’

statement-list:
statement ‘`;`’
| *statement* ‘`;`’ *statement-list*

function:

type id '(' 'void' ')' '{' *var-declarations statement-list* '}'

| *type id* '(' *arg-list* ')' '{' *var-declarations statement-list* '}'

functions:

function

| *function functions*

hsm-program:

type-definitions base-function-prototypes function-prototypes functions

Appendix B

The Image Thresholding Code

Contents

B.1 The HSM Skeleton Superstructure 166
B.2 The Pure HSM Solution 169
B.3 The HSM/Fork95++ Solution 170
B.4 The HSM/C++/MPI Solution 173
B.5 The Pure Fork95++ Solution 178
B.6 The Pure C++/MPI Solution 181

This appendix gives the full code listings of the image thresholding solutions described in Chapter 8.

B.1 The HSM Skeleton Superstructure

```
/* $Id: image_thresholding.hsm,v 1.13 1997/07/18 17:33:53 marr Exp $
 *
 * Image Thresholding Algorithm - top level HSM code
 */


/*
 * define a new type to hold a single entry in the histogram
 */
typedef struct {
    int i;          /* pixel intensity */
    int c;          /* pixel count     */
} histogram_element_t;

/*
 * define the histogram type
```

```
 */
typedef array<histogram_element_t> histogram_t;


/*
 * Base level functions
 */

/* load the image */
basefn array<int> load_image_fn(
    string filename
);

/* save the image */
basefn void save_image_fn(
    string filename,
    array<int> image
);

/* sum two histograms together - used in the reduce skeleton */
basefn histogram_t histsum_fn(
    void global,                    /* there is no global argument */
    histogram_t histogram_a,
    histogram_t histogram_b
);

/* find the threshold intensity value */
basefn int find_threshold_fn(
    int threshold_count,
    histogram_t cumulative_histogram
);



/*
 * HSM functions
 */

/*
 * convert a pixel intensity into a histogram
 *     - used in the first map skeleton
 */
histogram_t inttohist_fn(
    void global,               /* there is no global argument */
    int pixel_intensity
)
{
    /* only one element is set as the others are implicitly set to
     * zero on creation of the array
```

```
    */
    histogram_t histogram;

    histogram = array_create(1, MAX_INTENSITY);
    histogram[pixel_intensity].i = pixel_intensity;
    histogram[pixel_intensity].c = 1;

    return( histogram );
}


/*
 * sum two elements of a single histogram
 *      - used in the scan skeleton
 */
histogram_element_t histelementsum_fn(
    void global,                        /* there is no global argument */
    histogram_element_t element_a,
    histogram_element_t element_b
)
{
    histogram_element_t element_c;

    element_c.i = element_a.i;
    element_c.c = element_a.c + element_b.c;

    return( element_c );
}


/*
 * convert an intensity pixel into a boolean pixel
 *      - used in the second map skeleton
 */
int threshold_fn(
    int threshold_intensity,     /* the global argument */
    int pixel_intensity
)
{
    return ( if( pixel_intensity>=threshold_intensity, 1, 0 ) );
}



void main(void)
{
    /*
     * declare the types of the variables:
     *      image : 2D array of ints
```

```
     *
     */
    int threshold_count, threshold_intensity;
    array<int> image, binary_image;
    histogram_t full_histogram, cumulative_histogram;
    array<histogram_t> histogram_array;

    /* load the image */
    image = load_image_fn( "image.data" );
    threshold_count = (int)(0.7 * (real)(array_size(image)[0]
                                          * array_size(image)[1]));

    /* map each pixel to its histogram */
    histogram_array = map( inttohist_fn, NULL, image );

    /* reduce the histograms */
    full_histogram = reduce( histsum_fn, NULL, histogram_array );

    /* find the cumulative frequencies */
    cumulative_histogram = scan( histelementsum_fn, full_histogram );

    /* find the actual threshold intensity value */
    threshold_intensity = find_threshold_fn( threshold_count,
                                              cumulative_histogram);

    /* perform the thresholding operation */
    binary_image = map( threshold_fn, threshold_intensity, image);

    /* save the resulting binary image */
    save_image_fn( "thresholded_image.data", binary_image );
}
```

B.2 The Pure HSM Solution

```
/* $Id: image_thresholding_purehsm.hsm,v 1.3 1997/07/18 17:33:54 marr Exp $
 *
 * Image Thresholding Algorithm - pure HSM solution
 */


/*
 * sum two histograms together
 *      - used in the reduce skeleton
 */
histogram_t histsum_fn(
    void global,                /* there is no global argument */
    histogram_t histogram_a,
    histogram_t histogram_b
```

```
)
{
    return( zip( histelementsum_fn, NULL, histogram_a, histogram_b ) );
}


/*
 * return true if the element has a pixel count at least as great
 * as the threshold
 *      - used in the filter skeleton
 */
bool histelementcmp_fn(
    int threshold_count,            /* global argument */
    histogram_element_t element
)
{
    return ( element.c >= threshold_count );
}


/*
 * find the threshold intensity value
 */
int find_threshold_fn(
    int threshold_count,
    histogram_t cumulative_histogram
)
{
    threshold_histogram = filter( histelementcmp_fn, threshold_count,
                                  cumulative_histogram);
    return( threshold_histogram[0].i );
}
```

B.3 The HSM/Fork95++ Solution

```
/* $Id: image_thresholding_fork.c,v 1.10 1997/07/08 23:30:14 marr Exp $
 *
 * Image Thresholding Algorithm - Fork95++ base language level
 * Assumes an arbirary or priority write-conflict resolution strategy.
 */


/*
 * load the image
 */
sync HSM_array<int>& image_load( HSM_string& filename )
{
```

```
    HSM_array<int>& image = *(new HSM_array<int>);

    farm if ($==0) {
        /* File I/O is only sequential at the moment */

        pr FILE *filehandle;
        pr int rows, columns, i, j;

        filehandle=fopen(filename, "r");
        fscanf(filehandle, "%d%d", &rows, &columns);

        image.create( 2, rows, columns );
        for (i=0; i<rows; i++) {
            for (j=0; j<columns; j++) {
                fscanf(filehandle, "%d", &image(i,j) );
            }
        }
    }

    return image;
}


/*
 * save the image
 */
sync void image_save( HSM_string& filename, HSM_array<int>& image )
{
    farm if ($==0) {
        /* File I/O is only sequential at the moment */

        pr FILE *filehandle;
        pr int i, j;

        filehandle=fopen(filename, "w");
        fprintf(filehandle, "%d %d\n", image.rows(), image.columns() );

        for (i=0; i<rows; i++) {
            for (j=0; j<columns; j++) {
                fprintf(filehandle, "%d", image(i,j));
            }
            printf("\n");
        }
    }
}



/*
```

```
 * sum two histograms together
 *      - used in the reduce skeleton
 */
sync HSM_histogram_t histsum_fn(
    HSM_void& global,
    HSM_histogram_t& histogram_a,
    HSM_histogram_t& histogram_b
)
{
    /*
     * sum the two histograms
     */
    pr int i;
    sh int n;
    HSM_histogram_t& histogram_c=*(shnew HSM_histogram);

    n=(sh int)histogram_a.size(0);     // async function
    histogram_c.create(1, n);          // sync function

    for (i=$; i<n; i+=groupsize()) {
        histogram_c(i).i = histogram_a(i);
        histogram_c(i).c = histogram_a(i).c+histogram_b(i).c;
    }

    return( histogram_c );
}



/*
 * find the threshold intensity value
 */
sync HSM_int& find_threshold_fn(
    HSM_int& threshold_count,
    HSM_histogram_t& cumulative_histogram
)
{
    pr int i;
    sh int n;
    HSM_int& threshold_intensity=*(shnew HSM_int);

    n=(sh int)histogram_a.size();     // async function
    threshold_intensity=0;
    for (i=$+1; i<n; i+=groupsize() ) {
        if (cumulative_histogram(i-1).c<threshold_count
            && cumulative_histogram(i).c>=threshold_count) {
            threshold_intensity=i;
        }
    }
```

```
    return( threshold_intensity );
}
```

B.4 The HSM/C++/MPI Solution

```
/* $Id: image_thresholding_c.c,v 1.17 1997/07/08 23:30:13 marr Exp $
 *
 * Image Thresholding Algorithm - C++/MPI base language level
 */


#include <assert.h>
#include <mpi.h>
#include <stdio.h>
#include "image_thresholding.hsm.c"


/*
 * load the image
 */
HSM_array<HSM_int>& image_load_fn( const HSM_string& filename )
{
    FILE *filehandle;
    int rows, columns, i, j;
    HSM_array<HSM_int>& image = *(new HSM_array<HSM_int>);

    filehandle=fopen(filename, "r");
    fscanf(filehandle, "%d%d", &rows, &columns);

    image.create(2, rows, columns, HSM_DISTRIBUTION_STANDARD);
    for (i=0; i<rows; i++) {
        for (j=0; j<columns; j++) {
            fscanf(filehandle, "%d", &image(i,j) );
        }
    }

    return image;
}


/*
 * save the image
 */
HSM_void image_save_fn( const HSM_string& filename, HSM_array<HSM_int>& image )
{
    FILE *filehandle;

    filehandle=fopen(filename, "w");
```

```
    fprintf(filehandle, "%d %d\n", image.size(0), image.size(1) );

    for (int i=0; i<image.size(0); i++) {
        for (int j=0; j<image.size(1); j++) {
            fprintf(filehandle, "%d", image(i,j));
        }
        printf("\n");
    }
}


/*
 * sum two histograms together
 *      - used in the reduce skeleton
 */
HSM_histogram_t& histsum_fn(
    HSM_void& global,                    /* there is no global argument */
    HSM_histogram_t& histogram_a,
    HSM_histogram_t& histogram_b
)
{
    /*
     * This function should be parallel.
     *
     * The first histogram will be distributed evenly over the first
     * half of processors, and the second histogram over the second
     * half.  The resulting histogram should be evenly distributed
     * over all the processors.
     *
     * This can be done by sending each processor the elements
     * required for the resulting distribution.  This may require two
     * messages for each histogram.
     */

    int rank, size, i, n;
    HSM_histogram_t& histogram_c=*(new HSM_histogram);
    HSM_histogram_t histogram_newa, histogram_newb;

    /* the index tuples are just integers */
    int first, last, mid, half_size;
    int left_destination, right_destination;

    assert( histogram_a.size(0) == histogram_b.size(0) );
    assert( histogram_a.distribution() == HSM_DISTRIBUTION_STANDARD );
    assert( histogram_b.distribution() == HSM_DISTRIBUTION_STANDARD );

    n = histogram_a.size(0);
    histogram_c.create(1, n, HSM_DISTRIBUTION_STANDARD);
```

```
MPI_Comm_rank(HSM_Comm_world, &rank);
MPI_Comm_size(HSM_Comm_world, &size);

if (size==1) {
    /* There's only one processor, so there's nothing parallel
     * about it.
     * Standard array notation is used.
     */
    for (i=0; i<histogram_c.size(0); i++) {
        histogram_c(i).i = histogram_a(i).i;
        histogram_c(i).c = histogram_a(i).c + histogram_b(i).c;
    }
}
else {
    /* Distribute the histograms to where they are needed and
     * perform the summation in parallel.
     *
     * By building up from the standard distribution, we know that
     * the resulting structure will have half as many elements as
     * the input structures.  If we restrict the numbers of
     * processor to powers of two the boundaries match up so that
     * the local input elements can be divided into two blocks
     * with each block being sent where it is needed.
     */

    /* the index tuples are just integers */
    int left_destination, right_destination, block_size;
    HSM_index first, last, ind;
    HSM_histogram_t histogram_newa, histogram_newb;
    MPI_Status status;

    /* create new data structures to hold the redistributed input
     * arrays */
    histogram_newa.create(1, n, HSM_DISTRIBUTION_STANDARD);
    histogram_newb.create(1, n, HSM_DISTRIBUTION_STANDARD);

    /* the block size is the number of elements needed for the
     * redistributed data structures */
    first = histogram_c.first_local();
    last = histogram_c.last_local();
    block_size = last - first + 1;

    if (rank>=histogram_a.distribution_range.from
        && rank<=histogram_a.distribution_range.to) {
        /* processor belongs to distribution of first histogram */

        /* find the index of the first and last elements and
         * calculate the number of elements in each block */
        first = histogram_a.first_local();
```

```
        /* find out where the elements need to be sent */
        left_destination=histogram_c.element(first).where_is();
        right_destination=histogram_c.element(first+blocksize).where_is();

        /* send the two blocks */
        MPI_Send( &histogram_a.element(first), block_size,
                  histogram_a.mpi_type(), left_destination,
                  0, HSM_Comm_world);
        MPI_Send( &histogram_a.element(first+block_size), block_size,
                  histogram_a.mpi_type(), right_destination,
                  0, HSM_Comm_world);
    }
    else {
        /* processor belongs to distribution of second histogram */

        /* find the index of the first and last elements and
         * calculate the number of elements in each block */
        first = histogram_b.first_local();

        /* find out where the elements need to be sent */
        left_destination=histogram_c.element(first).where_is();
        right_destination=histogram_c.element(first+block_size).where_is();

        MPI_Send( &histogram_b.element(first), block_size,
                  histogram_b.mpi_type(), left_destination,
                  0, HSM_Comm_world);
        MPI_Send( &histogram_b.element(first+block_size), block_size,
                  histogram_b.mpi_type(), right_destination,
                  0, HSM_Comm_world);
    }

    /* receive one block from each input array */
    int a_source, b_source;

    first = histogram_c.first_local();
    a_source = histogram_a.element(first).where_is();
    b_source = histogram_b.element(first).where_is();
    MPI_Recv( &histogram_newa.element(first), block_size,
              histogram_newa.mpi_type(), a_source,
              0, HSM_Comm_world, &status);
    MPI_Recv( &histogram_newb.element(first), block_size,
              histogram_newb.mpi_type(), b_source,
              0, HSM_Comm_world, &status);

    /* iterate through the index tuples stored locally, updating
     * the result histogram */
    for (ind=histogram_c.first_local();
         ind<=histogram_c.last_local();
```

```
            ind=histogram_c.next_local()) {
            histogram_c.element(ind).i = histogram_newa.element(ind).i;
            histogram_c.element(ind).c = histogram_newa.element(ind).c
                                        + histogram_newb.element(ind).c;
        }
    }
}


/*
 * find the threshold intensity value
 */
HSM_int& find_threshold_fn(
    HSM_int& threshold_count,
    HSM_histogram_t cumulative_histogram
)
{
    int size, i, found, local_index, *indices;
    HSM_int& threshold_intensity=*(new HSM_int);
    HSM_index ind;

    MPI_Comm_size(HSM_Comm_world, &size);

    /* find the local threshold value */
    found=0;
    local_index=-1;
    for (ind=cumulative_histogram.first_local();
         !found && ind<=cumulative_histogram.last_local();
         ind=cumulative_histogram.next_local()) {
        found = (cumulative_histogram.element(ind)>=threshold_count);
        local_index = ind[0];
    }
    if (size==1) {
        threshold_intensity = local_index;
    }
    else {
        /* need to resolve conflict over all the processors */
        indices = new int[size];

        MPI_Gather( local_index, 1, MPI_INT, indices, 1, MPI_INT,
                    threshold_intensity.where_is(), MPI_COMM_WORLD );

        if (threshold_intensity.is_local()) {
            i=0;
            while (i<size && indices[i]<0) i++;
            threshold_intensity = indices[i];
        }
    }
```

```
    return( threshold_intensity );
}
```

B.5 The Pure Fork95++ Solution

```
/* $Id: image_thresholding_purefork.c,v 1.6 1997/07/08 23:30:16 marr Exp $
 *
 * Image Thresholding Algorithm - Pure Fork95++ code
 * Assumes an arbirary or priority write-conflict resolution strategy.
 */

#define MAX_INTENSITY 65536

#include <fork.h>


/*
 * a class for two dimensional arrays
 */
class array2d_t {
    sh int *data;
    sh int nrows, ncolumns;
public:
    sync void create( sh int rows, sh int cols ) {
        nrows=rows;
        ncolumns=cols;
        data = shnew int[nrows*ncolumns];
    }
    async int rows() {
        return nrows;
    }
    async int columns() {
        return ncolumns;
    }
    async int& operator()( pr int i, pr int j ) {
        return data[i*rows+j];
    }
}



/*
 * load the image
 */
sync sh array2d_t& image_load( sync char* filename )
{
    sh array2d_t& image = *(new array2d_t);
```

```
    farm if ($==0) {
        /* File I/O is only sequential at the moment */

        pr FILE *filehandle;
        pr int rows, columns, i, j;

        filehandle=fopen(filename, "r");
        fscanf(filehandle, "%d%d", &rows, &columns);

        image.create( rows, columns );
        for (i=0; i<rows; i++) {
            for (j=0; j<columns; j++) {
                fscanf(filehandle, "%d", &image(i,j) );
            }
        }
    }

    return image;
}



/*
 * save the image
 */
sync void image_save( sync char* filename, array2d_t& image )
{
    farm if ($==0) {
        /* File I/O is only sequential at the moment */

        pr FILE *filehandle;
        pr int i, j;

        filehandle=fopen(filename, "w");
        fprintf(filehandle, "%d %d\n", image.rows(), image.columns() );

        for (i=0; i<rows; i++) {
            for (j=0; j<columns; j++) {
                fprintf(filehandle, "%d", image(i,j));
            }
            printf("\n");
        }
    }
}



/*
```

```
 * the main function
 */
void main(void)
{
    /* declare private variables */
    pr int i, row, col;

    start {
        /* processors must all be synchronised before shared
         * variables ca be declared
         */
        sh double threshold_fraction=0.7;
        sh int threshold_count, threshold_intensity;
        sh int sum, shared_i;
        sh array2d_t image, binary_image;
        sh int histogram[MAX_INTENSITY];
        sh int cumulative_histogram[MAX_INTENSITY];

        /* load the image */
        image = image_load("image.data");
        threshold_count = (int)(threshold_fraction *
                                (real)(image.rows()*image.columns()));

        /* reset the histogram */
        for (i=$; i<MAX_INTENSITY; i+=groupsize(); ) {
            histogram[i]=0;
        }

        /* calculate the global histogram */
        for (i=$; i<image.rows()*image.columns(); i+=groupsize()) {
            row=i/image.columns();
            col=i%image.columns();
            mpadd( &histogram[image(row,col)], 1 );
        }

        /* perform the prefix sum operation on the histogram */
        sum = 0;
        for (i=$; i<MAX_INTENSITY; i+=groupsize()) {
            /* note that mpadd() returns the sum *before* adding */
            cumulative_histogram[i] = histogram[i]
                + mpadd( &sum, histogram[i] );
        }

        /* find the threshold value */
        threshold_intensity=0;
        for (i=$+1; i<n; i+=groupsize()) {
            if (cumulative_histogram[i-1].c<threshold_count
                && cumulative_histogram[i].c>=threshold_count) {
                threshold_intensity=i;
```

```
        }
    }

    /* construct the binary image */
    binary_image.create( image.rows(), image.columns() )
    for (i=$; i<image.rows()*image.columns(); i+=groupsize()) {
        row=i/image.columns();
        col=i%image.columns();
        val=image(row,col);
        if (image(row,col)>=threshold_intensity) {
            binary_image(row,col)=1;
        }
        else {
            binary_image(row,col)=0;
        }
    }

    /* save the image */
    image_save("thresholded_image.data", binary_image);
  }
}
```

B.6 The Pure C++/MPI Solution

```
/* $Id: image_thresholding_purec.c,v 1.6 1997/07/08 23:30:15 marr Exp $
 *
 * Image Thresholding Algorithm - Pure C++/MPI code
 */


#define MAX_INTENSITY 65536

#include <assert.h>
#include <mpi.h>
#include <stdio.h>
#include "image_thresholding.hsm.c"


class array2d_t {
    int *data;
    int nrows, ncolumns;
public:
    void create( int rows, int cols ) {
        nrows=rows;
        ncolumns=cols;
        data = new int[nrows*ncolumns];
    }
    int rows() {
```

```
        return nrows;
    }
    int columns() {
        return ncolumns;
    }
    int& operator()( int i, int j ) {
        return data[i*rows+j];
    }
}


/*
 * load the image
 */
array2d_t& image_load_fn( char *filename )
{
    FILE *filehandle;
    int rows, columns, i, j;
    array2d_t& image = *(new array2d_t);

    filehandle=fopen(filename, "r");
    fscanf(filehandle, "%d%d", &rows, &columns);

    image.create( rows, columns );
    for (i=0; i<rows; i++) {
        for (j=0; j<columns; j++) {
            fscanf(filehandle, "%d", &image(i,j) );
        }
    }

    return image;
}


/*
 * save the image
 */
HSM_void image_save_fn( char *filename, array2d_t& image )
{
    FILE *filehandle;

    filehandle=fopen(filename, "w");
    fprintf(filehandle, "%d %d\n", image.rows(), image.columns() );

    for (int i=0; i<image.rows(); i++) {
        for (int j=0; j<image.columns(); j++) {
            fprintf(filehandle, "%d", image(i,j));
        }
        printf("\n");
```

```
    }
}


/*
 * main function
 */
void main(int argc, char *argv[]) {
{
    int rank, size, nrows, ncols, i, j;
    double threshold_fraction=0.7;
    int threshold_count, threshold_intensity;
    array2d_t image, full_image;
    int local_histogram[MAX_INTENSITY], full_histogram[MAX_INTENSITY];
    int cumulative_histogram[MAX_INTENSITY];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(HSM_Comm_world, &rank);
    MPI_Comm_size(HSM_Comm_world, &size);

    /* load the image to the root processor*/
    if (rank==0) {
        full_image = image_load_fn("image.data");
        threshold_count = (int)(threshold_fraction *
                              (double)(full_image.rows()*full_image.columns()));
        nrows=full_image.rows();
        ncols=full_image.columns();
    }

    /* Broadcast the total dimensions to everyone */
    MPI_Bcast( &nrows, 1, MPI_INT, 0, MPI_COMM_WORLD );
    MPI_Bcast( &ncols, 1, MPI_INT, 0, MPI_COMM_WORLD );

    /* create local image */
    assert( nrows%size==0 );
    image.create(nrows/size, ncols);

    MPI_Scatter( &full_image(0,0), nrows*ncols/size, MPI_INT,
                 &image(0,0), nrows*ncols/size, MPI_INT,
                 0, MPI_COMM_WORLD );

    /* reset the local histogram */
    for (i=0; i<MAX_INTENSITY; i++) {
      local_histogram[i]=0;
    }

    /* calculate the local histogram */
    for (i=0; i<image.rows(); i++) {
        for (j=0; j<image.columns(); j++) {
```

```
            local_histogram[image(i,j)]++;
        }
    }

    /* reduce the local histograms to a single full histogram on
     * the root processor
     */
    MPI_Reduce( local_histogram, full_histogram, MAX_INTENSITY, MPI_INT,
                MPI_SUM, 0, MPI_COMM_WORLD) ;

    if (rank==0) {
        /* calculate the threshold value */
        int count=0;
        threshold_intensity=0;
        while (count<threshold_count
               && threshold_intensity<MAX_INTENSITY) {
          count+=full_histogram[threshold_intensity];
          threshold_intensity++;
        }
    }
    MPI_Bcast( &threshold_intensity, 1, MPI_INT, 0, MPI_COMM_WORLD );

    /* construct the local binary image */
    for (i=0; i<image.rows(); i++) {
        for (j=0; j<image.columns(); j++) {
            image(i,j) = (image(i,j)>=threshold_intensity);
        }
    }

    /* gather the full image back to the root processor */
    MPI_Gather( &image(0,0), nrows*ncols/size, MPI_INT,
                &full_image(0,0), nrows*ncols/size, MPI_INT,
                0, MPI_COMM_WORLD );

    /* save the resulting binary image */
    if (rank==0) {
        image_save_fn("thresholded_image.data", full_image);
    }

    MPI_Finalize();
}
```

Appendix C

The Merge-Sort Problem

Contents

C.1 The HSM Skeleton Superstructure 185
C.2 The Pure HSM Solution 188
C.3 The HSM/Fork95++ Solution 192
C.4 The HSM/C++/MPI Solution 193
C.5 The Pure Fork95++ Solution 198
C.6 The Pure C++/MPI Solution 201

This appendix gives the full code listings of the merge-sort solutions described in Chapter 9.

C.1 The HSM Skeleton Superstructure

```
/* $Id: mergesort.hsm,v 1.5 1997/07/08 23:30:17 marr Exp $
 *
 * Merge-Sort Algorithm - top level HSM code
 */


/*
 * Base level functions
 */

/* load the numbers */
basefn array<int> load_numbers_fn(
    string filename
);

/* save the numbers */
```

```
basefn void save_numbers_fn(
    string filename,
    array<int> alpha
);

/* the merge operation */
basefn array<int> merge_fn(
    void global,      /* there is no global argument */
    array<int> a,     /* the first array */
    array<int> b      /* the second array */
);



/*
 * HSM functions - recursive definition
 */

/* sort the array of numbers, using either the base case or the
 * recursive case */
array<int> sort_fn(
    array<int> numbers
)
{
    int n;

    n = array_size(numbers)[0];
    return(
        if ( n==1,
            numbers,                          /* base case */
            divide_and_merge_fn( numbers )    /* recursive case */
        )
    );
}

/* sort the array of numbers, using recursion */
array<int> divide_and_merge_fn(
    array<int> numbers
)
{
    int n;
    array<array<int>> list, new_list;
    array<int> new_numbers;

    n = array_size(numbers)[0];
    list = array_create( 1, 2 );
    list[0] = numbers[0..n/2-1];
    list[1] = numbers[n/2..n-1];
    new_list = map( sort_fn, NULL, list );
```

```
    new_numbers = merge_fn( NULL, new_list[0], new_list[1] );
    return( new_numbers );
}


/*
 * HSM function - if using the map skeleton instead of the divide phase
 */

/* convert an integer into a singleton list */
array<int> inttosingleton_fn(
    void global,
    int x
)
{
    array<int> singleton;

    singleton = array_create( 1, 1 );
    singleton[0] = x;
    return( singleton );
}



/*
 * HSM functions - divide phase and reduce phase solution
 */

/* the divide test */
bool divide_test_fn(
    void global,        /* there is no global argument */
    array<int> alpha    /* the array of numbers */
)
{
    /* return true if the array should be divided */
    return( array_size(alpha)[0]>1 );
}

/* the divide test for a single processor base case */
bool cpp_divide_test_fn(
    void global,        /* there is no global argument */
    array<int> alpha    /* the array of numbers */
)
{
    /* return true if the array should be divided */
    return( node_size()>1 && array_size(alpha)[0]>1 );
}
```

```
/* the divide operation */
array<array<int>> divide_numbers_fn(
    void global,          /* there is no global argument */
    array<int> numbers    /* the array of numbers */
)
{
    /* divide the array into two smaller arrays */
    int n;
    array<array<int>> list;

    /* create a one-dimension array to hold the two sub-arrays */
    list = array_create( 1, 2 );
    n = array_size(numbers)[0];
    list[0] = numbers[0..n/2-1];
    list[1] = numbers[n/2.. n-1];

    return( list );
}


/* the main routine */
void main(void)
{
    array<int> numbers;
    array<array<int>> singletons;

    /* load the array of numbers */
    numbers = load_numbers_fn( "unsorted.data" );

    /* divide the array into singleton arrays */
    singletons = divide( divide_test_fn, divide_numbers_fn, NULL, numbers );

    /* reduce the array of singleton arrays back to one array */
    numbers = reduce( merge_fn, NULL, singletons );

    /* save the sorted array */
    save_numbers_fn( "sorted.data", numbers );
}
```

C.2 The Pure HSM Solution

```
/* $Id: mergesort_purehsm.hsm,v 1.6 1997/07/18 17:34:44 marr Exp $
 *
 * Merge-Sort Algorithm - pure HSM solution
 */


typedef struct { int a, b; } int_pair;
```

```
typedef struct { array<int> a, b; } array_pair;


/*
 * concatenate two arrays
 */
array<int> arrayconcat_fn(
    void global,
    array<int> a,
    array<int> b
)
{
    return( a++b );
}


/*
 * find the rank of an element in a list
 *      - this is a parallel algorithm
 */
int find_rank_fn(
    array<int> list,     /* global argument */
    int pivot_val
)
{
    array<int> indices;
    array<int_pair> indexed_list, greater_elements;

    indices = vector_create_indexed( array_size(list)[0] );
    indexed_list = zip( inttointpair_fn, NULL, indices, list );
    greater_elements = filter( intpair_cmp, pivot_val, indexed_list );
    return( greater_elements[0].a );
}


/*
 * decide whether or not to divide a pair of arrays further
 */
bool divide_pair_test_fn(
    void global,
    array_pair alpha
)
{
    /* return true if the arrays should be divided further,
     * ie the arrays overlap
     */
    return( array_size(alpha.a)[0]>0
            && array_size(alpha.b)[0]>0
```

```
            && alpha.a[0]<alpha.b[array_size(alpha.b)[0]-1]
            && alpha.a[array_size(alpha.a)[0]-1]>alpha.b[0] );
}


/*
 * divide a pair of sorted arrays into two disjoint pairs of sorted
 * arrays
 */
array<array_pair> divide_pair_fn(
    void global,
    array_pair alpha
)
{
    /* divide the pair of arrays into two independent pairs of arrays,
     * knowing that the arrays are guarnateed to overlap
     */
    int n, pivot_val, pivot_rank_in_a, pivot_rank_in_b;
    array<array_pair> pair_list;

    /* select a pivot value in alpha.a which lies within the
     * range of alpha.b
     */
    n = array_size(alpha.a)[0];
    pivot_val = alpha.a[n/2];
    pivot_val = if (pivot_val>=alpha.b[0], pivot_val, alpha.b[0] );
    pivot_val = if (pivot_val>=alpha.b[0], pivot_val, alpha.b[0] );
    pivot_val = if (pivot_val<=alpha.b[array_size(alpha.b)[0]-1],
                    pivot_val, alpha.b[array_size(alpha.b)[0]-1] );

    /* find the rank of the pivot value in each input array
     * and decide where the arrays should be divided
     */
    pivot_rank_in_a = find_rank_fn( alpha.a, pivot_val );
    end_of_a = if ( alpha.a[pivot_rank_in_a]==pivot_val,
                    pivot_rank_in_a, pivot_rank_in_a-1 );
    pivot_rank_in_b = find_rank_fn( alpha.b, pivot_val );
    end_of_b = if ( alpha.b[pivot_rank_in_b]==pivot_val,
                    pivot_rank_in_b, pivot_rank_in_b-1 );

    /* the following array subscripts rely on the HSM language
     * correctly returning zero length arrays where appropriate
     */
    pair_list = array_create( 1, 2 );
    pair_list[0].a = alpha.a[0..end_of_a];
    pair_list[0].b = alpha.b[0..end_of_b];
    pair_list[1].a = alpha.a[end_of_a+1 .. array_size(alpha.a)[0]-1];
    pair_list[1].b = alpha.b[end_of_b+1 .. array_size(alpha.b)[0]-1];
}
```

```
/*
 * convert a pair of disjoint sorted arrays into a single
 * sorted array
 */
array<int> pairtoarray_fn(
    void global,
    array_pair alpha
)
{
    return(
        if ( array_size(alpha.a)[0]==0, alpha.b,
             if ( array_size(alpha.b)[0]==0, alpha.a,
                  if ( alpha.a[array_size(alpha.a)[0]−1]<=alpha.b[0],
                       alpha.a++alpha.b,
                       alpha.b++alpha.a
                  )
             )
        )
    );
}


/*
 * merge two sorted arrays into a single sorted array
 */
array<int> merge_fn(
    void global,      /* there is no global argument */
    array<int> a,     /* the first array */
    array<int> b      /* the second array */
)
{
    array_pair original_pair;
    array<array_pair> pair_list;
    array<array<int>> array_list;
    array<int> final_array;

    original_pair.a = a;
    original_pair.b = b;

    pair_list = divide( divide_pair_test_fn, divide_pair_fn,
                        NULL, original_pair );
    array_list = map( pairtoarray_fn, NULL, pair_list );
    final_array = reduce( arrayconcat_fn, NULL, array_list );

    return( final_array );
}
```

C.3 The HSM/Fork95++ Solution

```
/* $Id: mergesort_fork.c,v 1.3 1997/07/08 23:30:19 marr Exp $
 *
 * Merge-Sort Algorithm - Fork95++ base language level
 */


sync HSM_array<int>& merge_fn(
    HSM_void& global,       /* there is no global argument */
    HSM_array<int>& a,      /* the first array */
    HSM_array<int>& b       /* the second array */
)
{
    pr int i, rank;
    HSM_array<int>& c=*(shnew HSM_array<int>);

    c.create( 1, a.size(0)+b.size(0) );
    for ( i=$, i<a.size(0); i+=groupsize() ) {
        rank = find_rank_fn( b, 0, b.size(0)-1, a(i) );
        while ( rank>0 && a(i)==b(rank-1) ) rank--;
        c(rank+i) = a(i);
    }
    for ( i=$, i<b.size(0); i+=groupsize() ) {
        rank = find_rank_fn( a, 0, a.size(0)-1, b(i) );
        while ( rank<a.size(0) && b(i)==a(rank) ) rank++;
        c(rank+i) = b(i);
    }

    return( c );
}

async pr int find_rank_fn(
    HSM_array<int>& list,
    pr int strt,
    pr int end,
    pr int val
)
{
    pr int rank, mid;

    if (end<strt) {
        rank = strt;            /* base case */
    }
    else {
        mid=(strt+end)/2;
        if (val==list(mid)) {
            rank = mid;         /* exact element found */
        }
```

```
        else if (val<list(mid)) {
            rank = find_rank_fn( list, strt, mid-1, val );
        }
        else {
            rank = find_rank_fn( list, mid+1, end, val );
        }
    }

    return( rank );
}
```

C.4 The HSM/C++/MPI Solution

```
/* $Id: mergesort_c.c,v 1.5 1997/07/08 23:30:18 marr Exp $
 *
 * Merge-Sort Algorithm - C++/MPI base language level
 */


#include <assert.h>
#include <mpi.h>
#include <stdlib.h>
#include "mergesort.hsm.c"


/* Integer comparison function
 *      - used in call for qsort()
 */
int intcompare(int *i, int *j)
{
    int ret=0;

    if ( *i>*j ) {
        ret=1;
    }
    else if ( *i<*j ) {
        ret=-1;
    }

    return( ret );
}

/* Sequential sort operation */
HSM_array<int>& seq_sort_fn( HSM_array<int>& list )
{
    qsort( &list(0), list.size(0), sizeof(int), intcompare_fn );
    return( list );
}
```

```
/* Merge operation */
HSM_array<int>& merge_fn(
    HSM_void global,
    HSM_array<int>& list_a,
    HSM_array<int>& list_b
)
{
    /* Select suitable pivot elements for splitting the list.
     * This is done by selecting N elements from each of A and B,
     * sorting them and then using every second element.
     */

    int rank, size;
    int i, j, n, local_selected[2], *all_selected;

    MPI_Comm_rank(HSM_Comm_world, &rank);
    MPI_Comm_size(HSM_Comm_world, &size);
    all_selected = new int[size*2];

    /* each processor will select exactly two tripples */
    n = list_a.size(0)/size;
    for (i=0, j=0; i<list_a.size(0); i+=n) {
        if (a(n).is_local()) {
            local_selected[j++]=a(n);
        }
    }
    for (i=0, j=0; i<list_b.size(0); i+=n) {
        if (b(n).is_local()) {
            local_selected[j++]=b(n);
        }
    }
    MPI_Allgather( local_selected, 2, MPI_INT,
                   all_selected, 2, MPI_INT, HSM_Comm_world );

    /* sort the selected elements
     *     - this is actually a merge, but using qsort is easier
     */
    qsort( all_selected, 2*size, sizeof(int), intcompare_fn );

    /* Each processor merges and insterts into C the elements of two
     * subsequences, one from A and one from B.
     */

    int pivot_val[size-1];

    for (i=0; i<size-1; i++) {
```

```
    pivot_val[i] = all_selected[i*2+1];
}



/*
 * Redistribute array 'a'
 */

/* find out how many elements need to be sent to each processor */

HSM_index ind = a.first_local();

int send_block_size[size];
for (i=0; i<size; i++) send_block_size[i]=0;

for (i=0; i<size; i++) {
    while (ind<=a.last_local()
           && (i==size-1 || a.element(ind)<pivot_val[i])) {
        send_block_size[i]++;
    }
}

/* send the elements to the respective processors
 *     - strictly speaking I should set the buffer space for the
 *       bufferd send
 */
ind = a.first_local();
for (i=0; i<size; i++) {
    if (send_block_size[i]>0) {
        MPI_Bsend( &a.element(ind), send_block_size[i], MPI_INT,
                   i, 0, HSM_Comm_world );
        ind+=send_block_size[i];
    }
}

/* find out how many elements will be received from each processor */
int recv_block_size[size];
MPI_Alltoall( send_block_size, 1, MPI_INT,
              recv_block_size, 1, MPI_INT, HSM_Comm_world );

/* find out the total number of local elements to be received */
for (i=0, count_a=0; i<size; i++) {
    count_a+=recv_block_size[i];
}

/* send the results to everyone so that they can calculate
 * the index of their first element
 */
```

```
int block_size[size];
MPI_Allgather( &count_a, 1, MPI_INT,
               block_size, 1, MPI_INT, HSM_Comm_world );
for (i=0, offset_a=0; i<rank; i++) {
    offset_a+=block_size[i];
}

/* create the new ad-hoc array */
HSM_array<int> new_a;
new_a.create( 1, a.size(0), HSM_DISTRIBUTION_ADHOC );
for (i=0; i<count_a; i++) {
    new_a.add_local(offset_a+i);
}

/* receive the data elements from everyone */
MPI_Status status;
for (i=0, offset=offset_a; i<size; i++) {
    if (recv_block_size[i]>0) {
        MPI_Recv( &new_a(offset), recv_block_size[i], MPI_INT,
                  i, 0, HSM_Comm_world, &status );
        offset+=recv_block_size[i];
    }
}



/*
 * Redistribute array 'b'
 */

/* find out how many elements need to be sent to each processor */

HSM_index ind = b.first_local();

for (i=0; i<size; i++) send_block_size[i]=0;
for (i=0; i<size; i++) {
    while (ind<=b.last_local()
           && (i==size-1 || b.element(ind)<pivot_val[i])) {
        send_block_size[i]++;
    }
}

/* send the elements to the respective processors */
ind = b.first_local();
for (i=0; i<size; i++) {
    if (send_block_size[i]>0) {
        MPI_Bsend( &b.element(ind), send_block_size[i], MPI_INT,
                   i, 0, HSM_Comm_world );
        ind+=send_block_size[i];
```

```
    }
}

/* find out how many elements will be received from each processor */
MPI_Alltoall( send_block_size, 1, MPI_INT,
              recv_block_size, 1, MPI_INT, HSM_Comm_world );

/* find out the total number of local elements to be received */
for (i=0, count_b=0; i<size; i++) {
    count_b+=recv_block_size[i];
}

/* send the results to everyone so that they can calculate
 * the index of their first element
 */
int block_size[size];
MPI_Allgather( &count_b, 1, MPI_INT,
               block_size, 1, MPI_INT, HSM_Comm_world );
for (i=0, offset_b=0; i<rank; i++) {
    offset_b+=block_size[i];
}

/* create the new ad-hoc array */
HSM_array<int> new_b;
new_b.create( 1, b.size(0), HSM_DISTRIBUTION_ADHOC );
for (i=0; i<count_b; i++) {
    new_b.add_local(offset_b+i);
}

/* receive the data elements from everyone */
MPI_Status status;
for (i=0, offset=offset_b; i<size; i++) {
    if (recv_block_size[i]>0) {
        MPI_Recv( &new_b(offset), recv_block_size[i], MPI_INT,
                  i, 0, HSM_Comm_world, &status );
        offset+=recv_block_size[i];
    }
}

/* The locally stored arrays can now be merged sequentially, but
 * the result array 'c' needs to be declared with an ad-hoc
 * distribution.
 */

c.create( 1, a.size(0)+b.size(0), HSM_DISTRIBUTION_ADHOC );
count_c = count_a+count_b;
offset_c = offset_a+offset_b;
for (i=0; i<count_c; i++) {
    c.add_local(offset_c+i);
```

```
    }

    ind_a = new_a.first_local();
    ind_b = new_b.first_local();
    for ( ind_c = c.first_local();
          ind_c <= c.last_local();
          ind_c = c.next_local() ) {
        if (ind_a>new_a.last_local()) {
            c.element(ind_c) = new_b.element(ind_b);
            ind_b = new_b.next_local();
        }
        else if (ind_b>new_b.last_local()) {
            c.element(ind_c) = new_a.element(ind_a);
            ind_a = new_a.next_local();
        }
        else if (new_a.element(ind_a)<=new_b.element(ind_b)) {
            c.element(ind_c) = new_a.element(ind_a);
            ind_a = new_a.next_local();
        }
        else {
            c.element(ind_c) = new_b.element(ind_b);
            ind_b = new_b.next_local();
        }
    }

    return( c );
}
```

C.5 The Pure Fork95++ Solution

```
/* $Id: mergesort_purefork.c,v 1.6 1997/07/10 15:47:59 marr Exp $
 *
 * Merge-Sort - Pure Fork95++ code
 */

#include <fork.h>

#define MAX_NUMBERS 65536



/*
 * a class for one dimensional arrays
 */
class array1d_t {
    sh int *data;
    sh int n;
public:
```

```
    sync void create( sh int nelems ) {
        n=nelems;
        data = shnew int[n];
    }
    async int nelems() {
        return n;
    }
    async int& operator[]( pr int i ) {
        return data[i];
    }
}


async pr int find_rank_fn(
    pr array1d_t& list,
    pr int start,
    pr int end,
    pr int val
)
{
    pr int rank, mid;

    if (end<start) {
        rank = start;            /* base case */
    }
    else {
        mid=(start+end)/2;
        if (val==list[mid]) {
            rank = mid;          /* exact element found */
        }
        else if (val<list[mid]) {
            rank = find_rank_fn( list, start, mid-1, val );
        }
        else {
            rank = find_rank_fn( list, mid+1, end, val );
        }
    }

    return( rank );
}


/*
 * the main function
 */
void main(void)
{
```

```
  pr int rank;

  start {
    sh array1d_t numbers;
    sh int local_size;
    numbers = numbers_load_fn( "numbers.data" );

    /* make sure there are enough processors to go round */
    assert( numbers.nelems()==groupsize() );

    local_size = 1;
    while (local_size<=groupsize()) {
      fork( numbers.nelems()/local_size; @=$%(numbers.nelems()/local_size); ) {
        /* calculate the ranks of the left half */
        if ($%local_size < local_size/2) {
          /* processor is in left half */
          block_start=($/local_size)*local_size+local_size/2;
          block_end=block_start+local_size/2-1;
          rank = find_rank_fn( numbers, block_start, block_end,
                               numbers[$] );
          while (rank>0
                 && numbers[$]==numbers[block_start+rank-1]) {
            rank--;
          }
          rank += $;
        }
        else {
          /* processor is in right half */
          block_start=($/local_size)*local_size;
          block_end=block_start+local_size/2-1;
          rank = find_rank_fn( numbers, block_start, block_end,
                               numbers[$] );
          while (rank<local_size/2
                 && numbers[$]==numbers[block_start+rank]){
            rank++;
          }
          rank += $-(local_size/2);
        }
        numbers[rank]=numbers[$];
      }
      local_size*=2;
    }
  }
  numbers_save("sorted_numbers.data", numbers);
}
```

C.6 The Pure C++/MPI Solution

```
/* $Id: mergesort_purec.c,v 1.6 1997/07/10 16:22:27 marr Exp $
 *
 * Merge-Sort Algorithm - Pure C++/MPI code
 */


#include <assert.h>
#include <mpi.h>
#include <stdio.h>
#include "image_thresholding.hsm.c"


class array1d_t {
    int *data;
    int n;
public:
    void create( int nelems ) {
        n=nelems;
        data = new int[n];
    }
    int nelems() {
        return n;
    }
    int& operator[]( int i ) {
        return data[i];
    }
}


/* Integer comparison function
 *      - used in call for qsort()
 */
int intcompare(int *i, int *j)
{
    int ret=0;

    if ( *i>*j ) {
        ret=1;
    }
    else if ( *i<*j ) {
        ret=-1;
    }

    return( ret );
}

/*
```

```
 * main function
 */
void main(int argc, char *argv[]) {
{
    int rank, size, nelems, neighbour, jump;
    array1d_t full_numbers, local_numbers, tmp_numbers_a, tmp_numbers_b;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(HSM_Comm_world, &rank);
    MPI_Comm_size(HSM_Comm_world, &size);

    if (rank==0) {
        /* load the numbers from a file */
        full_numbers = numbers_load_fn( "numbers.data" );
        nelems = full_numbers.nelems();
        assert( nelems%size==0 );
    }
    else {
        full_numbers.create( 0 );
    }

    /* broadcast the number of elements to be held by each processor */
    MPI_Bcast( nelems, 1, MPI_INT, 0, MPI_COMM_WORLD );

    /* scatter the elements
     *      - arrays must be distinct
     */
    local_numbers.create( nelems )
    MPI_Scatter( &full_numbers[0], nelems/size, MPI_INT,
                 &local_numbers[0], nelems/size, MPI_INT,
                 0, MPI_COMM_WORLD );

    /* sort the local elements */
    qsort( &local_numbers[0], nelems/size, sizeof(int), intcompare_fn );

    /* perform the reduction */
    tmp_numbers_a.create( nelems );
    tmp_numbers_b.create( nelems );
    for ( jump=1; jump<size && rank%jump==0; jump*=2 ) {
        /* find out who the neighbour is */
        if (rank%(jump*2)==0) {
            neighbour=rank+jump;
        }
        else {
            neighbour=rank-jump;
        }

        if (neighbour<rank) {
```

```
            /* just send the entire array */
            MPI_Send( &local_numbers[0], nelems/size*jump, MPI_INT,
                    neighbour, 0, MPI_COMM_WORLD );
        }
        else {
            /* receive the neighbours array */
            MPI_Recv( &tmp_numbers_a[0], nelems/size*jump, MPI_INT,
                    neighbour, 0, MPI_COMM_WORLD, &status );
            /* copy the local array */
            for (i=0; i<nelems/size*jump; i++) {
                tmp_numbers_b[0]=local_numbers[0];
            }

            for (a=0, b=0, i=0; i<2*nelems/size*jump; i++) {
                if (a>=nelems/size*jump) {
                    local_numbers[i++]=tmp_numbers_b[b++];
                }
                else if (b>=nelems/size*jump) {
                    local_numbers[i++]=tmp_numbers_a[a++];
                }
                else if (tmp_numbers_a[a]<=tmp_numbers_b[b]) {
                    local_numbers[i++]=tmp_numbers_a[a++];
                }
                else
                    local_numbers[i++]=tmp_numbers_b[b++];
            }
        }
    }

    if (rank==0) {
        /* the root processor holds the final sorted numbers */
        numbers_save_fn( "numbers.data", local_numbers );
    }

    MPI_Finalize();
}
```

Appendix D

The All-Pairs Shortest Paths Problem

Contents

D.1 The HSM Skeleton Superstructure 204
D.2 The Pure HSM Solution 205
D.3 The HSM/Fork95++ Solution 209
D.4 The HSM/C++/MPI Solution 211
D.5 The Pure Fork95++ Solution 212
D.6 The Pure C++/MPI Solution 215

This appendix gives the full code listings of the all-pairs shortest paths solutions described in Chapter 10.

D.1 The HSM Skeleton Superstructure

```
/* $Id: shortestpaths.hsm,v 1.3 1997/07/11 00:18:47 marr Exp $
 *
 * All-Pairs Shortest Paths Algorithm - top level HSM code
 */


/*
 * Base level functions
 */

/* load the adjacency matrix */
basefn array<int> load_graph_fn(
    string filename
);
```

```
/* save the adjacency matrix */
basefn void save_graph_fn(
    string filename,
    array<int> graph
);


/* calculate the single-source shortest paths */
basefn array<int> single_source_fn(
    array<int> graph,     /* global argument */
    int vertex
);


/* join two arrays together */
array<int> arrayconcat_fn( void global, array<int> a, array<int> b )
{
    return( a++b );
}


/* the main routine */
void main(void)
{
    array<int> graph, vertices, weights;
    array<array<int>> adjacency_rows;

    /* load the graph */
    graph = load_graph_fn( "graph.data" );

    /* calculate the single-source problems */
    adjacency_rows = map( single_source_fn, graph,
                          vector_create_indexed( array_size(graph)[0] ) );

    /* concattenated the (2D) adjacency rows to form
     * the 2D array of weights */
    weights = reduce( arrayconcat_fn, NULL, adjacency_rows );

    /* save the sorted path weights */
    save_graph_fn( "weights.data", weights );
}
```

D.2 The Pure HSM Solution

```
/* $Id: shortestpaths_purehsm.hsm,v 1.11 1997/07/11 00:19:53 marr Exp $
 *
 * All-Pairs Shortest Paths Algorithm - top level HSM code
 */
```

```
/* declare a new vertex type to hold more information */
typedef struct {
    int index;
    int cost;
    bool used;
} vertex_t;


/* create a new structure to hold a vertex and a single edge cost */
typedef struct {
    int edge_cost;
    vertex_t vertex;
} vertex_edge_t;



/* convert an index and an adjacency matrix entry into a
 * vertex structure
 *      - used in the zip skeleton
 */
vertex_t costtovertex_fn(
    void global, int index, int cost
)
{
    vertex_t s;

    s.index = index;
    s.cost = cost;
    s.used = FALSE;
    return( s );
}



/* convert a vertex structure element into an adjacency matrix entry
 *      - used in the map skeleton
 */
int vertextocost_fn(
    void global, vertex_t v
)
{
    return( v.cost );
}



/* return the minimum of the two indexed costs
```

```
 *     - used to reduce the array of vertices to a single vertex
 */
vertex_t min_vertex_fn(
    void global, vertex_t a, vertex_t b
)
{
    return( if( b.used, a,
                if( a.used, b,
                    if(a.cost<=b.cost, a, b ) ) ) );
}



/* return true if more paths have still to be found
 *     - used in the while skeleton
 */
bool more_paths_fn(
    array<int> graph,
    array<vertex_t> work
)
{
    /* the global parameter, graph, is ignored */
    return( !(reduce( min_vertex_fn, NULL, work ).used) );
}



/* augmented_work_join_fn
 *     - used in add_vertex_fn to join a vertex with its
 *       potentially better edge cost
 */
vertex_edge_t augmented_work_join_fn(
    void global,
    int edge_cost,
    vertex_t vertex
)
{
    vertex_edge_t local;

    augmented_work.edge_cost = edge_cost;
    augmented_work.vertex = vertex;

    return( augmented_work );
}



/* update a vertex
 *     - used in the map skeleton of add_vertex_fn()
```

```
 */
vertex_t update_vertex_fn(
    vertex_t new_vertex,
    vertex_edge_t local
)
{
    /* Only one column of the adjacency matrix is actually used */
    vertex_t updated_vertex;

    updated_vertex = local.vertex;
    updated_vertex.cost = updated_vertex.cost + local.edge_cost;
    return( if( local.vertex.used || local.vertex.cost<=updated_vertex.cost,
                local.vertex, updated_vertex ) );
}



/* join a vertex with an edge cost */
vertex_edge_t local_work_join_fn(
    void global, int edge_cost, vertex_t vertex
)
{
    vertex_edge_t joined;

    joined.edge_cost = edge_cost;
    joined.vertex = vertex;

    return( joined );
}



/* find and add the next vertex */
array<vertex_t> add_vertex_fn(
    array<int> graph,          /* global parameter */
    array<vertex_t> work
)
{
    int n;
    vertex_t new_vertex;
    array<vertex_edge_t> augmented_work;
    array<vertex_t> new_work;

    /* find the minimum vertex, add it and update the minimum costs */
    new_vertex = reduce( min_vertex_fn, NULL, work );

    /* join each vertex with its potentially better edge cost */
    n = array_size(graph)[0];
    augmented_work = zip( augmented_work_join_fn, NULL,
```

```
                        graph[new_vertex.index][0..n-1], work );
    augmented_work[new_vertex.index].vertex.used = TRUE;

    /* update the work[*].cost entries */
    new_work = map( update_vertex_fn, new_vertex, augmented_work );

    return( new_work );
}


/* calculate the single-source shortest paths */
array<int> single_source_fn(
    array<int> graph,    /* global argument */
    int vertex
)
{
    /* this function must return a 2D array of integers */
    int n;
    array<vertex_t> work;
    array<int> weights;

    /* create the 'work' unit by converting each edge cost into
     * a vertex structure
     */
    n = array_size(graph)[0];
    work = zip( costtovertex_fn, NULL,
            vector_create_indexed( n ), graph[vertex][0..n-1] );
    work[vertex].used = TRUE;

    /* iterate untill all paths have been found */
    work = while( more_paths_fn, add_vertex_fn, graph, work );

    /* convert the work back into a single rowed adjacency matrix */
    weights = array_create( 2, 1, n );
    weights[0][0..n-1] = map( vertextocost_fn, NULL, work );

    return( weights );
}
```

D.3 The HSM/Fork95++ Solution

```
/* $Id: shortestpaths_fork.c,v 1.5 1997/07/10 15:50:22 marr Exp $
 *
 * All-Pairs Shortest Paths Algorithm - HSM/Fork95++ base language
 */
```

```
#include <stdlib.h>


sync HSM_array<int>& single_source_fn(
    HSM_array<int>& graph,      /* global argument */
    HSM_int& vertex
)
{
    pr int i;
    sh int *used, finished=0, mincost, minpos;
    HSM_array<int>& weights=*(shnew HSM_array<int>);

    /* declare the new arrays */
    weights.create( 2, 1, graph.size(0) );
    used = shnew int[graph.size(0)];
    for ( i=$; i<graph.size(0); i+=groupsize() ) {
        used[i]=0;
        weights(i)=graph(vertex,i);
    }

    while ( !finished ) {
        /* find the vertex with minimum cost */
        mincost=INT_MIN;    /* largest negative value */
        for ( i=$; i<graph.size(0); i+=groupsize() ) {
            if ( !used[i] ) {
                /* the current vertex hasn't been used yet */
                mpmax( &mincost, -weights(i) );
                if (mincost==-weights(i)) {
                    /* the current cost is minimal,
                     * so resolve arbitrarily */
                    minpos=i;
                }
            }
        }
        used[minpos]=TRUE;

        /* update the minimum cost matrix */
        for ( i=$; i<graph.size(0); i+=groupsize() ) {
            if (!used[i] && weights(i)>=weights(minpos)+graph(minpos,i)) {
                weights(i)=weights(minpos)+graph(minpos,i);
            }
        }

        /* see if we're finished yet */
        for ( i=$; i<graph.size(0); i+=groupsize() ) {
            mpand( &finished, used[i] );
        }
    }
```

```
    return( weights );
}
```

D.4 The HSM/C++/MPI Solution

```
/* $Id: shortestpaths_c.c,v 1.7 1997/07/10 15:49:26 marr Exp $
 *
 * All-Pairs Shortest Paths Algorithm - HSM/C++/MPI base language
 */


#include <mpi.h>
#include <limits.h>
#include "shortestpaths.hsm.c"



/* calculate the single-source shortest paths */
HSM_array<int>& single_source_fn(
    HSM_array<int>& graph,          /* global argument */
    HSM_int& start_vertex
)
{
    int vertex, n, size, iteration, valloc[2], minvalloc[2];
    HSM_array<int> used, mincost, adjmatrow;
    HSM_index ind;

    /* calculate the size of the local machine */
    MPI_Comm_size(HSM_Comm_world, &size);

    /* create the distributed 'used' and 'minimum cost' arrays */
    n = graph.size(0);
    used.create( 1, n, HSM_DISTRIBUTION_STANDARD );
    mincost.create( 1, n, HSM_DISTRIBUTION_STANDARD );
    adjmatrow.create( 1, n, HSM_DISTRIBUTION_STANDARD );

    /* broadcast the start vertex */
    if (start_vertex.is_local()) vertex = start_vertex;
    MPI_Bcast( &vertex, 1, MPI_INT,
               start_vertex.where_is(), HSM_Comm_world );

    /* initialise the 'used' array */
    for ( ind = used.first_local(); ind <= used.last_local();
          ind = used.next_local() ) {
        used.element(ind) = 0;
    }
    if ( used(vertex).is_local() ) used(vertex)=1;
```

```
    /* scatter the appropriate row of the adjacency matrix */
    MPI_Scatter( &graph( vertex, 0 ), n/size, MPI_INT,
                 &mincost.first_local(), n/size, MPI_INT,
                 graph( vertex, 0 ).where_is(), HSM_Comm_world );

    /* add the vertices one at a time until all the shortest path
     * costs have been calculated
     */
    for ( iteration=n-1; iteration>0; iteration-- ) {
        /* find the local minimum of unused vertices */
        valloc[0] = INT_MAX;
        for ( ind = used.first_local(); ind <= used.last_local();
              ind = used.next_local() ) {
            if (!used.element(ind) && mincost.element(ind)<valloc[0]) {
                valloc[0]=mincost.element(ind);
                valloc[1]=ind(0);
            }
        }

        /* find the global minimum of unused vertices */
        MPI_Allreduce( valloc, minvalloc, 1, MPI_2INT,
                       MPI_MINLOC, HSM_Comm_world );
        vertex = minvalloc[1];

        /* add the minimum vertex to V_T */
        if ( used(vertex).is_local() ) used(vertex)=1;

        /* scatter the appropriate row of the adjacency matrix */
        MPI_Scatter( &graph( vertex, 0 ), n/size, MPI_INT,
                     &adjmatrow.first_local(), n/size, MPI_INT,
                     graph( vertex, 0 ).where_is(), HSM_Comm_world );

        /* update the minimum cost matrix */
        for ( ind = used.first_local(); ind <= used.last_local();
              ind = used.next_local() ) {
            if (!used.element(ind)
                && mincost.element(ind)>mincost(vertex)+adjmatrow(ind)) {
                mincost.element(ind)=mincost(vertex)+adjmatrow(ind);
            }
        }
    }
}
```

D.5 The Pure Fork95++ Solution

```
/* $Id: shortestpaths_purefork.c,v 1.5 1997/07/10 16:20:41 marr Exp $
 *
 * All-Pairs Shortest Paths - Pure Fork95++ code
```

```
 */

#include <fork.h>

/*
 * a class for one dimensional arrays
 */
class array1d_t {
    sh int *data;
    sh int n;
public:
    sync void create( sh int nelems ) {
        n=nelems;
        data = shnew int[n];
    }
    async int nelems() {
        return n;
    }
    async int& operator[]( pr int i ) {
        return data[i];
    }
}

/*
 * a class for two dimensional arrays
 */
class array2d_t {
    sh int *data;
    sh int nrows, ncolumns;
public:
    sync void create( sh int rows, sh int cols ) {
        nrows=rows;
        ncolumns=cols;
        data = shnew int[nrows*ncolumns];
    }
    async int rows() {
        return nrows;
    }
    async int columns() {
        return ncolumns;
    }
    async int& operator[]( pr int i, pr int j ) {
        return data[i*rows+j];
    }
}

/*
 * the main function
 */
```

```
void main(void)
{
    pr int i;
    start {
        sh array2d_t graph, mincost, used;
        sh array1d_t minval, minpos;

        /* load the adjacency matrix */
        graph = load_graph_fn( "graph.data" );
        mincost.create( graph.rows(), graph.columns() );
        used.create( graph.rows(), graph.columns() );
        minval.create( graph.rows() );
        minpos.create( graph.rows() );

        /* solve the n single-source problems */
        fork ( graph.rows(); @=$%graph.rows(); $=$/graph.rows() ) {
            for ( i=$; i<graph.rows(); i+=groupsize() ) {
                used[@,i]=0;
                mincost[@,i]=graph[@,i];
            }
            used[@,@]=1;

            /* find the vertex with minimum cost */
            finished[@]=0;
            while ( !finished[@] ) {
                minval[@]=INT_MIN;
                for ( i=$; i<graph.rows(); i+=groupsize() ) {
                    if ( !used[@,i] ) {
                        mpmax( &minval[@], -mincost[@,i] );
                        if ( minval[@]==-mincost[@,i] ) {
                            minpos[@]=i;
                        }
                    }
                }
                used[@,minpos[@]]=1;

                /* update the minimum cost matrix */
                for ( i=$; i<graph.rows(); i+=groupsize() ) {
                    if ( !used[@,i]
                         && mincost[@,i]>=mincost[@,minpos[@]]
                                          +graph[minpos[@],i] ) {
                        mincost[@,i]=mincost[@,minpos[@]]
                                          +graph[minpos[@],i];
                    }
                }

                /* see if we're finished yet */
                for ( i=$; i<graph.rows(); i+=groupsize() ) {
                    mpand( &finished[@], used[@,i] );
```

```
                }
            }
        }
    }
    save_graph_fn("weights.data", graph);
}
```

D.6 The Pure C++/MPI Solution

```
/* $Id: shortestpaths_purec.c,v 1.7 1997/07/10 16:34:40 marr Exp $
 *
 * All-Pairs Shortest Paths Algorithm - pure C++/MPI solution
 */

#include <limits.h>
#include <assert.h>
#include <mpi.h>

class array1d_t {
    int *data;
    int n;
public:
    void create( int nelems ) {
        n=nelems;
        data = new int[n];
    }
    int nelems() {
        return n;
    }
    int& operator[]( int i ) {
        return data[i];
    }
}

class array2d_t {
    int *data;
    int nrows, ncolumns;
public:
    void create( int rows, int cols ) {
        nrows=rows;
        ncolumns=cols;
        data = new int[nrows*ncolumns];
    }
    int rows() {
        return nrows;
    }
    int columns() {
        return ncolumns;
```

```
    }
    int& operator[]( int i, int j ) {
        return data[i*rows+j];
    }
}

void main(int argc, char *argv[])
{
    int i, nvertices, rank, size, iteration;
    int local_group_index, local_rank, local_size, local_offset, local_width;
    int single_source, single_source_width;
    int single_source_offset, single_source_index;
    array1d_t mincost_array, vertex_used;
    array2d_t graph, graph_columnblock, allpairs_weights, singlesource_weights;
    MPI_Comm single_source_comm, single_source_leadercomm;
    int mpi_columnblock_blocklengths[2];
    MPI_Aint mpi_columnblock_displacements[2];
    MPI_Datatype mpi_columnblock_row_type, mpi_columnblock_type;
    MPI_Datatype mpi_columnblock_types[2];
    struct { int val, loc; } local_min_vertex, global_min_vertex;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );


    /* load the graph at the root processor */
    if (rank==0) {
        graph = load_graph_fn( "graph.data" );
        nvertices=graph.rows();
        assert( nvertices%size==0 );
    }
    MPI_Bcast( &nvertices, 1, MPI_INT, 0, MPI_COMM_WORLD );

    /* create new communicators for each of the groups and one for
     * the leaders of those groups */
    local_group_index = rank%nvertices;
    MPI_Comm_split( MPI_COMM_WORLD, local_group_index, rank,
                    &single_source_comm );
    MPI_Comm_rank( single_source_comm, &local_rank );
    MPI_Comm_size( single_source_comm, &local_size );
    MPI_Comm_split( MPI_COMM_WORLD, local_rank==0, rank,
                    &single_source_leadercomm );

    /* broadcast the graph from the root processor to the group leaders */
    if (local_rank==0) {
        if (rank>0) graph.create( nvertices, nvertices );
        MPI_Bcast( &graph[0,0], nvertices*nvertices, MPI_INT,
                   0, single_source_leadercomm );
```

```
}
else {
    graph.create( 0, 0 );
}

/* scatter the adjacency matrix using columnwise stripping over the
 * single source group */
assert( nvertices%local_size==0 );
local_width = nvertices/local_size;
local_offset = local_width*local_rank;

/* create a datatype for a column block of the matrix, each row
 * containing 'width' elements */
MPI_Type_vector( nvertices, local_width, nvertices, MPI_INT,
                 &mpi_columnblock_row_type );
MPI_Type_commit( &mpi_columnblock_row_type );

/* create a datatype for a whole column block of the matrix,
 * making sure that successive elements of this type are correctly
 * displaced */
mpi_columnblock_blocklengths[0] = 1;
mpi_columnblock_blocklengths[1] = 1;
mpi_columnblock_displacements[0] = 0;
mpi_columnblock_displacements[1] = local_width*sizeof(int);
mpi_columnblock_types[0] = mpi_columnblock_row_type;
mpi_columnblock_types[1] = MPI_UB;

MPI_Type_struct( 2, mpi_columnblock_blocklengths,
                 mpi_columnblock_displacements,
                 mpi_columnblock_types,
                 &mpi_columnblock_type );
MPI_Type_commit( &mpi_columnblock_type );

/* distribute the adjacency matrix */
graph_columnblock.create( nvertices, local_width );
MPI_Scatter( &graph[0,0], 1, mpi_columnblock_type,
             &graph_columnblock[0,0], nvertices*local_width, MPI_INT,
             0, single_source_comm );

/* Calculate the number of vertices allocated locally and their offset */
if (nvertices<size) {
    single_source_width = 1;
}
else {
    single_source_width = nvertices/size;
}
single_source_offset = (rank%nvertices)*single_source_width;

/* Create the distributed minimum cost array and vertex set */
```

```
mincost_array.create( local_width );
vertex_used.create( graph_columnblock.columns() );

/* create the single-source shortest path cost array */
if (local_rank==0) {
    singlesource_weights.create( single_source_width, nvertices );
}
else {    /* dummy array */
    singlesource_weights.create( 0, 0 );
}

/* create the all-pairs shortest path cost array */
if (rank==0) {
    allpairs_weights.create( nvertices, nvertices );
}
else {
    allpairs_weights.create( 0, 0 );
}


/*
 * loop through each single_source allocated locally
 */
for (single_source_index = 0; single_source_index < single_source_width;
     single_source_index++) {

    single_source = single_source_offset + single_source_index;
    for (i=0; i<mincost_array.nelems(); i++) {
        mincost_array[i] = graph_columnblock[single_source,i];
    }
    for (i=0; i<vertex_used.nelems(); i++) {
        vertex_used[i] = 0;
    }

    /* iterate, adding one vertex each time */
    global_min_vertex.loc = 0;
    for (iteration=0; iteration<nvertices &&global_min_vertex.loc>=0;
         iteration++) {

        /* calculate the local minimum value */
        local_min_vertex.val = INT_MAX;
        for (i=0; i<mincost_array.nelems(); i++) {
            if ( !vertex_used[i]
                 && local_min_vertex.val> mincost_array[i]) {
                local_min_vertex.val = mincost_array[i];
                local_min_vertex.loc = local_offset+i;
            }
        }
```

```
            /* find the global minimum and broadcast to the others */
            MPI_Allreduce( &local_min_vertex, &global_min_vertex, 1, MPI_2INT,
                           MPI_MINLOC, single_source_comm );

            /* mark the vertex as used */
            if (global_min_vertex.loc == local_min_vertex.loc) {
                vertex_used[global_min_vertex.loc-local_offset] = 1;
            }

            /* update the cost matrix */
            for (i=0; i<mincost_array.nelems(); i++) {
                if ( !vertex_used[i] && mincost_array[i]
                         > global_min_vertex.val
                           + graph_columnblock[global_min_vertex.loc,i] ) {
                    mincost_array[i] = global_min_vertex.val
                      + graph_columnblock[global_min_vertex.loc,i];
                }
            }
        }

        /*
         * Gather the vector to the single source leader
         */
        MPI_Gather( &mincost_array[0], local_width, MPI_INT,
                    &singlesource_weights[single_source_index,0],
                    local_width, MPI_INT, 0, single_source_comm );
    }

    /* gather the single source solutions */
    if (local_rank==0) {
        MPI_Gather( &singlesource_weights[0,0],
                    single_source_width*nvertices, MPI_INT,
                    &allpairs_weights[0,0],
                    single_source_width*nvertices, MPI_INT,
                    0, single_source_leadercomm );
    }

    /* Save the shortest path weights */
    if (rank==0) {
        save_graph_fn( "weights.data", allpairs_weights ) ;
    }

    MPI_Finalize();
}
```

Bibliography

[Agh90] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[Akl89] Selim G Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall International Editions, 1989.

[AKP91a] F Abolhassan, J Keller, and W J Paul. On the cost-effectiveness of PRAMs. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 2–9. IEEE; ACM, IEEE Comput. Soc., December 1991.

[AKP91b] Ferri Abolhassan, Jörg Keller, and Wolfgnag J Paul. On the cost-effectiveness and realisation of the theoretical PRAM model. SFB 124-C1 Report 09/1991, Universität des Saarlandes, W-6600 Saarbrücken, Germany, 1991.

[AS91] Richard J Anderson and Lawrence Snyder. A comparison of shared and nonshared memory models of parallel computation. *Proceedings of the IEEE*, 29(4):480–487, April 1991.

[Bac78] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bel92] Trudy E Bell. Beyond today's supercomputers. *IEEE Spectrum*, 29:72–75, September 1992.

[BK95] George H. Botorog and Herbert Kuchen. Algorithmic skeletons for adaptive multigrid methods. In A. Ferreira and J. Rolim, editors, *Proceedings of Irregular '95*, volume 980 of *LNCS*, pages 27–41. Springer, 1995.

[BK96a] George H. Botorog and Herbert Kuchen. Efficient parallel programming with algorithmic skeletons. In L. Bouge, P. Fraigniaud,

A. Mignotte, and Y. Robert, editors, *Proceedings of EuroPar '96*, volume 1123 of *LNCS*, pages 718–731. Springer, 1996.

[BK96b] George H. Botorog and Herbert Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society, 1996.

[BK96c] George H. Botorog and Herbert Kuchen. Using algorithmic skeletons with dynamic data structures. In A. Ferreira, J. Rolim, Y. Saad, and T. Yang, editors, *Proceedings of Irregular '96*, volume 1117 of *LNCS*, pages 263–276. Springer, 1996.

[BKG+93] Helmar Burkhart, Carlos Falso Korn, Stephan Gutzwiller, Peter Ohnacker, and Stephan Waser. BACS: Basel algorithm classification scheme. Technical Report 93-3, University of Basel, Switzerland, March 1993.

[Ble87] G.E. Blelloch. Scans as primitive parallel operations. In *Proc. of the International Conference on Parallel Processing*, pages pp. 355–362, August 1987.

[Ble95] Guy E. Blelloch. Nesl: A nested data-parallel language. Technical Report CMU-CS-95-170, Carnegie Mellon University, September 1995.

[BM93] R H Bisseling and W F McColl. Scientific computing on bulk synchronous parallel architectures. In *General Purpose Parallel Computing* [Bri93].

[Bra95] Tore Andreas Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, November 1995.

[Bra92] T.A. Bratvold. Determining useful parallelism in higher order functions. In *Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages*, 92.

[Bri93] The British Computer Society. *General Purpose Parallel Computing*, Parallel Processing Specialist Group, University of Westminster, London, December 1993.

[Cam94] Duncan K. G. Campbell. *CLUMPS: a candidate model of efficient, general purpose parallel computation.* PhD thesis, Department of Computer Science, University of Exeter, October 1994.

[Cam96] Duncan K. G. Campbell. Further results with algorithmic skeletons for the clumps model of parallel computation. Technical Report YCS 275, Department of Computer Science, University of York, December 1996.

[CFSV94] Thomas Cheatham, Amr Fahmy, Dan C. Stefanescu, and Leslie G. Valiant. Bulk synchronous parallel computing–a paradigm for transportable software. Technical Report TR-36-94, Harvard University, Computer Science Department, 1994.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[Col91] M. Cole. Graph reduction and pipelined parallel algorithms. technical report CSR-18-91, Computer Science Dept, Edinburgh University, 1991.

[Col93a] Murray Cole. List homomorphic parallel algorithms for bracket matching. Technical Report CSR-29-93, Department of Computer Science, University of Edinburgh, 1993.

[Col93b] Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. Technical Report CSR-25-93, Department of Computer Science, University of Edinburgh, 1993.

[Col94] Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In G.R. Joubert, editor, *Parallel Computing: Trends and Applications*, pages 489–492. North-Holland, 1994.

[Col97] Murray Cole. Dividing and conquering. Technical Report ECS-CSG-31-97, Department of Computer Science, University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, March 1997.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Parallel Architectures and Languages Europe*, volume 694 of *LNCS*, pages 146–160. Springer-Verlag, June 1993.

[DGTJ95] J. Darlington, Y.K Guo, H.W. To, and Y. Jing. Skeletons for structured parallel composition. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[DMO+92] Marco Danelutto, Roberto Di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220, 1992.

[Dob94] Simon Dobson. An introduction to the theories of bulk data types. Technical Report TallShiPP/R/01, Informatics Department, Rutherford Appleton Laboratory, July 1994.

[DP93a] M Danelutto and S Pelagatti. Structuring parallelism in a functional framework. Technical Report TR-29/93, Dipartimento Di Informatica, Universitá Di Pisa, Corso Italia, 4056125 Pisa, Italy, November 1993.

[DP93b] Marco Danelutto and Susanna Pelagatti. Structuring parallelism in a functional framework. Technical Report TR-29/93, University of Pisa, Dept. of Computer Science, 1993.

[DS92] R Drefenstedt and D Schmidt. On the physical design of butterfly networks for PRAMs. In H J Siegel, editor, *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 202–209. IEEE, IEEE Comput. Soc., October 1992.

[For94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 5 1994.

[FS92] Yosee Feldman and Ehud Shapiro. Spatial machines: A more realistic approach to parallel computation. *Communications of the ACM*, 35(10):61–73, October 1992.

[GP83] Zvi Galil and Wolfgang J Paul. An efficient general purpose parallel computer. *Journal of the Association for Computing Machinery*, 30(2):360–387, April 1983.

[GS92] A M Gibbons and P Spirakis, editors. *Lectures on Parallel Computation. Proceedings 1991 ALCOM Spring School on Parallel Computation.* Cambridge International Series on Parallel Computation. Cambridge University Press, 1992.

[GV94] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[Han72] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[Han75] Per Brinch Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[HGLS86] W Daniel Hillis and Jr Guy L Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[HHMW94] Rolf Hempel, Anthony J G Hey, Oliver McBryan, and David W Walker. Guest editorial: Message passing interfaces. *Parallel Computing*, 20:415–416, 1994.

[HL93] Todd Heywood and Claudia Leopold. Models of parallelism. Technical Report CSR-28-93, Department of Computer Science, University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, July 1993.

[HMS+97] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. Bsplib: The bsp programming library. Technical Report May, Oxford University Computing Laboratory, 1997.

[Hoa78] C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[How96] Fred Howell. *Approaches to Parallel Performance Prediction.* PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[HR92a] Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation: Part i. the model. *Journal of Parallel and Distributed Computing*, 16:212–232, 1992.

[HR92b] Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation: Part ii. binary tree and fft algorithms. *Journal of Parallel and Distributed Computing*, 16:233–249, 1992.

[HSS90] T Hagerup, A Schmitt, and H Seidl. FORK: A high level language for PRAMs. SFB 124 Report 22/1990, Universität des Saarlandes, W-6600 Saarbrücken, Germany, December 1990.

[HSS92] T Hagerup, A Schmitt, and H Seidl. FORK: A high-level language for PRAMs. *Future Generation Computer Systems*, 8:379–393, 1992.

[HT93] W Daniel Hillis and Lewis W Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[ICC+96] R. Ibbett, G. Chochia, P.S. Coe, M.I. Cole, P.E. Heywood, T. Heywood, R.J. Pooley, P. Thanisch, and N.P. Topham. Algorithms, Architectures and Models of Computation. technical report ECS-CSG-22-96, Dept. of Computer Science, Edinburgh University, March 1996.

[Jor86] Harry F Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3:93–110, 1986.

[JQ93] C R Jesshope and M Abdul Qadar. BSPC and the n-computer. In *General Purpose Parallel Computing* [Bri93], pages 24–43.

[JS91] Guido K Jouret and David Sharp. A skeletons primer. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, London SW7 2BZ, May 1991.

[JW94] Joseph JáJá and Pearl Y Wang. Special issue on data parallel algorithms and programming. *Journal of Parallel and Distributed Computing*, 21:1–3, 1994.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc, 1994.

[KR90] Richard M Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 17, pages 871–941. Elsevier Science Publishers B.V., 1990.

[KS95a] C W Kessler and H Seidl. Fork95 language and compiler for the SB-PRAM. In *5th International Workshop on Compilers for Parallel Computers*, June 28-30 1995.

[KS95b] C. W. Kessler and H. Seidl. Integrating synchronous and asynchronous paradigms: the fork95 parallel programming language. Technical Report 95-05, Fachbereich 4 Informatik, Univ. Trier, 1995.

[KS97] C. W. Kessler and H. Seidl. The fork95 parallel programming language: Design, implementation, application. *International Journal on Parallel Programming*, 25(1):pp17–50, 1997.

[KT96] C. W. Kessler and J. Traff. A library of basic pram algorithms in fork. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[KWW93] N Kalantery, S C Winter, and D R Wilson. From BSP to a virtual von neumann machine. In *General Purpose Parallel Computing* [Bri93], pages 92–99.

[LF80] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[MC96] Marcus Marr and Murray Cole. Hierarchical skeletons and ad hoc parallelism. In E. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing: State-of-the-Art and Perspectives*, volume 11 of *Advances in Parallel Computing*, pages 673–676. Elsevier, 1996.

[McB94] Oliver A McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–444, 1994.

[McC92] W F McColl. General purpose parallel computing. In Gibbons and Spirakis [GS92].

[McC93a] W F McColl. An architecture independent programming model for scalable parallel computing. In *General Purpose Parallel Computing* [Bri93], pages 1–17.

[McC93b] W. F. McColl. Gl: An architecture independent programming language for scalable parallel computing. Technical Report 93-072-3-9025-1, NECI, June 1993.

[McC94] W. F. McColl. The bsp approach to architecture independent parallel programming. Technical report, Oxford University Computing Laboratory, December 1994.

[McC94] W F McColl. Bsp programming. In G Blelloch, M Chandy, and S Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, May 94. American Mathematical Society.

[MD92] J Magee and N Dulay. Mp: A programming environment for multicomputers. In *Programming environments for parallel computing*, 1992.

[MDK92] J Magee, N Dulay, and J Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[Mil93] Richard Miller. A library for bulk-synchronous parallel programming. In *General Purpose Parallel Computing* [Bri93], pages 100–108.

[Mis93] Jayadev Misra. Powerlist: A structure for parallel recursion. Got paper from Todd., May 1993.

[MR94] Dan C Marinescu and John R Rice. On high level characterization of parallelism. *Journal of Parallel and Distributed Computing*, 20:107–113, 1994.

[NDD93] J M Nash, P M Dew, and M E Dyer. Scalable parallel application design. In *General Purpose Parallel Computing* [Bri93], pages 44–79.

[Pel93] Susanna Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs*. PhD thesis, TD-11/93, Department of Computer Science, University of Pisa (Italy), 1993.

[Pre92] Roger S Pressman. *Software Engineering: A Practitioner's Approach.* McGraw Hill, 1992.

[PY88] Christos H Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 510–513. ACM Press, May 1988.

[Rab93] Fethi A. Rabhi. Exploiting parallelism in functional languages: a paradigm oriented approach. In T. Lake and P. Dew, editors, *Workshop on Abstract Machine Models for Highly Parallel Computers*, Leeds, 1993.

[Ran95] Roopa Rangaswami. HOPP - a Higher Order Parallel Programming Model. In Marc Moonen, editor, *Algorithms and Parallel VLSI Architectures.* Elsevier, 1995.

[RS92] Gudula Rünger and Kurt Sieber. A trace-based denotational semantics for the PRAM-language FORK. SFB 124-C1 01/1992, Universität des Saarlandes, W-6600 Saarbrücken, Germany, January 1992.

[RS93] F A Rabhi and J Schwarz. "paradigm-oriented" design of parallel iterative programs using functional languages. Draft article to appear in 'applications of supercomputers in engineering', bath (uk), september 1993, Department of Computer Science, University of Hull, Hull HU6 7RX, 1993.

[Sch91] A Schmitt. A formal semantics of FORK. SFB 124-C1 Report 11/1991, Universität des Saarlandes, W-6600 Saarbrücken, Germany, May 1991.

[SDB93] Anthony Skjellum, Nathan E Doss, and Purushotham V Bangalore. Writing libraries in MPI. In Anthony Skjellum and Donna S Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference.* IEEE Computer Society Press, October 1993.

[SDV94] Anthony Skjellum, Nathan E Doss, and Kishore Viswanathan. Intercommunicator extentions to MPI in the MPIX (MPI eXtention library). ftp://aurora.cs.msstate.edu:/pub/reports/mpix-icae.ps.Z, July 1994.

[Sei93] Helmut Seidl. Equality of instances of variables in FORK. SFB 124-C1 Report 06/1993, Universität des Saarlandes, W-6600 Saarbrücken, Germany, April 1993.

[SGDM94] V S Sunderam, G A Geist, J Dongarra, and R Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20:531–545, 1994.

[SHM96] David B. Skillicorn, Jonathan M.D. Hill, and W.F. McColl. Questions and answers about bsp. Technical Report 96-15, Oxford University Computing Laboratory, November 1996.

[Ski90] David B Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, pages 38–50, December 1990.

[Ski91] D B Skillicorn. Practical parallel computation: Part I. models of computation. Technical Report TR-91-313, Department of Computing and Information Science, Queen's University, Kingston, Canada, August 1991.

[Ski92] D B Skillicorn. Parallelism and the bird-meertens formalism. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Canada, April 1992.

[Ski94] David B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[Ski96a] D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39:115–125, 1996.

[Ski96b] David B. Skillicorn. minibsp: A bsp language and transformation system. Found in SEL-HPC archive, November 1996.

[ST96] David B. Skillicorn and D. Talia. Models and languages for parallel computation. Found in SEL-HPC archive, 1996.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Wal94] David W Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.

[Weg89] Peter Wegner. Guest editor's introduction to programming language paradigms. *ACM Computing Surveys*, 21(3):253–258, September 1989.

[WIS94] Gregory V Wilson, R Bruce Irvin, and Anil Sukul. Assessing and comparing the usability of parallel programming systems. FTP: ftp.cs.vu.nl:pub/gvw/toys/toys.ps, August 1994.