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.

# 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
Chapter 5  A Shared Memory Target Architecture for the HSM  
5.1 The HSM Node Hierarchy  
5.2 The HSM Data Structures  
5.3 The HSM/Fork95++ Interface  
5.4 A HSM/Fork95++ Example  

Chapter 6  A Distributed Memory Target Architecture for the HSM  
6.1 The HSM Node Hierarchy  
6.2 The HSM Data Structures  
6.2.1 The HSM Scalar Types  
6.2.2 The HSM Array Type  
6.2.3 The HSM Compound Structure Type  
6.2.4 Data Structure Distribution after Assignment  
6.3 The HSM/C++/MPI Interface  
6.3.1 The HSM Scalar Types  
6.3.2 The HSM Array Type  
6.3.3 The HSM Index Type  
6.3.4 The HSM Compound Structure Types  
6.4 A HSM/C++/MPI Example  

II Evaluation  

Chapter 7  Evaluation of the HSM Model  
7.1 The Problems
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

4
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:

```fortran
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:

```fortran
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 $i$th largest physical processor ID receives the (private) value $s_0 + \sum_{j=0}^{i-1} e_j$ 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 docu-
ment [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 de-
pendent 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 ab-
straction 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. Unfor-
nately, the BSP model does not allow for subsets of processors to be synchron-
ised 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, broadcast, applybroadcast, send, and fetch. Lastly, the computational skeletons abstract commonly used parallel computational patterns 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 into 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 $P^2L$ 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 chapter 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 \texttt{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 \textit{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\footnote{An instance of an algorithmic skeleton is simply expressed as a function call with the same name.}. The non-terminals \textit{type} and \textit{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:

\begin{verbatim}
int add_fn(int x, int y) {
    return(x+y);
}
\end{verbatim}

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

\begin{verbatim}
  hsm-program:
    type-declarations base-function-prototypes function-prototypes functions
\end{verbatim}
**type-declarations:**
/* nothing */ | type-declaration type-declarations

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

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

**functions:**
/* nothing */ | function functions

For example:
```c
typedef array < int > bag;
baseln sum_fn( bag x );
int main( array < string > argv );
```

A HSM function prototype can be described as follows:

**function-prototype:**
```c
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 `baseln`:

**base-function-prototypes:**
```c
'saseln' type id '(' 'void' ')' ';'
| 'baseln' 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.:
```c
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 acknowledg-
ledges 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

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Data Structure Declaration and Creation</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>The HSM Scalar Types</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>The HSM Array Type</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>The HSM Struct Type</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>Defining New Types</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>Other Data Types</td>
<td>37</td>
</tr>
</tbody>
</table>

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.

\[
\text{var-declaration:}\]
\[
\quad \text{type id-list } \cdot \cdot ;
\]

\[
\text{id-list:}
\]
\[
\quad id | id \cdot , id-list
\]

For example:

\[
\text{int } x, y; \quad \text{array< int > } m;
\]

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, multidimensional 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:

\[ \text{type:} \]
\[ \text{‘void’ | ‘fnnname’} \]

The \textit{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 \textit{fnnname} 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 \texttt{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.

\[ \text{type:} \]
\[ \text{‘bool’ | ‘int’ | ‘real’ | ‘string’} \]

For example:

\begin{verbatim}
bool b;
int i, j, k;
real x, y, z;
string s;
\end{verbatim}

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: \textit{bool}, \textit{int}, \textit{real} and \textit{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 \textit{bool-exp}, \textit{int-exp}, \textit{real-exp} and \textit{string-exp} to indicate an expression of a specific type. The term \textit{num-exp} is also used to indicate the use of either an \textit{int-exp} or \textit{real-exp}.

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:

\begin{verbatim}
expr:
  variable
  | id \texttt{(' \textquotesingle \textquotesingle} \texttt{' \textquotesingle \textquotesingle)}
  | id \texttt{(' \textquotesingle expr-list \textquotesingle \textquotesingle)}
  | \texttt{(' \textquotesingle expr \textquotesingle \textquotesingle)}
\end{verbatim}

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

\begin{verbatim}
expr:
  \texttt{\textquotesingle TRUE\textquotesingle \textquotesingle | \textquotesingle FALSE\textquotesingle}
  | \texttt{\textquotesingle \! \textquotesingle} bool-exp
  | bool-exp \texttt{\textquotesingle || \textquotesingle} bool-exp
  | bool-exp \texttt{\textquotesingle \&\& \textquotesingle} bool-exp
  | expr \texttt{\textquotesingle \&\& \textquotesingle} expr
  | expr \texttt{\textquotesingle !== \textquotesingle} expr
  | expr \texttt{\textquotesingle != \textquotesingle} expr
  | num-exp \texttt{\textquotesingle < \textquotesingle} num-exp
  | num-exp \texttt{\textquotesingle <= \textquotesingle} num-exp
  | num-exp \texttt{\textquotesingle > \textquotesingle} num-exp
  | num-exp \texttt{\textquotesingle >= \textquotesingle} num-exp
\end{verbatim}

Integer and real constants are composed as expected:

\begin{verbatim}
int-const:
  \texttt{[0-9][0-9]*}
\end{verbatim}

\begin{verbatim}
real-const:
  int-const \texttt{\textquotesingle .\textquotesingle} int-const
\end{verbatim}

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

\begin{verbatim}
expr:
  int-const
\end{verbatim}
| real-const
| num-expr * num-expr
| num-expr / num-expr
| '(' 'int' ')' real-expr
| '(' 'real' ')' int-expr

For example:

\[
\begin{align*}
b &= (i <= j); \\
i &= j * k; \\
x &= (real)(i * j * k); \\
s &= "Hello World";
\end{align*}
\]

3.3 The HSM Array Type

A multi-dimensional array is a structure of several dimensions, say \( d \) dimensions, where each dimension \( D_i \) 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 \( D_i \). 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.

\[
\text{type:}
\]
\[
\text{array}'<\text{type}>'
\]

For example:

\[
\begin{align*}
\text{array}<\text{int}>a; \\
\text{array}<\text{array}<\text{real}>\>r;
\end{align*}
\]

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 \textit{variable}:

\textit{variable:}

\textit{variable ref-operator}

\textit{ref-operator:}

\[ [' \text{int-expr} '] \]

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 \textit{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 \textit{ref-operator} to describe the range operator:

\textit{ref-operator:}

\[ [' \text{int-expr} ' \ldots ' \text{int-expr} '] \]

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 \( b \):

\begin{verbatim}
array < int > c;
c = a[0..1][0..1][0..1];
b[2..3][2..3][2..3] = c;
\end{verbatim}

i.e.:

\[
c_{<t_1,t_2,t_3>} = a_{<t_1,t_2,t_3>} ; \quad \left\{ 0 \leq t_1, t_2, t_3 < 2 \right\}
\]

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 \( a \):

```cpp
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 topmost dimension, i.e. \( D_1 \). Both arrays must have the same number of dimensions and the size of all the dimensions except \( D_1 \) must be the same for both arrays.

```cpp
expr:
array-expr `++` array-expr
```

This has the effect of performing the following assignments:

- \( c_{<t_1,t_2,\ldots,t_d>} = a_{<t_1,t_2,\ldots,t_d>} \quad 0 \leq t_1 < a_{n_1}, \quad \forall t_2, \ldots, t_d \)
- \( c_{<t_1,t_2,\ldots,t_d>} = b_{<t_1-a_{n_1},t_2,\ldots,t_d>} \quad a_{n_1} \leq t_1 < a_{n_1} + b_{n_1}, \quad \forall t_2, \ldots, t_d \)

where \( a_{n_1} \) and \( b_{n_1} \) are the sizes of dimension \( D_1 \) for arrays \( a \) and \( b \) respectively and \( c \) has the following dimension sizes:

\[
c_{n_1} = a_{n_1} + b_{n_1} \\
c_{n_i} = a_{n_i} = b_{n_i} \quad 2 \leq i \leq d
\]
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.

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

i.e.

\[
c_{t_1, t_2, t_3}^{c} = a_{t_1, t_2, t_3}^{a} \quad 0 \leq t_1 < 10, \quad 0 \leq t_2 < 20, \quad 0 \leq t_3 < 5
c_{t_1, t_2, t_3}^{c} = b_{t_1-10, t_2, t_3}^{b} \quad 10 \leq t_1 < 25, \quad 0 \leq t_2 < 20, \quad 0 \leq t_3 < 5
\]

### 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:

```c
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`:

```c
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:

\[
x = \text{array\_size(array\_size(a))}[0];
\]

### 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()`.

\[
\text{expr:}
\text{`array_create_init(` int\_expr `;`, `int\_expr\_list `;`, `expr `')'}
\]

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.:

\[
\text{array< array< real >> r;}
\text{r = array_create_init( 1, 10, array_create( 2, 5, 5 ) );}
\]

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:

\[
\text{r = array_create_init( 1, 5, array_create( 1, 5 ) )}
\text{++ array_create_init( 1, 5, array_create( 2, 5, 5 ) );}
\]

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()`:

\[
\text{expr:}
\text{`array_create_indexed(` int\_expr `;`, `int\_expr\_list `')'}
\]

34
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:

```c
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:

```c
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*.

```c
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 \texttt{struct{}}, similar to the \texttt{struct} used in ‘C’. This allows several different data structures to be grouped together under a single type:

type:

‘struct’ ‘{’ var-declarations ‘}’

For example:

\texttt{struct} \{ \texttt{int} x, y; \texttt{real} z; \texttt{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:

‘.’ \texttt{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.:

```c
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

Skeletions 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.:

\[ z = \texttt{map} ( \texttt{body} \_\texttt{fn}, x, y ) ; \]

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

\[
\texttt{array< type-\beta > map (}
\texttt{ fnname body}\_\texttt{fn, type-\gamma global, array< type-\alpha > local }
\texttt{)} ;
\]

where type-\alpha, type-\beta and type-\gamma 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.:

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

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 $t$. 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.

---

1The 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:

```plaintext
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:

\[
\text{if}(t, x, y) = \begin{cases} 
   x & \text{iff } t = \text{TRUE} \\
   y & \text{iff } t = \text{FALSE} 
\end{cases}
\]

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

```plaintext
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 $\gamma$ and a local data structure of type $\alpha$. Both the named functions take two arguments, of types $\gamma$ and $\alpha$ respectively. The first function returns a value of type `bool`, while the second returns a value of type $\alpha$. The skeleton returns a single data structure of type $\alpha$.

\[
\begin{align*}
\text{type-}\alpha & \quad \text{while} ( \\
\quad \text{fname} \ test\_fn, \ \text{fname} \ body\_fn, \\
\quad \text{type-}\gamma \ \text{global,} \ \text{type-}\alpha \ \text{local} \\
) ; \\
\text{bool} \ test\_fn ( \\
\quad \text{type-}\gamma \ \text{global,} \ \text{type-}\alpha \ \text{local} \\
) ; \\
\text{type-}\alpha \ \text{body}\_fn ( \\
\quad \text{type-}\gamma \ \text{global,} \ \text{type-}\alpha \ \text{local} \\
) ;
\end{align*}
\]

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 $\alpha$. 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.

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

The following example will call the `body-fn` exactly 10 times, and return the value ‘10’:

```c
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 );
```

43
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-fns.

### 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 \( \gamma \), and an array of local elements of type \( \alpha \). The named function takes two arguments, of types \( \gamma \) and \( \alpha \) respectively, and return a value of type \( \beta \). The skeleton returns an array of type \( \beta \) with the same number of dimensions and size as the input local array.

\[
\text{array}<\text{type-}\beta> \text{ map } ( \\
\text{ fnname body_fn, type-}\gamma\text{ global, array}<\text{type-}\alpha> \text{ local } \\
); \\
\text{type-}\beta \text{ body_fn ( } \\
\text{ type-}\gamma\text{ global, type-}\alpha\text{ local } \\
); \\
nonumber
\]

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

\[
\text{map} (b, g, l)_t = b( g, l_t) \quad \forall t
\]

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:

\[
\text{int body_fn( void global, array< int > tuple ) } \\
\text{ return ( tuple[0]*tuple[1] ); } \\
\]

\[
\text{array< int > x; } \\
\text{array< int >> ind; } \\
\text{ind = array_create_indexed( 2, 10, 10 ); } \\
\text{x = map ( body_fn, NULL, ind ); }
\]

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

\footnote{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 \( \alpha \) elements that it has been assigned and create
the corresponding type \( \beta \) 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 \( \alpha \) from the local array and execute
the body-fn over the local node to create one new element of type \( \beta \).

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 \( \beta \)
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 \( \gamma \), and two arrays of local data structures of
types \( \alpha \) and \( \beta \) respectively. The named function takes three arguments, of types
\( \gamma \), \( \alpha \) and \( \beta \) respectively, and returns a value of type \( \delta \). The skeleton returns an
array of type \( \delta \) with the same number of dimensions and size as the local input
arrays.

\[
\text{array}<\text{type-}\delta> \text{ zip (}
\quad \text{fname body_fn, type-}\gamma \text{ global,}
\quad \text{array}<\text{type-}\alpha> \text{ local}_a, \quad \text{array}<\text{type-}\beta> \text{ local}_b
\)
\];

\[
\text{type-}\delta \text{ body_fn (}
\quad \text{type-}\gamma \text{ global, type-}\alpha \text{ local}_a, \quad \text{type-}\beta \text{ 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:
\[
\text{zip}(b, g, l_a, l_b), = b(g, l_{a_t}, l_{b_t}) \quad \forall t
\]

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

```c
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 $\gamma$, and an array of local elements of type $\alpha$.
The named function takes three arguments, of types $\gamma$, $\alpha$ and $\alpha$ respectively, and
returns a single value of type $\alpha$. The skeleton returns a single value of type $\alpha$.

```c
type-\alpha \text{ reduce} (  
    \text{fname} body\_fn, type-\gamma \text{ global}, \text{array}< type-\alpha > \text{ local}  
);

type-\alpha \text{ body}\_fn (  
    type-\gamma \text{ global}, type-\alpha \text{ local}_a, type-\alpha \text{ 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 \((\text{array}<1>)x\) which creates a one-dimensional array of arrays as explained in \S 3.3.3.7.

\[
\text{reduce}(b, g, l) = \begin{cases} 
\text{reduce}(b, g, (\text{array}<1>)l) & \text{if } ||l|| > 1 \\
 b\left(g, \text{reduce}(b, g, l_{<o>}, [\frac{|l|}{2}]_{-1}), \text{reduce}(b, g, l_{[\frac{|l|}{2}]_{-1}})) \right) & \text{if } ||l|| = 1 \quad \text{and} \quad |l| > 1 \\
 l_{<o>} & \text{if } ||l|| = 1 \quad \text{and} \quad |l| = 1 
\end{cases}
\]

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

```c
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 \(\alpha\) 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 $\gamma$, and a local element of type $\alpha$. The named functions take two arguments, of types $\gamma$ and $\alpha$ respectively. The first function returns a value of type $\text{bool}$, and the second an array of values of type $\alpha$. The skeleton returns a one-dimensional array of type $\alpha$ whose size is dependent on the number of divisions that were carried out.

```plaintext
array< type-\alpha > divide (  
    fnname test_fn,  fnname divide_fn,  
    type-\gamma global,  type-\alpha local
);
bool test_fn (  
    type-\gamma global,  type-\alpha local
);
array< type-\alpha > divide_fn (  
    type-\gamma global,  type-\alpha local
);
```

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

\[
\text{divide}(t, b, g, l) = \begin{cases} 
\text{reduce} \left( \text{++}, \ \text{NULL}, \ \text{map}(\text{map-fn}, g, b(g,l)) \right) \\
\quad \text{iff} \quad t(g, l) = \text{TRUE} \\
\text{array_create_init}(1, 1, l) \\
\quad \text{iff} \quad t(g, l) = \text{FALSE}
\end{cases}
\]

\[
\text{map-fn}(g, l) = \text{divide}(t, b, g, l)
\]

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.

\[
\begin{array}{l}
\text{bool test_fn( int a, int b )} \\
\quad \text{return ( b > a );}
\end{array}
\]

\[
\begin{array}{l}
\text{array< int > test_fn( int a, int b )} \\
\quad \text{array< int > ret;}
\quad \text{ret = array_create( 1, 3 );}
\quad \text{ret[0] = b/2;}
\quad \text{ret[1] = (b - ret[0])/2;}
\quad \text{ret[2] = b - (ret[0] + ret[1]);}
\quad \text{return ( ret );}
\end{array}
\]

\[
\begin{array}{l}
\text{array< int > z;}
\quad \text{z = divide( test_fn, divide_fn, 16, 65535 );}
\end{array}
\]

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 \(\gamma\), and an array of local elements of type \(\alpha\). The function named takes three arguments, of of types \(\gamma\), \(\alpha\) and \(\alpha\) respectively, and returns a single value of type \(\alpha\). The skeleton returns an array of values of type \(\alpha\) with the same number of dimensions and size as the input local array.

\[
\text{array}<\text{type-}\alpha>\;\text{scan}\;(
\text{fncname}\;\text{body_fn, type-}\gamma\;\text{global, array}<\text{type-}\alpha>\;\text{local})
\]

\[
\text{type-}\alpha\;\text{body_fn (}
\text{type-}\gamma\;\text{global, type-}\alpha\;\text{local_a, type-}\alpha\;\text{local_b})
\]

The result of the \text{scan} skeleton can be defined as follows:

\[
\text{scan}(b, g, l)_i = \text{reduce}(b, g, l_{0:i}) \quad \forall i
\]

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

```c
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 \text{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 \(\gamma\) and a local array of type \(\alpha\) as arguments. The function named takes two arguments, of types \(\gamma\) and \(\alpha\) respectively, and returns a value of type \text{bool}. The skeleton returns a one-dimensional array of type \(\alpha\).

\[
\text{array}<\text{type-}\alpha>\;\text{filter}\;(
\text{fncname}\;\text{test_fn, type-}\gamma\;\text{global, array}<\text{type-}\alpha>\;\text{local})
\]
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.

\[
\text{filter}(t, g, l) = \text{reduce}(\text{++, NULL, map}(\text{map-fn}, g, l))
\]

\[
\text{map-fn}(g, l) = \begin{cases} 
\text{array_create_init}(1, 1, l) & \text{iff } t(g, l) = \text{TRUE} \\
\text{array_create}(1, 0) & \text{iff } t(g, l) = \text{FALSE}
\end{cases}
\]

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

```c
bool iseven_fn( void global, int a ) {
    return ( a==2*(a/2) );
}

array<int> x, y;
array<int> 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 \( \gamma \) and an array of local elements of type \( \alpha_0 \). The functions named take two arguments, of types \( \gamma \) and \( \alpha_i \), respectively, and return an element of type \( \alpha_{i+1} \). The skeleton returns an array of type \( \alpha_n \), where \( n \) is the number of elements in the array of function names.
array<type-α_n> pipe(
    array<fname> test_fns,
    array<type-γ> global,
    array<type-α_0> local
);

array<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.

pipe(b, g, l)_i = b_{n-1} (g_{n-1}, b_{n-2} (g_{n-2}, \ldots b_1 (g_1, b_0 (g_0, l_i)) \ldots)) \forall i, \ 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.

```c
int id_fn( void g, int l ) {
    return ( l );
}
int mult_fn( int g, int l ) {
    return ( g*l );
}
array<fname> stage_fns;
array<int> g, y;
stage_fns = array_create_init( 11, id_fn )++array_create_init( 14, 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:

\[
\text{array}<\text{type-}\beta> \ 	ext{map} ( \\
\quad \text{fname} \ \text{body}_\text{fn}, \ \text{type-}\gamma \ \text{global}, \ \text{array}<\text{type-}\alpha> \ \text{local} \\
); \\
\text{type-}\beta \ \text{body}_\text{fn} ( \\
\quad \text{type-}\gamma \ \text{global}, \ \text{array}<\text{type-}\alpha> \ \text{local} \\
)
\]

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.
(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

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>The HSM Node Hierarchy</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>The HSM Data Structures</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>The HSM/Fork95++ Interface</td>
<td>57</td>
</tr>
<tr>
<td>5.4</td>
<td>A HSM/Fork95++ Example</td>
<td>59</td>
</tr>
</tbody>
</table>

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.

```cpp
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.:

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

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

```cpp
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.

```cpp
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:

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

the Fork95++ types will be `HSM_intpair`:

```cpp
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:

```cpp
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:

```cpp
HSM_int fork_body_fn(   
    HSM_void global,   
    HSM_array< HSM_int > localarray   
) {
    pr int i;
    HSM_int sum;
    for ( i = 0; i < localarray.size(0); i += isize() ) {           
        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 `isize()` 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:

\[
\textbf{int MPI_Comm_split ( MPI_Comm parent-comm, int colour, int key, MPI_Comm *comm_out )}
\]

There will be one colour for each child node, and the \textit{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 distri-
bution 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 ele-
ments are stored locally but knows nothing about the location of any other
data elements. A processor has to communicate with all the other pro-
cessors 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 in-
creased storage requirements.