UNIVERSITY
of
GLASGOW

Department of Computing Science

Staged Methodologies for Parallel Programming

Noel William Winstanley

A thesis submitted for a Doctor of Philosophy Degree in Computing Science at the University of Glasgow

April 2001

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

This thesis presents a parallel programming model based on the gradual introduction of implementation detail. It comprises a series of decision stages that each fix a different facet of the implementation. The initial stages of the model elide many of the parallelisation concerns, while later stages allow low-level control over the implementation details. This allows the programmer to make decisions about each concern at an appropriate level of abstraction. The model provides abstractions not present in single-view explicitly parallel languages; while at the same time allowing more control and freedom of expression than typical high-level treatments of parallelism.

A prototype system, called PEDL, was produced to evaluate the effectiveness of this programming model. This system allows the derivation of distributed-memory SPMD implementations for array based numerical computations. The decision stages are structured as a series of related languages, each of which presents a more explicit model of the parallel machine. The starting point is a high-level specification of the computational portion of the algorithm from which a low-level implementation is derived that describes all the parallelisation detail. The derivation proceeds by transforming the program from one language to the next, adding implementation detail at each stage. The system is amenable to producing correctness proofs of the transformations, although this is not required.

All languages in the system are executable: programs undergoing derivation can be checked and tested to provide the programmer with feedback. The languages are implemented by embedding them within a host functional language. Their structure is represented within the type system of the host language. This allows programs to be expressed in languages from a combination of stages, which is useful during derivation, while still being able to distinguish the different languages.

Once all the parallelisation details have been fixed the final implementation is generated by a process of transformation and translation. This implementation is a conventional imperative program in which communication is provided by the MPI library. The thesis presents case studies of the use of the system: programs undergoing derivation were found to be clear and concise, and it was found that the use of this system introduces little overhead into the final implementation.

Acknowledgements

I an grateful to my family, my friends and my supervisor John O'Donnell for all the advice, support, and encouragement they have given over the last four years. Thanks to you all.

Declaration

I hereby declare that this thesis has been composed by myself, that the work herein is my own except where otherwise stated, and that the work presented has not been presented for any university degree before.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel computation promises shorter execution times or the ability to process greater quantities of data compared to sequential computation. However, in practice it is hard to realize a parallel implementation that comes close to achieving its theoretical potential. This is because efficient cooperation between processors is difficult to implement.

Parallelism introduces a new set of concerns for the programmer: the scheduling of computations; placement of data; synchronization; and communication between processors. This adds greatly to the complexity to the programming task. An implementation must manage all these concerns in addition to computing a result.

Parallel programming languages and methodologies typically attempt to assist the programmer in one of two ways. The first approach is to provide layers of abstraction that hide the low-level details of the parallel machine from the user. This simplifies the programming task but reduces control over the finer details of the parallel implementation. Other languages provide as little abstraction as possible and require the parallelisation concerns to be managed explicitly. A skillful programmer can produce efficient implementations in such languages. However they are hard to use effectively; furthermore the code produced is often unclear, brittle and machine-specific.

The weakness of these two approaches is that they present a single fixed level of abstraction. Implementing parallel algorithms is more complicated than implementing their sequential counterparts, while at the same time the efficiency of the implementation is very important. This suggests a programming model that combines the benefits of both approaches: one that abstracts away from the complexity while still permitting fine control when necessary.

This thesis proposes a programming model based on the gradual and systematic introduction of implementation detail. This model provides a sequence of appropriate levels of abstraction. The initial stages of the process provide abstractions over the parallelisation

concerns. Additional concerns are exposed at each stage of the sequence until a fully-specified parallelisation results. This allows efficient implementations to be produced from programs that are clear, modular and portable.

To evaluate the effectiveness of this model we have produced a prototype system that applies the programming model to producing parallel implementations for array-based numerical algorithms.

1.1 Background

We begin with a brief review of some of the terminology, concepts and issues that arise in the field of parallel computation. The reader who requires further detail is referred to a standard textbook such as (Wilson, 1995).

Machine architecture

Parallel machine architectures divide into two broad classes – *shared memory systems* and *distributed memory systems*. Shared memory machines are characterised by a set of processors that all have direct access to a common memory store, through which they may communicate. Distributed memory machines are comprised of a set of nodes interconnected by a network. Each node is a processor with its own local memory. Data is exchanged between nodes by exchanging messages across the network.

We choose to focus our attention on distributed memory machines: these systems are more common, can be constructed from standard uniprocessors, and can accommodate large numbers of nodes, giving greater potential for parallelism.

Writing programs for distributed memory machines is more difficult than implementing a similar shared memory program. Communication is via message passing, which introduces concurrency and possibly non-determinacy: in particular deadlock, livelock and race conditions are all possible. Nondeterminacy greatly confuses reasoning about program behaviour. The characteristics of the interconnection network – its latency and bandwidth – must also be considered. Failure to do so may cause processors that are waiting for a message to block excessively or the network to become saturated.

Implementation issues

As well as specifying the computation to perform, the programmer must manage additional implementation concerns introduced by parallelism.

The most basic is which parallel decomposition to use; that is, how the algorithm is to be broken into independent computations that may be executed in parallel. There may be more than one way that a particular problem can be parallelised. A common method to use is a *data parallel* decomposition. This is suitable for algorithms which operate on regular data-structures such as a matrices or vectors. The data-structure is divided between the processors, which then each compute a portion of the result. More irregular computations are better suited to other decompositions. An example is the *processor farm*. Here a master processor distributes packets of computation to a pool of worker processors. When a worker processor completes a packet a result is returned to the master processor, which collates the results and allocates further computation to the free processor.

A related concern is balancing the computational load between processors. A parallel computation proceeds at the speed of the slowest component: therefore it is important to ensure that a computational component completes at approximately the same time as its result is required by another component.

For some problems is is possible to determine which are the more expensive computations by inspection. In these cases load balancing can be achieved statically by allocating these components more processors. In other situations a dynamic approach must be used in which the implementation adapts to expensive computations by re-routing work elsewhere. The processor-farm decomposition exhibits a simple form of dynamic load balancing: processors which terminate first are allocated more work. Typically, dynamic techniques require more communication of data than static solutions. This can introduce inefficiencies by congesting the communication network.

Another facet in the parallelisation of an algorithm is what scheduling of components to use. The data dependencies within an algorithm may allow a range of orders of execution for computational components. The most suitable ordering depends on the problem decomposition, load balancing and communication patterns used.

In all but the most trivially parallel computations, some data will be computed on one processor and required by another. Communicating data across a distributed memory machine is expensive – the network has significant latency and limited bandwidth. Therefore parallel algorithms are designed to minimize the number of data redistributions required. The aim is to decompose the problem so that as much as possible of the data required by a processor is generated locally or on nearby processors. Another technique is to bundle together in the same communication different data that is to be redistributed in the same way. This may require adjusting the scheduling so that these results become available at the same time.

It can be seen that these implementation issues are all inter-connected: changing one

facet of the design will impact on the others. Often these concerns are in conflict: in such cases an balance must be found between the demands of each of the concerns. Finding this balance is what makes parallel implementation such a black art.

Abstraction in parallelism

Much research into language design and programming methodologies has been concerned with introducing models of computation that abstract away from the low-level machine details. Whether it is an incremental change, such as the introduction of subroutines or the heap abstraction provided by C; or an innovation such as the execution model of Prolog; the aim is the same – to simplify the programming task by hiding some of the complexity of the underlying machine. This is achieved by delegating the management of some of the implementation concerns to supporting software in the compiler or runtime system. As the programmer is released from the requirement to manage these concerns they are more able to concentrate on higher-level problem solving.

Abstractions can have further benefits. For instance, a programming model that abstracts from the details of the underlying machine is more likely to give rise to code that is straightforward to port between platforms. Furthermore, program maintenance is simplified when working with clear code written using suitable abstractions.

However abstraction comes at a price: the programmer has less control over the implementation concerns that have been delegated. This is unimportant when the supporting software produces good results, for example in managing register allocation. Unfortunately this is not always the case. It is known to be computationally difficult to find solutions to some parallelisation concerns, such as assigning computational components to processors (Rauber and Rünger, 1996a). Therefore heuristics are commonly used.

When performance of the support software is suboptimal the programmer will have difficulty in correcting the problem. Although a programmer may have the skill to produce a higher quality implementation, the abstractions of the programming model may prevent them from doing so. It is sometimes possible to subvert abstractions when needed. However such work-arounds – dirty hacks – result in the programmer fighting against the very feature that was intended to make the programming task simpler. Such work-arounds also diminish the other benefits of programming with abstractions – such as transparency, safety, and portability.

The success of a parallel implementation can be assessed by a single measure: compare the runtime to that of an optimized sequential implementation. Because parallel processing is solely motivated by performance, it is often unacceptable to delegate implementation decisions to supporting software that may produce sub-optimal results.

Instead the programmer may favour a programming model that provides the low-level control necessary to produce the best result. This is even though a programming model with few abstractions may make the programming task more difficult, the code harder to reason about and verify, and the resulting implementation harder to debug and maintain.

1.2 Summary of Research

The purpose of this thesis is to show that it is possible to accommodate comfortably both abstraction and low-level control of parallelism within a single programming methodology; and that such a methodology possesses many of the benefits of both abstraction- and control-based approaches.

The methodology is composed of a series of n stages, each of which has an associated language $L_1, \ldots, L_n$. Language L_1 allows the expression of computations: all parallelisation details are left unspecified. Each of the following languages in the series $L_i, i = 2, \ldots, n$ extends the previous language L_{i-1} with constructs that make explicit the implementation decisions of an additional parallelisation concern. Therefore each language has a lower level of abstraction than its predecessor in the series.

The process starts by expressing the computational portion of the algorithm as a program in language L_1. Parallel implementation details are then incrementally introduced by rewriting this program in every language of the series in turn. Each transformation between stages only requires the programmer to make decisions about a single parallelisation concern: the decision is supported by a language that presents an appropriate level of abstraction for that concern. The series of stages provides structure to the derivation. The introduction of parallel implementation details is ordered so that the higher-level, more fundamental decisions are taken before lesser concerns are tackled. By the time the program has been rewritten in language L_n all the parallelisation details have been specified. A conventional implementation can then be produced with little further intervention from the programmer.

We have designed and implemented a prototype of an incremental programming system which we call PEDL (Parallel Embedded Derivation Languages). This system can be used to produce parallel implementations of array-based computations. The result of the system is an imperative program expressed in the Group-SPMD programming model (Rauber and Rünger, 1996b). The target architecture is a distributed-memory machine where communication is provided by the MPI library (MPI Forum, 1995).

PEDL comprises a series of stages which introduce the parallelisation concerns particular to the Group-SPMD model. The languages of the system are first-order functional and share a common core of constructs. All the languages of the series are well-defined

and executable. This allows programs undergoing derivation to be be verified, reasoned about, statically checked and tested. It is possible to perform equational reasoning and program transformation both within a language of a single stage and between languages of different stages. The intermediate programs of the derivation can be used as documentation, proof of correctness and as a starting point for porting the program to another architecture.

Once the program is expressed in the language of the final stage in the series all the implementation details have been provided. A process of transformation and simplification is then performed to produce an intermediate program. This can then be trivially translated to the target language. We have carried through some preliminary case studies to evaluate the utility of the PEDL system.

1.3 Contributions

This section describes the research contributions made by this thesis.

1.3.1 Primary Contributions

Staged programming methodology

This thesis reviews a range of parallel programming systems and their ability to mask the complexity of parallel programming while allowing low-level control where necessary. This survey highlights the inadequacy of a single level of abstraction. We examine a group of programming methods that allow the incremental introduction of detail: these systems provide different levels of abstraction during the implementation process. These systems are compared and their common properties are identified. These qualities are use to motivate our prototype PEDL system.

The PEDL system

The PEDL system was produced to investigate methods of combining different levels of abstraction within the same programming system. It is a staged programming method structured as a series of distinct languages. This combines the benefits of a concrete language semantics with varying levels of abstraction and the structured introduction of implementation decisions. These features permit transformation and reasoning on intermediate programs, and allows them to be simulated and verified.

We have performed case studies that demonstrate the use of the system. Programming in the system was found to be tractable: the intermediate programs are clear and concise while still allowing fine control over the parallel behaviour. The resulting programs are conventional imperative message-passing implementations and have been executed on a network of workstations.

Embedded implementation of languages

The PEDL stage languages are implemented by embedding them within a pure, lazy higher-order functional language. This simplifies the implementation and language design because many features can be inherited from the host language. A further benefit for our system is that the host language provides a common semantic base for all the stage languages.

This implementation serves as a case study of the technique of embedded implementation. As well as using existing methods, such as structuring computation as monads, the novel requirements of embedding a *series* of languages led to the development of new techniques. The languages of the different stages and layers in PEDL must remain distinct and yet in some circumstances can be safely combined. We describe the use of **Phantom Language Types** (Section 5.3) to represent within the type system of the host language the static semantics of the languages and their legal combinations.

1.3.2 Secondary Contributions

This section describes some secondary contributions made by the thesis.

Use of APMs

The parallel behaviour of the PEDL languages is described using the Abstract Parallel Machine (APM) methodology (O'Donnell and Rünger, 1997b). An abstract parallel machine is defined whose site states model the local store of each processor. The parallel constructs are then described by operations over this abstract machine.

APMs are usually used for algorithm derivation. As far as we know, this is the first time they have been used to describe the behaviour of language constructs. It was found that the existing methodology was sufficient to describe constructs which performed communication or distributed computation. However, it was necessary to extend the parallel operation formalism to express constructs that partitioned the parallel machine into independent groups.

Two layer languages

The PEDL stage languages combine two views of the parallel computation: a collective layer that describes coordination and communication and a individual processor layer for expressing computation.

Each layer may only safely manipulate distributed data in certain ways: we must ensure that computation does not occur in the collective layer, and that non-local accesses do not occur in the processor layer. An abstract datatype is used to represent distributed data: this enforces the correct usage by providing different operations for each layer. The collective layer may permutate entire data structures to perform communication, while the processor layer may access the local element of a data structure while performing computation.

The two-layer approach provides more safety and structure than the typical flat parallel programming language; it prevents many operations from being used inappropriately and ensures the integrity of replicated data. However it is not suitable for implementation directly by conventional imperative languages. We define the **Change of view transformation** (Section 7.1) which is used during the production of the final implementation to combine the two layers to leave a residual single-level program.

Mixed language programs

It was discovered that the derivation process was simplified by being able to mix languages from *adjacent* decision stages within a single program. At each stage, the computational models and constructs of the adjacent languages are so similar that a program expressed in a mixture of the languages can be considered to be well-defined. Mixing languages allows the programmer to perform a gradual transformation between stages while using the compiler to validate and execute each intermediate program. The phantom language types ensure that languages may only be mixed correctly and that a clean separation is maintained between components in different languages. This is another level of staging and decision delaying – the whole of a derivation step does not need to be thought through before preliminary steps can be made and verified.

Compilation method

Although the PEDL languages are first-order functional and are implemented by embedding within a higher-order functional host language, the final result of a parallel derivation is a conventional imperative program.

This thesis describes techniques for flattening and simplifying the results of a PEDL derivation so that a program in a simple intermediate language results (Section 7.2). This can then be straightforwardly translated to the target language (Section 7.4). This process can almost all be automated. However, user input is required to translate some small fragments of code expressed in the host language.

1.4 Thesis Structure

The structure of this thesis is as follows:

Chapter 2 evaluates a selection of parallel programming languages and methodologies. These systems are assessed for their ability to provide suitable levels of abstraction to mask the complexity of parallel programming while allowing low-level control where required. We also consider their generality, expressiveness and suitability for distributed-memory machines. We examine a class of systems that seem to offer the best solution and identify the features these systems have in common. From this analysis the criteria for the design of the PEDL system are formulated.

Chapter 3 introduces PEDL, our prototype incremental programming system. The series of decision stages and the corresponding languages are presented. The languages have a two-layer structure of a coordination layer with a collective view scheduling blocks of computational, processor-view code. Methods for allowing the two layers to manipulate distributed data safely are examined. A core set of constructs common to all languages is described, followed by an examination of the features specific to each decision stage language. We also present the intermediate language used in the generation of the final imperative implementation.

Chapter 4 gives a formal definition of the PEDL languages. The syntax and operational semantics of the common language core are described. Following this the constructs unique to each stage are defined. The parallel behaviour of the languages is then specified using the Abstract Parallel Machine methodology. Finally, the technique used for reasoning about the programs is described and examples given.

Chapter 5 describes the implementation of the PEDL system. The development of the technique of embedding within a host language is surveyed and our choice of host language justified. The remainder of the chapter describes techniques for structuring the embedding so that the occurrence of host language features is controlled and a distinction maintained between the different stages and levels of the system.

Chapter 6 evaluates the utility of the PEDL system by applying it to two case studies: the one-dimensional wave equation and the maximum segment sum problem. These studies

demonstrate an incremental programming system in action. The result of the derivations are conventional message-passing programs which have been executed on a network of workstations.

Chapter 7 describes how an imperative implementation is generated from a completed derivation. This has two phases. First the collective-view coordination layer is transformed away. The resulting single-processor view program is then simplified and translated to the target language. The design of libraries to support the implementation is also presented.

Chapter 8 draws conclusions about staged systems for parallel implementation. It evaluates the potential of the staged programming approach, lessons learned from the design and implementation of PEDL, and identifies areas of future work.

Chapter 2

Parallel Programming Models

Capsule

Many programming models have been proposed for parallel programming. The commonest present one view of the program at a single level of abstraction. This chapter reviews a range of these models and compares their strengths. We are particularly interested in models that permit the production of efficient implementations for distributed memory machines. Regardless of the level of abstraction chosen by the language designer, such models must compromise between allowing explicit control of the implementation details and providing increased clarity, structure and conciseness of expression.

Producing a parallel implementation is more complex than producing an equivalent sequential implementation, and yet it is important that the implementation is efficient. This would suggest that the programmer requires the full benefits of both high and low levels of abstraction, rather than a trade-off between the two.

We survey a group of programming models based on the gradual introduction of implementation detail. Such models can provide multiple views of a program, allowing it to be presented at different levels of abstraction where needed. The chapter identifies a set of qualities that are commonly exhibited by these models. These qualities form the basis of the design of the PEDL system.

Introduction

Parallelism requires the programmer to consider a set of factors not present in sequential code. The complexity these features create propagates throughout the program and can seriously hinder the production, maintenance and portability of efficient parallel implementations. A successful parallel programming system should support the programming task by: hiding the inherent complexity of the problem; simplifying reasoning about correctness and parallel behaviour; allowing prototypes to be produced; and be amenable to performance prediction. At the same time, a successful model must allow efficient, maintainable, and portable implementations to be produced. This chapter examines the ability of different systems proposed for parallel programming to satisfy these requirements

We first examine conventional methods based upon a programming language providing a single level of abstraction. Although successful in some circumstances, these models have difficulty in finding a balance between control and abstraction. In some cases the the level of abstraction is too low to aid the programmer much. Others provide the abstraction necessary to manage the complexity but prevent the programmer from having fine control over the parallel behaviour of the program. Instead, much of the decision making is delegated to an optimising compiler or runtime system. If such tools under-achieve, there is little the programmer can do to improve the performance.

Section 2.2 describes a set of program development models based on some form of delayed decision making. Here implementation decisions are made incrementally, often each presented at a different level of abstraction. The gradual introduction of detail make the programming task manageable while exposing the low-level details of the implementation towards the end of the process. We analyze the common properties of these staged models and develop a proposal for a programming system that exhibits all these properties in a concrete setting (Section 2.3).

2.1 Single-Stage Programming Models

In this section we survey the main approaches currently taken for programming parallel machines. We start with the mainstream method, writing parallel programs using a sequential language combined with a communication library. After identifying the strengths and weaknesses of this technique, we examine some alternatives.

2.1.1 Explicit Parallel Programming

The most direct method for programming parallel systems is to view them as a collection of interacting sequential processors. Each processor is programmed using a conventional sequential language. Communication is achieved by calling message-passing libraries. A level of portability can be achieved by using a standard communication API such as MPI (MPI Forum, 1995) or PVM (Geist et al., 1994) which are available on many different platforms.

The popularity of this method is understandable: the programmer does not need to learn a new language; the programmer has direct and absolute control over the way an algorithm is expressed; there are few abstractions to introduce execution overhead; and there is extensive experience, support and libraries of sequential code written using this method.

However, it is not easy to gain a clear picture of the global behaviour of the machine from the program source. Sections where the computations of different processors differ are typically expressed using case statements or expressions over the processor rank. This allows the behaviour of the entire system to be represented within a single executable but the code is liable to get very tangled.

Related to the difficulty in understanding the behaviour of the parallel machine is ensuring that the processors communicate and synchronize correctly. Communicating a single message typically requires two library calls – a send and a receive. As these calls will occur in different branches of the program it is hard to ensure that the pairs of library calls match up as intended. Similarly, calls to initialize communication system objects, such as the *communicators* of MPI, must be executed by all processors, and each must pass identical parameters to the call. The sequential language provides no support for these constraints, and so deadlock and other hard-to-find errors are a common occurrence during development.

Furthermore, this method makes it hard to develop parallel programs incrementally – exploration and experimentation are discouraged because they require extensive restructuring of the code. Rather, the programmer makes a set of arbitrary implementation decisions and then codes the program. Provided the performance is acceptable, the first implementation is often settled upon. It is also difficult to prove properties of programs written in this style because non-determinism is exposed by the communication model.

The use of standard communication libraries such as MPI allow programs to be compiled on various machines. However, performance will suffer unless the configuration of the target machine is identical to the original. To improve performance, the program has to be reworked to the new machine configuration. This is problematic when the program was implemented directly on the original machine, as there is no prototype on which a

ported version can be based, and the tangled nature of the code makes comprehension difficult.

In summary, explicit parallel programming provides fine control of all aspects of the parallel execution but exposes the programmer to great complexity. The code produced is brittle, hard to write, understand, maintain and port to other machines.

Although this is one of the commonest techniques used to produce applications for parallel machines, its many limitations have led researchers to investigate other ways to express parallel computation. The following sections review some of the various solutions proposed. We do not aim for an exhaustive survey, rather an representative sample of the main approaches. A more thorough survey can be found in (Skillicorn and Talia, 1998).

2.1.2 Data Parallel Programming

The data parallel model is concerned with the parallelism that arises from the manipulation of large monolithic data-structures such as arrays. There is a wide class of language designed for programming data-parallel applications. While many of these languages were designed expressly for this purpose, others are languages which were already *collection-oriented* and were later adapted to generate parallel codes. Collection-oriented languages allow the expression of operations in terms of whole data-structures, rather than in terms of iterations over the elements of the data structure. The undefined execution order within the collection-oriented operations allows the computation of the operation to possibly take place in parallel.

The main characteristics of data parallel languages are a global namespace and a single thread of control. Communication and coordination are implicit. Data parallel programming languages are generally scalable and easy to use, but lack fine-grained control mechanisms: the programmer is often limited to just providing hints to the compiler and cannot take further control of the process.

A typical division of labour is to allow the programmer, at most, to express the distribution of the data structures. All other parallel implementation decisions are left to the compiler. Although this model works well for regular data-parallel problems, performance may suffer when the algorithm is irregular. Furthermore, this model is unsuited to problems that are more naturally expressed in a control-parallel manner.

HPF

A well known example of a data parallel language is High Performance FORTRAN (HPF) (HPF Forum, 1993). This is an extension of FORTRAN by a system of annotations

that allow the programmer to describe the parallel behaviour of a program.

The main method for expressing parallel behaviour is the `forall` loop construct. This has the same form and behaviour as the conventional `do` loop but indicates to the compiler when loop iterations can be computed independently. This is more feasible than the compiler analyzing every conventional `do` loop in the program and trying to determine which loops have independent iterations *and* contain useful parallelism. In general, the compiler cannot verify that the loop body of a `forall` construct is actually independent. It is a burden of proof on the programmer that there is no implied sequencing in the loop body. This is a source of errors.

Collection-oriented array operations are also provided by the language, such as element-wise map operations and reduces. These are implemented (conceptually at least) by the `forall` loop construct and provide a higher level of abstraction for the programmer.

The other feature of the HPF system is a system of directives that describe the desired distribution of arrays. Iterations of parallel loop bodies are distributed between processors according to where the corresponding array elements are located. Although HPF has a redistribution directive, the compiler is free to redistribute data when needed. For instance, an element-wise collection-oriented operation involving two arrays with different distributions will result in a redistribution. However, the programmer has no knowledge of how and when the redistribution is performed. If he did he may be able to take advantage of it later in the program, otherwise he may write code that requires more redistribution than is necessary. The only way to circumvent this is to describe all redistributions explicitly, which places a burden on the programmer and leads to over-specified programs.

NESL

NESL (Blelloch, 1992) is a functional language that allows the expression of nested data-parallelism. NESL is a strongly-typed first order language which is loosely based on ML. It has polymorphic types and a system of ad-hoc function overloading. In this language sequences are a primitive type; parallelism arises exclusively in operations on these sequences.

Nested data parallelism is not supported by many data-parallel languages. It simplifies the parallelization of algorithms but complicates the implementation and cost modeling. In particular it is useful for expressing irregular data structures; divide and conquer problems; and algorithms using recursive data structures. Furthermore, it allows parallel components to be reused without the programmer having to be aware of the internal parallel behaviour.

NESL provides an 'apply to each' construct, which maps a function over the elements of

a sequence and also allows filters to be expressed. It is similar to the set comprehension of mathematics, or the list comprehension of Miranda (Turner, 1985). The element computations expressed using this construct are independent and are executed in parallel. The 'apply to each' construct can be nested to express nested parallelism. The user-defined functions mapped over sequences may themselves contain further parallelism. The language also provides a collection of primitive collection-oriented functions that operate over an entire sequence, for example sum, maximum, and reverse. Many of these primitives have a parallel implementation.

Performance prediction can be calculated from the source. Values for two complexity measures, work complexity and depth complexity, are defined for each of the primitives and can be composed across expressions. The work complexity gives the execution time if executed on a serial RAM machine model. The depth complexity represents the deepest path taken by an expression: this gives the execution time when executed on an unbounded number of processors. When used together, the two complexities place an upper bound on the asymptotic execution time for a PRAM machine model.

Compilation of a NESL program generates code for an abstract machine in an intermediate language called VCODE. This is a stack-based language where the objects on the stack are vectors of atomic values. The most important phase of the compilation is flattening the nested parallelism. Nested sequences are converted to sets of flat vectors, and nested maps converted into VCODE operations over the flattened representation. The VCODE is then compiled to C for the target architecture.

In common with many other single assignment languages for parallelism, NESL has an exclusively implicit parallel model. The parallelism is encapsulated within the constructs and primitives provided by the language. Compared to the approach of HPF, this leads to a clean and comprehensible programming style, where the cost models allow informed implementation decisions to be made.

NESL is designed to exploit fine-grained parallelism, and is well suited to fine-grain architectures (such as the connection machine), vector processors and shared memory systems. On such systems, its performance is comparable to hand-coded FORTRAN (Blelloch et al., 1994). However, on distributed memory large-grain systems, much of the potential parallelism in a NESL program will be unproductive. There is no mechanism for the programmer to supply further information to identify the useful parallelism of a program.

FISh

FISh (Jay and Steckler, 1997; Jay, 1998) is an Algol-like language for expressing array computations. It has a higher-order polymorphic type system that maintains information

about the *shape* (Jay, 1995) of data structures; that is, the dimension and size of an array is part of its type.

This type system allows the FISh compiler to determine the shapes and update behaviour of array expressions: using this information functional code can be converted to procedural code that updates in-place and an appropriate amount of storage allocated on which these procedures can operate. Because of this the performance of FISh programs is comparable to hand-coded C, and can exceed it in polymorphic situations.

(Jay, 2000) describes work in progress to produce a data-parallel variant of FISh. Their approach is to isolate a purely functional subset of the language and extend it with second-order array primitives (such as map and reduce) that encapsulate parallel behaviour.

A data-parallel program written in this functional coordination language is compiled into efficient procedural FISh code for each processor. It is claimed that shape-based cost analysis will allow the calculation of efficient data distributions for the parallelised arrays. Furthermore, the programmer may specify data distributions within the existing shaped type system using nested array types. The design and strengths of FISh suggest that this language would be particularly effective at performing low-level optimisations after the algorithmic details have been handled at a higher level.

Bird–Meertens Formalism

The Bird–Meertens formalism (BMF) (Bird, 1987; Bird and de Moor, 1996) is a calculus for reasoning about functional specifications. Most work has focussed on the theory of lists but other datatypes have been explored as well. A datatype and a set of operators over it are defined, along with a collection of basic laws that describe the interactions of the operators. By repeatedly applying these laws it is possible to start with a formal naive specification and, step by step, convert this to an efficient program.

(Skillicorn, 1990; Skillicorn, 1991) advocates the use of BMF as a model for parallel computation. BMF combinators such as map, reduce and filter are considered to contain implicit data-parallelism. The parallel implementation of these operation is usually left unspecified although costs may be provided for each operation.

BMF is sufficiently abstract to be architecture independent, while the strong theory provides a framework for constructing correct code. The equational laws can be used to transform from one parallel realization to another. This makes it an attractive framework for designing and reasoning about data-parallel algorithms. However, as BMF is a transformational notation rather than a programming language there is no defined syntax, semantics, cost model, or implementation. Therefore once an algorithm has been

designed it must be implemented by another means, possibly using skeletons or a data-parallel language that provides operations similar to the combinators used.

2.1.3 Synchronous Parallel Programming

Programming models that been proposed which allow data-parallel computations to be expressed in a more explicit fashion than the mostly-implicit approaches described in the previous section. In these models the choice of communication, data distribution and scheduling is made by the programmer rather than by the supporting software. Barrier synchronization of all the processors occurs at regular intervals. This divides a program into a series of steps and eliminates visible non-determinism, which simplifies reasoning and performance prediction. Two of the best known examples are BSP (Valiant, 1989; Skillicorn et al., 1996) and SPMD, which is described in (Pfister, 1998).

A BSP program is structured as a series of *supersteps*. In each superstep processors may perform local computation and issue requests to send and receive data. A superstep is terminated by a global synchronization, in which all communications requested within the superstep are completed. Communication is asynchronous: a request for remote data is not guaranteed to be satisfied until the end of the current superstep, when all processors are synchronised.

This pattern of communication and synchronisation prevents most cases of deadlock and livelock occurring. The only time deadlock may happen is when processors do not all participate in the synchronization operation that delimits supersteps. The prevention of deadlock is a valuable aid to the programmer.

BSP provides a cost model for calculating the computation and communication costs for a superstep. While the model is simple, it is accurate enough to allow the programmer to make informed implementation decisions.

Although there exist languages specifically designed to support BSP-style programming – GPL (McColl and Miller, 1995) and OPAL (Knee, 1994), an object-based programming language – BSP implementation is commonly done using a conventional programming language in conjunction with a BSP communication library such as that produced at Oxford (Miller, 1993). Although this provides more support and safety for the programmer than working with raw communication libraries such as MPI, the method retains many of the weaknesses of programming parallel machines with augmented sequential languages. For instance, there is still little protection from incorrect use of the library functionality; the compiler cannot ensure that all processors call the synchronize procedure at the same points in the program.

The SPMD programming model is similarly structured as a series of supersteps. Each

superstep in this model has a computation phase, where each processor performs local computation, followed by a communication and synchronization phase where processors exchange data using a synchronous collective communication.

A collective communication is a regular pattern of message exchanges that all the processors participate in. Examples include broadcast, gather and fold. The behaviour of these operations is known in advance, allowing finely tuned implementations and cost models to be constructed. Collective operations provide a clearer view of the data redistribution pattern then arbitrary sends and receives. This improves comprehension: it is possible to ascertain from a single point of the program what communication pattern is being performed.

The synchronisation provided by the model removes the non-determinacy inherent to the more general message passing style. Although the implementation of a collective communication may be non-deterministic, this is hidden within the operation. After the synchronization barrier the machine is in a stable known state. As with BSP, this means that deadlock is less likely to occur.

Not all algorithms can be expressed efficiently in the SPMD model. Group-SPMD (Rauber and Rünger, 1996b) is an extension of this model that allows SPMD computations to be composed in parallel to calculate intermediate results of a more complex computation. The processors of the machine may be partitioned into disjoint groups. Each group executes independently in the SPMD style: no synchronization or communication takes place between groups. Therefore each group can be reasoned about in isolation. The results of the computation of the child groups are made available to the parent computation, which resumes once all the child group computations have completed.

Although these model appear restrictive, they may lead to better implementations than the arbitrary message-passing model because programs are more amenable to reasoning and performance analysis. These models reduce the occurrence of dangerous non-determinism and provide abstractions over the simple message-passing routines. This allows more accurate cost analysis of implementations and better comprehension of the code.

A limitation of these models is that they lack support for specification or prototyping. When programming with an imperative language and a communication library, there is no enforcement of the programming model or detection of aberrant programs. Although programs which conform to the model have better properties for reasoning, little guidance is provided on how to design and construct a program so that it fits the model.

2.1.4 Skeleton Programming

It has been observed by (Danelutto et al., 1992) that explicitly parallel programs are made up of two different kinds of code: task specific code that implements the steps of the algorithm; and code for structuring the program into patterns of computation and communication for parallel execution. The second kind of code deals with the problematic aspects of parallel programming and handles the low-level details of the target machine.

Although the code used to structure a parallel computation is complex, it often forms familiar patterns. For instance (Pritchard et al., 1987; Pritchard, 1988) identify three basic classes of parallelisation paradigm: processor farms, where computational tasks are distributed to independent worker processes; geometric decompositions, where a data structure is distributed uniformly between processors so that each is responsible for an portion of the model; and algorithmic decompositions where each processor executes a component of the total algorithm and data is passed between processors in pipelines.

(Cole, 1989) names such parallelization patterns 'Algorithmic Skeletons' and examines their use to structure parallel programming. Skeletons are high-level programming language constructs which encapsulate a particular pattern of parallelism. They are parameterised by blocks of sequential code that describe the task-specific parts of the parallel computation. The programmer is supplied with a fixed repertoire of skeletons: parallel applications are produced by composing skeleton constructs and providing the task-specific code.

There are two descriptions associated with an algorithmic skeleton. A semantics gives the relationship between the inputs and outputs of a skeleton, while a behavioural model describes its parallel complexity and cost. This is all the information provided about the skeleton: its implementation is completely hidden from the programmer. Indeed, a skeleton may have more than one realization on a particular machine. A skeleton compiler uses the behavioural models to calculate an optimal instantiation of the skeletons in the source program for the target machine.

There are many different skeleton systems in the literature. They differ in the presentation of the skeletons, the underlying computational language and the palette of skeletons made available to the programmer.

- In his thesis Cole notes that the functionality of many skeletons can be expressed concisely as higher order functions in a functional programming language. He identifies four basic skeletons and produces performance models and implementations for them.

- The Pisa Parallel Processing Language (P^3L) (Danelutto et al., 1992; Bacci et al., 1995) has two components: a sequential language for writing computational code

and a set of constructs (skeletons) for structuring parallelism. C++ is used as the computational language. The supported skeletons include farm, map, linear and tree pipelines, geometric and loop parallelizations.

P^3L skeletons may be nested. The compiler selects one of several implementation techniques for each skeleton in the program using heuristics. Further tools can then be used to optimize the program for a particular architecture.

The language allows the programmer to escape from the skeleton paradigm and hand-code communication by calling the MPI library directly. This could be seen as confirmation of a common criticism of skeleton approaches: that a limited selection of parallel abstractions is insufficient for some programming tasks. Furthermore, the ability of the skeleton compiler to produce efficient code for such mixed-paradigm programs is questionable.

- Skeletons have been embedded as higher order functions in the functional language Hope+ (Darlington et al., 1991; Darlington et al., 1993). The semantics of each skeleton is given by the definition of the corresponding higher order function. Abstractions are provided for farm, pipe, divide & conquer, map & fold and dynamic message passing parallelism.

  Efficient portability over a range of architectures is claimed to be possible by using program transformation to rewrite an application as a composition of skeletons known to be efficient on the target machine. This is guided by performance prediction models for each machine.

The benefits claimed for skeleton-based languages include: less programming is required to produce an application; a higher level of abstraction than explicit communication; a modular program structure that aids comprehension and code reuse; portability, perhaps only requiring recompilation for the new architecture; and a management of complexity that allows scalability.

Although skeletons appear an excellent solution to the complexities of parallel programming, they are best suited to 'run of the mill' applications that fit into patterns foreseen by the language designer. It may not be possible to efficiently express an algorithm requiring a novel parallelisation in the fixed repertoire of abstractions provided.

The skeleton compiler takes the decision of which combination of skeleton instantiations will provide the best performance. This hides much of the complexity from the programmer. However, if the performance achieved is less than expected, there is little the programmer can do to rectify this. Indeed, as the compilation process is opaque, it is often difficult to see precisely where the optimisation is failing.

2.1.5 Dataflow Languages

In the dataflow model of computation, of which (Herath et al., 1987) is a description, a program is represented by a directed graph. The nodes are functional units, which can be any size but are usually small, while arcs represent data dependencies between units. A functional unit may *fire* whenever data is present on all its input arcs: that is it consumes the inputs in computing a result which is placed on the output arc.

As execution of functional units depends solely on data dependencies several nodes may fire simultaneously. This gives the dataflow model rich opportunities for parallel evaluation. A parallelising compiler decomposes the dataflow graph between processors. The arcs crossing processor boundaries then represent data that must be communicated. Because of the fire-when-ready behaviour, all communication is asynchronous.

Once a node has computed an output value, this value cannot subsequently be changed. Therefore languages that prevent multiple assignment are more naturally translated to this model: single-assignment or first-order functional languages are typically used. There are a range of such languages, Id (Arvind and Nikhil, 1989) and Sisal (McGraw et al., 1985) being good examples.

These languages have similar features: referential transparency with special looping constructs that give an imperative rather than a recursive feel to programs. Parallelism is implicit, being extracted from loop iterations, monolithic array operations and parallel evaluation of function arguments. They commonly lack more advanced functional features. An interesting exception is Lucid (Ashcroft and Wadge, 1985) which introduces infinite streams of data as a primitive type.

Mentat (Grimshaw, 1993; Grimshaw et al., 1991) takes a different approach to dataflow computing. This language is an extension of C++ which allows the programmer to exploit the parallelism in calling methods on independent objects and the parallelism within single methods. Mentat class and member function definitions may be annotated to indicate useful parallelism and hint at placement. The language is compiled to a medium-grain dataflow model where communication and synchronisation are managed by the compiler and runtime system.

The dataflow model is abstract, simple, and can be efficiently implemented: on shared memory systems Sisal performs as well as FORTRAN code (McGraw, 1993). However the unstructured communication means that the dataflow model becomes inefficient on distributed memory machines. Furthermore, the dynamic essence of the computational model makes cost prediction impossible.

2.1.6 Parallel Functional Languages

Graph reduction (Peyton Jones, 1987) is the computational model underlying functional langauges. Functions are represented as trees, while shared subtrees represent common subexpressions. Computation rules select a portion of the graph, reduce the subgraph to a simpler form, and then update the program graph with the reduced expression.

The Church-Rosser theorem (Church, 1941) states that a functional program will compute the same result under any evaluation order, including concurrent orders. Therefore this computational model can be parallelised by concurrently applying computation rules to independent sections of the program graph. Expressions that are ready for reduction are tasks that embody potential parallelism. Multiple processors can search the graph for such expressions and evaluate them in parallel.

While simple in principle, in practice this pure approach has given disappointing results. There is a high level of parallelism within a typical program graph, but much of it is too fine grained to be useful. Hence practical parallel graph reduction systems rely on various degrees of programmer annotation to indicate the of the intended parallel behaviour of the program. The level of control varies from highly explicit languages to mostly implicit approaches. A review of many of these systems is given in (Hammond, 1994) while (Hammond and Michaelson, 1999) describes current areas of research in this and related fields.

Parafunctional programming (Hudak, 1986) is a lazy functional programming system which provides a large amount of control over the parallel behaviour. While communication and synchronisation are implicit, annotations allow the programmer to control the evaluation order and specify on which processor a given expression should be evaluated.

Caliban (Kelly, 1989) is even more explicit. The language contains computational and wiring sub-languages. The functional wiring language can be used to describe task placement, network topologies and communication patterns. As it is a fully-featured functional language, higher-order functions can be defined that act as templates for these parallelisation concerns. The wiring program is evaluated statically to produce the parallel structure within which the computational program is executed.

The properties of a functional language allow parallelisation patterns to be conveniently defined and reused. However these explicit approaches suffer many of the problems of conventional explicitly-parallel languages: the program contains detailed knowledge of the target architecture, and so portability suffers.

An example of a more implicit approach is *Glasgow Parallel Haskell* (GpH) (Trinder et al., 1996). Parallelism is expressed soley in terms of the following Haskell operators.

$$a \text{ `}seq\text{` } b :: a \rightarrow b \rightarrow b$$
$$a \text{ `}par\text{` } b :: a \rightarrow b \rightarrow b$$

The *seq* operator specifies the order of evaluation for two expressions. The first argument is evaluated to weak-head normal form and then the second argument, which is returned. The *par* operator introduces parallelism. It indicates that its first argument could be evaluated in parallel; the second argument is simply returned. This operator only provides a hint to the runtime system: whether a parallel thread is actually created depends on the load at the time.

Used in combination, these operators can express rich parallel behaviours. However the computational code can become cluttered with annotations which reduces the modularity and comprehensibility of the program. *Evaluation strategies* (Trinder et al., 1998) provide a more structured way to describe the evaluation order and parallelisation of computational code in GpH. An strategy is a function that traverses a data structure applying the two primitive operators to the elements of the data structure to force evaluation and spark computations: the primitive operators are now only used within strategic functions. A strategic function makes no contribution towards the value being computed and is evaluated purely for effect. By applying a strategy function to the result of a computation the evaluation order and parallelisation of that computation can be controlled. The computational code itself is unchanged. This technique have been used successfully to parallelise large sequential Haskell programs (Loidl et al., 1998) with a minimal amount of modification to the source.

Parallel functional languages can express both data- and control-parallelism. They have shown acceptable results for parallelising irregular computations on shared-memory systems. However the unstructured communication and small granularity makes them unattractive for distributed-memory machines. Non-strict evaluation makes it difficult to produce cost measures for these languages. Instead performance is improved by extensive profiling of the program on simulators (Hammond et al., 1994) and the target machine.

Parallel graph reduction necessitates a large runtime system. The structure of the program graph changes constantly during evaluation. An efficient runtime system must have mechanisms for granularity control, load balancing, scheduling and resource allocation. In addition to removing much of the control from the programmer, the implementation and tuning of such a system requires much effort.

2.1.7 Summary

The parallel models reviewed in this section can be ordered according to the level of abstraction they provide over the underlying parallel machine.

At the one extreme are explicit languages that provide few abstractions and require the program to manage distribution, communication and synchronisation itself. Such languages have predictable cost models, are efficient on all architectures and allow fine control of the parallel behaviour of the program. However, programming in these languages is a difficult task. There is no support for specification or prototyping. It is all too easy to introduce concurrent errors such as deadlock. Programs are by necessity architecture-dependent, reducing portability. Code comprehension is reduced because the algorithmic code is muddled with machine-dependent details.

At the other end of the scale are implicit parallel languages which hide the architectural details and parallelisation decisions from the programmer. Programs in these languages are concise, elegant and easy to code and reason about. The high level of abstraction allows portable programs to be written, and prevents many concurrency errors from occurring.

Unfortunately, the computational models of some of these languages are unsuitable for distributed memory architectures and make performance prediction difficult. Other languages restrict the forms of parallelism that can be exploited, making them unusable for some applications. Still others rely on compiler analyses to produce efficient parallelisations. In cases where the compiler underperforms there is little the programmer can do, even if they have insight into the best way to parallelize the code.

Between the two extremes of control and abstraction lie a range of languages that attempt to find an acceptable balance of the advantages and disadvantages of each. Just as in any other kind of programming there are acceptable tradeoffs in parallel implementation. A suboptimal implementation may be acceptable: perhaps it achieves the desired runtime; or maybe further improvements are not worth the greater effort necessary. However, where performance is disappointing and cannot be improved on using the current set of abstractions, the programmer has no choice but to move to a more explicit programming model that provides greater control.

2.2 Many-Stage Programming Models

The essence of the problem is that the parallel programming models surveyed so far present a single fixed level of abstraction. This is acceptable for sequential programming tasks: an appropriate choice of language can be determined by the complexity of

the problem domain and the importance of efficiency. However parallel algorithms have greater complexity than the corresponding sequential algorithm but at the same time efficiency is a prime motivation. This suggests a programming system that exhibits the benefits of both the high- and low-level programming models.

One solution would be to use a combination of programming systems. For instance, algorithms could be designed and analyzed using an abstract model such as BMF or PRAM, prototyped using a high-level language such as GpH, and then efficiently implemented in C+MPI. This would allow the benefits of each approach to be used at appropriate stages in the process. However, it is questionable whether such a technique would ease the implementation process or merely replicate work. Further issues are the degree to which the capabilities of the different models match and the added cognitive load of working with, and translating between, a variety of different systems. The final efficient implementation will still contain unstructured architecture specific details: this may have to be discarded altogether when porting to another architecture.

A more attractive approach is a unified system that allows effective specification, prototyping, reasoning and implementation at appropriate levels of abstraction within a single programming environment. This section reviews a series of programming models, both sequential and parallel, that go some way towards achieving this.

2.2.1 Programming by Transformation

The general idea of transformational programming is to start with an intuitively clear but probably inefficient algorithm and to transform it until an equivalent solution with an efficient implementation is reached. The best known approach to program transformation is the fold–unfold methodology (Burstall and Darlington, 1977). This defines six simple rules based around replacing function calls with the definition of the function, and vice versa. This methodology has been effective at a broad range of program transformations. However it does not guarantee to preserve total correctness – the termination properties – of the program.

Constructive programming is another approach to program transformation. It is based on the theory of the algebra of programming (Bird and de Moor, 1996). The program is expressed in terms of a set of recursion combinators such as catamorphisms (folds). Each recursion combinator is accompanied by a set of laws that characterize its behaviour. The program is transformed by a series of applications of these laws. Expressing the program solely in terms of these combinators make the recursion structure explicit, simplifying the analysis required.

Transformation has also been applied to developing parallel implementation, using both

manual and automated approaches. The manual techniques tends to be more flexible but burden the programmer while the automated transformations are easier to use but may not produce the best results. An example is (Fitzpatrick et al., 1994) which uses automated transformations to derive parallel implementations from functional specifications of numerical algorithms.

Programming by transformation appears to be a promising technique. The programmer first produces a specification or prototype, which is then directly manipulated by the methodology. The transformed program is provably correct, while the result of a derivation is the implementation rather than a description of the implementation. The derivation process incrementally introduces detail into a clear specification to produce the final implementation.

However transformation by hand is arduous for the programmer. Much of the work has a strong exploratory flavour: there is little guidance provided by the methodology on which transformation to apply at a certain stage in the derivation. Although suitable for research into algorithmics, transformation by hand does not provide enough structure for the average programmer.

2.2.2 SAT: Stages and Transformations

Stages and Transformations (SAT) (Gorlatch, 1996; Gorlatch, 1998) is a methodology for deriving SPMD programs. It provides two views of the implementation – the abstraction view and the performance view.

In the abstraction view, programs are single-threaded and expressed as compositions of stages. Each stage is represented by a BMF combinator which encapsulates parallelism. There is no parallelism between stages. A BMF combinator has a corresponding skeleton implementation and a cost measure. Equational reasoning is used to transform simple program specifications into more efficient compositions of skeletons.

Once an efficient parallelisation had been found the abstraction view program can be used to generate an equivalent performance view program. These programs are expressed in an imperative pseudo-code resembling FORTRAN with MPI communication constructs. To achieve this, each BMF combinator is translated to the corresponding skeleton implementation, while user-supplied functions must be converted by hand into computational pseudo-code. During this process, further optimisations are applied to remove the implicit gathering of intermediate data structures between each BMF combinator in the abstraction view.

The result is a detailed but concise pseudo-code specification of the optimised algorithm, which may then be used as a plan for the implementation.

SAT is more general than typical skeleton programming systems because it allows the programmer to define new parallel computational components where needed: in a skeleton system the programmer has a fixed toolkit of abstractions. To add a new abstraction, the programmer must supply a pseudo-code description of the skeleton, and then introduce a new BMF combinator and laws that model the properties of the skeleton. The new component may then be used in further program derivations.

The split level approach provides a clean separation of concerns and appropriate levels of abstraction for each stage. Program optimisation and reasoning is simplified by using a functional representation such as BMF. Meanwhile the performance view provides a clear description of how the program should be implemented.

As SAT uses BMF in the specification stage, it suffers from some of the same problems as BMF.

- Only data-parallel algorithms can be expressed in this model. Furthermore, building programs solely using composition can sometimes lead to quite awkward expressions.
- Parallelism is treated informally. Although the skeletons of the performance view give a description of the parallel behaviour of a combinator, this information is not available within the abstraction view.
- The system lacks concreteness. BMF is a transformational notation with no defined syntax or semantics. Therefore it cannot be verified or simulated. Similarly, the pseudo-code of the performance view lacks a defined semantics. The result of a SAT derivation is a description of the implementation, rather than the implementation itself. The ease of producing this implementation relies on the description being well-formed and sensible.

Another limitation of this method is that it is only concerned with the efficient combination of parallel components. It provides no guidance for constructing new components. Although many components will be quite simple and have straightforward textbook implementations, this may not be the case for all user-defined combinators.

2.2.3 FAN: Formal Abstract Notation

Formal Abstract Notation (FAN) (Gorlatch and Pelagatti, 1999) is a successor to SAT that provides an abstraction level in which to design skeleton programs. Again it is a two level derivation method. The specification view is now expressed in a simple functional language. This allows names to be bound to the results of intermediate computations and

provides a language definition that allows verification and testing. The pseudo-code of the performance view is replaced by the P^3L skeleton programming system discussed earlier (Section 2.1.4). This provides concreteness and removes the need for a further implementation phase.

The skeletons available in P^3L are modeled as combinators within the functional specification language. Each has a cost associated with it. The design process starts by writing a functional version of the algorithm; this is then manipulated via equational reasoning to improve the performance. The programming system provides a transformation engine: this has a store of transformation strategies which it attempts to apply to the program. Depending on the matches found, the system suggests a choice of alternatives to the user, along with a cost estimate for each.

After several iterations of design choice, the specification can then be translated to a P^3L program in the performance view. The P^3L compiler performs further optimisation to the code, although these are of a simpler nature than the transformations possible in the specification view.

FAN is a concretization of the earlier SAT approach which solves many of the problems we identified above. However, the output is a skeleton program, rather than explicitly-parallel code. The ability to add new abstractions to the derivation system is lost, and the reservations expressed about skeleton approaches – expressiveness, programmer control and efficiency – now apply.

2.2.4 Aspect Oriented Programming

The key abstraction and composition mechanism of imperative, object-oriented, and functional languages is some form of generalized procedure. Although the exact nature differs between language paradigms, design methods for these languages tend to perform a functional decomposition – they break a system down into units of behaviour or function. Such design methods are effective at structuring the computational components of a system. The components can be cleanly encapsulated as generalized procedures which can then be easily accessed and composed.

In addition to containing computational components, a system also contains *aspects*. These are properties that affect the performance or semantics of the computational components in a systematic way. Examples of aspects include policies for: synchronization; error or failure handling; and communication. Aspects typically cannot be cleanly encapsulated as generalized procedures because they *cross-cut* the functionality of the system.

The inability of functional decomposition to isolate aspects leads to tangled programs where aspect code occurs throughout the computational components of the system. A

familiar example of this is producing error-tolerant code. Adding good support for failure handling to a simple system prototype requires many little additions and changes throughout the system. This violates the abstraction boundaries provided by the design method.

The goal of *Aspect Oriented Programming* (AOP) (Kiczales et al., 1997) is to cleanly separate components and aspects from each other. An AOP system provides a computational language in which to implement the computational components. It also provides one or more aspect languages in which the programmer can specify aspect policies. A tool called an *aspect weaver* is then used to combine a component program and aspect programs to produce the final implementation. The aspect programs describe transformations that implement policies which the aspect weaver applies to the computational program.

Aspect language designs range from the high-level where policies are expressed by declarative statements to low-level approaches where policies are expressed as explicit transformations of the computational language. It is possible that particular designs are better suited to different classes of aspect. However it is clear that further research of the design space for aspect languages is needed.

Aspect oriented programming gives a separation of concerns and enables the programmer to focus on one aspect of the implementation at a time. A prototype can be developed in the computational language and then combined with simple aspect programs that describe default policies. Detail can then be added incrementally to the prototype by refining the aspect programs. The code remains clear because the weaver program performs the tangling of aspects, rather than requiring the programmer to add them by hand.

It may be possible to produce an aspect-oriented parallel programming system since parallelisation concerns such as communication and synchronisation are commonly cited examples of aspects. Such a system would allow the algorithmic code to be tested and verified in isolation, while the parallel behaviour of the code was described by separate aspect programs. However, much research is still required into the practicality of the AOP approach. Open questions include the best division between computation and aspect concerns, design guidelines for the various languages, theoretical support for the model and methods to construct correct aspect weavers.

2.2.5 TwoL Programming Methodology

The *Two-Level* model (TwoL) (Rauber and Rünger, 1996b; Rauber and Rünger, 1996a) is a programming methodology for deriving efficient Group-SPMD implementations of numerical methods for distributed memory machines. The methodology can express

algorithms that possess two levels of potential parallelism: a high level task parallelism which may contain a nested lower level of data parallelism. This makes it suitable for a large class of partially-irregular problems.

The methodology is structured as a fixed series of design stages. At each stage a high-level modeling language is used to express the sequential and parallel combination of *basic modules*. These are data-parallel components implemented in the SPMD model. The most efficient method of combining basic modules can be calculated using the methodology's cost model.

The programmer starts by writing a *module specification* from a mathematical description of the numerical method. This divides the algorithm into submethods or modules. The specification has a hierarchical structure and expresses data dependency and independence between modules. This describes the maximum degree of parallelism available and removes the need for dependency analysis, as all is made explicit in the specification.

From the specification a *parallel frame program* is derived. This is a non-executable description of the final implementation which fixes the distributions of data, execution order of modules, assignment of processors to modules and all other implementation decisions. This description can be used to predict the runtime of the final implementation.

The derivation of a parallel frame program from a module specification is split into several decision stages. Each stage in the derivation introduces one major decision about the realization of the algorithm. TwoL provides cost analysis methods for predicting the impact of the alternative choices available. At each stage the module specification is augmented with additional constructs to capture the decisions made for that stage. Therefore, the specification language has several dialects or extensions.

The first stage fixes the scheduling of modules. The programmer decides which modules are to be executed concurrently on disjoint processor groups and which consecutively by all processors in the group. The next stage balances the computational load between modules that execute concurrently to ensure they terminate at approximately the same time. This is done by fixing the sizes of the disjoint processor groups that each concurrent module will execute on. The final stage fixes the data distributions of the inputs and outputs of modules. The aim of this stage is to minimise the amount of data redistribution, and so communication, required within and between modules. Where data is passed between modules with incompatible data distributions, redistribution operations are inserted to indicate that communication is necessary.

TwoL provides a cost analysis based on an abstract machine which is described by a few parameters. The values of these parameters are found experimentally for a particular target machine. Cost functions can be calculated compositionally from the components of the partially-specified frame program. The parameters to these functions are the costs of

the unspecified portions of the program. Mathematical optimisation techniques can then be used to minimise these cost functions and so make optimal implementation decisions.

It is interesting that an optimal data distributions for a basic module need not lead to a globally optimal solution. On the contrary, the optimal global solution may require suboptimal distributions – and so extra redistribution – for some components.

Analysis

The strength of TwoL is that it allows the systematic derivation of efficient implementations while requiring the programmer to provide only a description of the computational method. From this specification an optimal implementation can be calculated using the cost models.

The TwoL modeling language expresses the composition of modules and the distribution of their inputs and outputs rather than the computation each module performs. Therefore intermediate programs are not executable, and cannot be tested to ensure that the correct result is being computed. Furthermore, the result of a TwoL derivation is a description of the implementation, rather than the implementation itself. The implementation must still be coded in the conventional way, which can lead to problems ensuring that the implementation faithfully follows the specification.

The TwoL methodology is primarily concerned with calculating an efficient composition of modules. Its derivation stages are concerned with the task-parallel level of the algorithm. The system assumes the existence of libraries of data-parallel basic modules, with a range of different parallelised versions and runtime information for each. Many basic modules are provided by common parallel numerical libraries. However, the methodology provides no guidance for the construction of new basic modules.

The derivation process could, at least partially, be automated by producing interactive or automatic compilation tools for each stage of the design process. (Fissgus et al., 1999) describes the implementation of a prototype TwoL compilation system, focusing primarily on the generation of a final C+MPI implementation from the parallel frame program. This also provides a simple single-assignment language in which to implement basic modules, but this lacks the staged design methodology and supporting cost model of the TwoL system proper.

2.2.6 Abstract Parallel Machines

Abstract Parallel Machines (APM) (O'Donnell and Rünger, 1997b; O'Donnell and Rünger, 2000) is a methodology that derives parallel programs though a sequence of stages, starting with an abstract specification and finishing with the executable program. Decisions

about the parallelism are introduced gradually, one per stage, where each decision is made using an appropriate model of parallelism. Each stage may be proved equivalent to the previous one using equational reasoning.

An Abstract Parallel Machine is a model of parallel computation. It comprises a set of sites of computation, each with a local state, and some parallel operations (ParOps) that manipulate these site states. The external view of a ParOpis a function that takes some input data and the machine state and computes some results and a new state. The internal

the computations performed at each site and how data is communicated between them. ParOps are expressed in a standard form that facilitates comparison, reasoning and transformation. By choosing the appropriate view the programmer can reason about just the

Algorithms are expressed by combining the ParOps of a single APM using a coordi-

coordination language. For more complex algorithms that require control structures and naming of intermediate results a pseudo-code, a standard imperative programming language, or a functional language such as Haskell could be used. The non-strict nature of Haskell is also convenient for implementing ParOps: the standard form of a ParOp

language.

grammer can test and experiment with early versions of a program, use the compiler to detect many errors, and use the language semantics as a foundation for equational reasoning.

APMs are arranged into a tree hierarchy where child node APMs are able to realize the operations of the parent node APM. Transformation rules can be used to describe how this is achieved. A particular realization may only be possible in certain circumstances: the edges of the tree can be annotated with side-conditions that capture these cases.

A program is derived via a sequence of program transformations. Some (*horizontal transformations*) transform the algorithm within the same computational model. Other steps (*vertical transformations*) change the computational model by switching from one APM to another. In this case, the program is essentially unchanged but the parallel behaviour of the program is realized using the operations of another APM. The correctness of such a step relies on theorems relating the semantics of the operations of the two machine models. It may be the case that there is a one-to-one correspondence between the operations of the APMs; in other cases several parallel operations of the target APM may be composed together to realize an operation of the source APM.

It is common for there to be several ways to transform a particular version of the algorithm using different APMs: this can lead to implementations for different architectures. Furthermore different algorithm derivations may share some higher-level transformation steps and the APMs models they are expressed in.

It is possible to define new APMs where needed and link them into the hierarchy. As only a few case studies have been performed so far (O'Donnell and Rünger, 1995; Goodman et al., 1998) the hierarchy is still under development. However, these studies have shown that APMs can be collected and reused between derivations.

Analysis

The APM methodology allows incremental derivation of parallel programs supported by appropriate machine models at each step. The machine models are separated from the algorithm and organized into a hierarchy representing possible transformation paths. This simplifies derivations and encourages reuse. The separation also allows computation to be expressed in a combination of parallel models, which may be useful for larger algorithms.

As ParOps are represented as functions, the APM methodology cannot easily express algorithms which involve non-determinism. Finding a solution to this is the topic of ongoing research (Goodman and O'Donnell, 1999).

While APMs support a staged decision making process, the stages are not fixed. Additional APM models and transformations can be added to the hierarchy. This increases the versatility of the system and allows it to be applied to many different problem domains. However, this freedom provides less direction and guidance for the programmer than a fixed sequence of decision stages that target a particular architecture, such as is found in TwoL.

The APM methodology has no strong link to a particular coordination or computation language. This allows appropriate choices to be made for each derivation but may not be concrete enough for low-level implementation work. Modeling low-level details complicates the ParOp model and can make code unmanageable.

This suggests that the APM methodology is better suited to the exploration of new parallelisations of algorithms, rather than managing low-level implementation concerns for algorithms where most of the parallelisation details are already known. Once the high level algorithmic decisions have been made in the APM methodology, it may be more productive to translate the resulting specification to a more concrete implementation language than to continue the derivation in the APM system.

2.2.7 Discussion

The programming methodologies reviewed in the previous sections provide high levels of abstraction while still permitting efficient implementations. These methodologies differ in some of their details. However, they broadly share a common set of qualities, which we summarize below.

Incremental introduction of detail. The methodologies start with a clear specification of the core details of the algorithm and then incrementally introduce optimisations and further detail until an implementation results. This sequence of decisions may serve as documentation, proof of correctness and prototype for the implementation. Portability is assisted by the ability to backtrack up the decision sequence and branch at a suitable point when targeting a different machine architecture. The incremental approach may also encourage more alternatives to be considered before choosing one to implement.

Discrete stages. The introduction of implementation detail is commonly divided into discrete stages. This decomposes the implementation task into smaller, more manageable subtasks, allowing the programmer to focus on one concern at a time. Often a different level of abstraction is provided for each stage – in the form of notation, language or machine model – that is tailored to expressing the decisions made at that stage.

It seems preferable for an incremental programming system to comprise many small, manageable, stages rather than a few large stages with significant distance between them. However, the problem domain may restrict the number of decision stages that the process can be usefully decomposed into.

Fixed series of stages. Some methodologies have a fixed series of well-defined decision stages. This provides a stronger framework for the programmer to work within that removes some of the uncertainty and exploratory nature of the implementation task. However, such a framework often restricts the implementation to a particular target machine or parallelisation paradigm.

Progression by transformation. Many of the methodologies introduce implementation detail through transformation of the intermediate program. These transformations may either be formal or informal, although most systems allow the user to produce correctness proofs where required. Typically the transformation is guided by reasoning about the computational and parallel behaviour of the program, which in some cases is developed into a system of performance prediction. This reasoning is simplified by using multiple levels of abstraction, so that details that are too low level for a particular stage can be elided.

Executable intermediate programs. Methodologies which are based on concrete computational languages, rather than informal notations or specification languages, have a

defined semantics. Given a suitable interpreter, intermediate programs can be checked for syntactic validity; simulated on a single-processor machine; or even executed in parallel. Through such testing, the programmer can be reassured that the program being derived is well-formed and behaves correctly. Methodologies that use concrete languages tend to result in an executable implementation, while those using more informal languages often generate a description of the implementation, which must then be coded.

Programmer-directed tools. The authors of many of the methodologies discuss the provision of tool support in an advisory role, rather then using the methodology as the basis of a 'black box' compilation system. Tools are proposed that perform analyses to provide guidance to the programmer, or to automate the application of transformations selected by the programmer. This indicates a belief amongst the developers of these systems that human insight and involvement in the parallelisation process is necessary for the best results.

Extensible set of primitives. The most general methodologies allows the user to extend the primitives available in the system: for example new basic modules for TwoL, adding machines to the APM hierarchy, or introducing new skeleton combinators in SAT. The methods that do not provide this facility risk being limited to a particular problem domain or target architecture.

However, the methodologies provide little guidance in the design of new primitives. Perhaps this is unsurprising: they are concerned primarily with the composition of primitive operations to form algorithms. The introduction of new primitives is below the level of these models, and requires the use of another design and implementation technique.

Comparison of the Many-Staged Systems

Table 2.1 summarises the features provided by each of the many-staged programming methodologies surveyed. As program transformation encompasses a selection of different approaches, the first column in the table describes the features of a typical system.

It can be seen that some of the methodologies exhibit more of these features than others do. Furthermore, some features are supported to a higher degree in some systems. In the borderline case this is indicated by a combined tick and cross. For instance, there are only two discrete stages in the SAT methodology and the majority of the derivation takes place in the first, the abstraction view. Similarly, general program transformation techniques can be supported by tools to perform rewriting, but the ability to perform cost analyses may be limited by the formalism used.

	Program Trans.	SAT	FAN	Aspect-Oriented	TwoL	APMs
Incremental introduction of detail	✔	✔	✔	✔	✔	✔
Discrete stages	✘	✔/✘	✘	✔	✔	✔
Fixed series of stages	✘	✔/✘	✘	✔	✔	✘
Progression by transformation	✔	✔	✔	✘	✘	✔
Executable intermediate code	✔	✘	✔	✔	✘	✔
Programmer-directed tools	✔/✘	✔	✔	✘	✔	✔
Extensible set of primitives	✔	✔	✘	✘	✔	✔

Table 2.1: Features provided by different incremental programming methods

2.3 Our Design

This section locates the PEDL programming system within the design space outlined in the previous section. The following chapter presents the main features of PEDL, while Chapter 4 gives a formal definition of the languages within the system.

PEDL is a system to produce efficient Group-SPMD parallel implementations for array based algorithms. The aim of the design of this system is to combine all the qualities identified in the previous section with a concrete and rigorous programming model to provide a clear-cut implementation route for this problem domain.

Implementation details are introduced through a fixed series of discrete stages. These stages are structured as a sequence of executable languages that allow reasoning, transformation and testing of intermediate programs. As the PEDL system is restricted to a particular parallel model and problem domain, fixing the decision stages does not constrain the generality of the system any further.

The sequence of languages provide increasing levels of control over the parallel machine. Using a different language for each stage enforces a clean separation between the stages of the method. The languages are first order functional and have a well defined semantics. They share a core set of constructs and semantics which is extended by each language with constructs that capture the decisions made at that stage.

The parallel behaviour of the languages is described using the APM framework. An abstract parallel machine is defined whose ParOps represent language constructs that introduce parallelism or communication.

An implementation is produced by transforming a program through the language sequence, which at each stage introduces implementation details. Programs can be transformed within a single stage via standard equational reasoning. As the languages share

the same semantic underpinning, constructs in adjacent languages can be equated. This allows transformations that introduce implementation decisions to be proved correct. As each of the stage languages is executable, programs can be passed to an interpreter for static checking and testing.

The result of the derivation process is a conventional imperative message-passing implementation, not just a description of this implementation. Once all parallelisation decisions have been fixed this final implementation is generated by applying a set of transformations followed by translation to the target language. The process can be automated for much of the language, although the programmer must translate by hand some of the user-defined computational code. Further tool support could be provided to support program transformation at the earlier stages of the system.

As well as producing stand-alone parallel implementations, PEDL could also be used in conjunction with some of the staged methodologies listed above. Defining an APM that describes the parallel behaviour of the language constructs allows PEDL to be utilized as an implementation route for high-level specifications produced in the APM methodology. The PEDL system could also be used to systematically implement new basic modules for TwoL. If a TwoL derivation requires a new implementation of a module with a different data distribution, all that is required is to backtrack through the decision stages of PEDL and branch at some point – which reduces the amount of work required.

2.4 Summary

Both high and low levels of abstraction are beneficial at some point in the production of a parallel program. This chapter has examined the ability of a range of programming models to combine the benefits associated with both levels. It was found that programming models based on an incremental introduction of implementation detail permit different levels of abstraction to be presented to the programmer at various stages of the implementation process.

Through comparing a selection of incremental programming systems, some important qualities of these systems were identified. The motivation for the design of PEDL is to produce a programming system with these qualities that is concrete, rigorous and provides a well-defined implementation route. The following chapter describes the features of this system in more detail.

Chapter 3

PEDL – A Staged System for Group-SPMD Programming

Capsule

This chapter introduces a system, PEDL, the design of which explores the use of staged programming techniques. The purpose of this system is to implement array-based algorithms on message-passing parallel machines in the Group-SPMD model. The result of the system is a conventional message-passing imperative implementation.

The system starts from an abstract specification of the algorithm. A series of decision stages then introduce parallelisation details. First the maximum useful parallelism is identified, then the program is mapped onto a machine with a finite number of processors. After adding communication operations the parallel program is fully specified. A final implementation is generated by a process of transformation and simplification to an intermediate form that can then be translated to the target language.

Each stage of the system is supported by languages which provide an appropriate level of abstraction. During the decision stages the program is expressed in a two-level combination of a collective-view coordination language that schedules computational blocks expressed in a processor-view language. An abstract data type with a restricted set of operations is used to ensure that distributed data is manipulated safely by the two language levels.

Introduction

This chapter introduces PEDL– a system that produces Group-SPMD parallelisations of array-based algorithms. The motivations and issues encountered during the design of the system are discussed, and its key features are presented.

The chapter starts by describing the series of stages in the PEDL system. Section 3.2 presents the structure and constructs common to all the languages of the system. At some stages of the system the program is expressed in a combination of two languages. Section 3.3 explains the benefits of this, while Section 3.4 illustrates the importance of differentiating replicated and distributed data and the technique used to do this. The languages supporting each stage are then introduced. Stages that introduce parallelisation detail are presented in Section 3.5 while the back-end of the PEDL system is sketched in Section 3.6.

3.1 Stages of the System

The PEDL system comprises a series of distinct stages that progressively introduce more implementation detail. At each stage of the derivation the program is expressed in a language or combination of languages that allow implementation decisions to be captured while abstracting from currently unspecified details. These stage languages are concrete and executable: the programmer can ensure the algorithm has been specified completely and that is behaves as expected. This is a valuable property that an informal or pseudo-code approach would not allow.

(Winstanley, 1999a) introduces the stages of the PEDL system, which are repeated in this section. The system starts with a specification of the computational component of the algorithm. Three decision stages introduce implementation detail via user-directed transformation. At this point all the parallelisation details have been fixed. The remaining stages of the system transform, simplify and translate the program to produce a conventional imperative message-passing implementation.

The choice and ordering of the decision stages of the PEDL system is similar to that proposed by previous design methods. For instance, both an informal design method for C+MPI programming (Pacheo, 1996) and the more formal TwoL methodology (Rauber and Rünger, 1995) comprise first a step where the scheduling of computations is fixed, followed by a step where processors are divided between computations executing in parallel, and finish with a step that considers the communication required to satisfy data dependencies between computations. This division is in essence the same as the PEDL

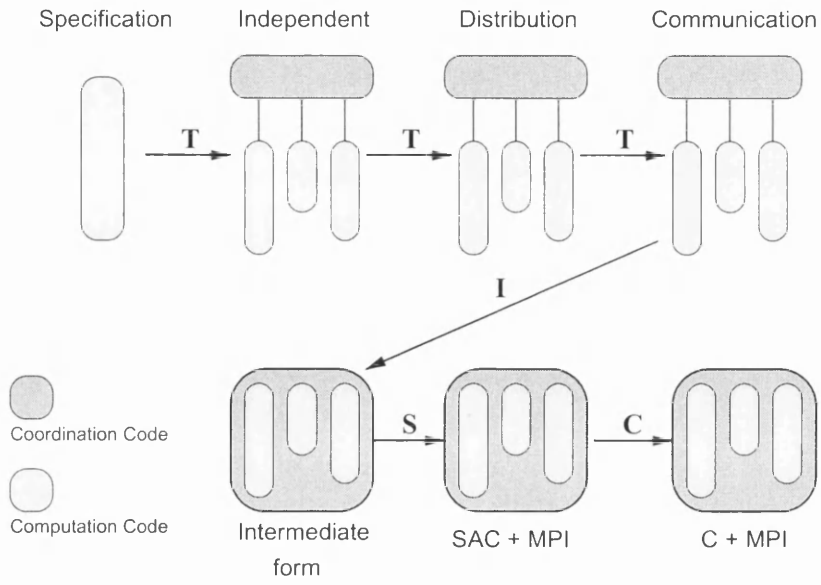

Figure 3.1: The stages of the PEDL system

decision stages. Where the current work differs is that the methodology is seated in a series of concrete languages, rather than informal notations.

This division of stages appears a sensible design, and was found suitable for the case studies that were performed. However, it is not clear that this is the only, or indeed the best, sequence of stages that could be used. This issue is examined further in Section 8.2.2

Figure 3.1 illustrates a program passing though the stages of the system and the transformations applied to it. The details of each stage in the system are as follows:

1. **Specification**. A simple language is used to specify the computation to be performed. This is a single-assignment language that contains loops, conditionals and operations to create and manipulate arrays.

2. **Independent**. In the decision stages the programmer introduces implementation detail by transforming the program. User-directed transformations are denoted as T in the figure.

   This is the first decision stage, which identifies the maximum amount of useful parallelism. Programs at this and the following two stages have a two-level structure. At the upper level is a coordination language with a collective view of the processor of the parallel machine. This level schedules blocks of computational code. These blocks are expressed in the computational language of the previous, specification, stage.

3. **Distributed**. The program is then transformed so that the coordination level program is expressed in the *distributed* stage language. At this stage the program is restricted to a fixed number of processors. The coordination program now has details of the placement and load balancing of computations.

4. **Explicit Communication**. The communication of data between processors has so far been implicit. This stage extends the coordination level program with collective communication constructs, which must be used to satisfy the redistribution requirements of the computational blocks. Once this has been done all the implementation details necessary to produce a final implementation are present in the program.

5. **Intermediate Form**. The target for the PEDL system is a conventional imperative program where communication is performed by calls to MPI. Such a language does not have a collective view of the parallel machine: the parallelism occurs 'outside' the processor-view program.

   The first step towards the final implementation is to remove the coordination level of the program by combining coordination and computational code in a residual processor-view program. This is achieved using the **Change of View Transformation**, whose application is denoted by **I** in the figure. The resulting program is expressed in the *intermediate language*. This models the features of the target language: it is simply the computational language used throughout the system extended with constructs that represent calls to the MPI communication system.

6. **SAC+MPI**. The final result of the system is a C+MPI program: this is necessary because many parallel computing environments offer little more than C and FORTRAN compilers and communication libraries. However, the language model of C is some distance from the single-assignment semantics of the intermediate language. Rather than describing the transformation between the two models by hand, we use *Single Assignment C* (SAC) (Scholz, 1994) as a bridging language. SAC is similar to the PEDL intermediate language and is compiled to ANSI C.

   The intermediate language program is unfolded and simplified. It can then be translated on a construct-by-construct basis to SAC. The SAC compiler is then used to generate the final C+MPI implementation.

3.2 Common Language Features

All the languages in the PEDL system, at both the coordination and computation layers, have the same characteristics, structure and share a common core of constructs. Important considerations in their design were that equational reasoning should be possible and that the features they provide should be easy to translate to the target imperative language. Thus the languages are strongly-typed single-assignment executable languages. In contrast to ML or Haskell, the languages have a first-order type system: functions are not first class, and so cannot be passed as parameters to constructs.

Common Structure. The languages are block structured. A block of code is made up of a sequence of computations that are executed in order. Computations may themselves be nested blocks. Each computation returns a result, which may either be bound to an identifier or discarded. Some computations return uninteresting results, and are used primarily for their side-effects. The block itself returns a result value, which is the result of the last computation in the block. The syntax and semantics of the languages is presented is detail in Chapter 4. For now we give a brief illustration of the syntax of language blocks:

$$
\begin{aligned}
bl \;=\; & \mathbf{do}\; a \leftarrow f \\
& \quad\; b \leftarrow g\; a \\
& \quad\; c \leftarrow \mathbf{do}\,\{d \leftarrow h\; a;\; k\; d\} \\
& \quad\; m\; b\; c
\end{aligned}
$$

This code snippet defines bl which comprises a block of computations. The block first executes f binding its result to a. Such results can be used as parameters to later computations, as with $g\ a$. The result returned by bl is the result of the final computation m in the block.

The identifier c is bound to the result of a nested block of computations. Identifiers bound in an outer block are visible in the bodies of later inner blocks, but not vice-versa. The structure of blocks may either be indicated using layout, or with explicit { ; } punctuation.

Common Constructs. All the languages share a common core of conditional and iteration constructs. These are not required for the purposes of this chapter and are defined in Chapter 4.

Computational Constructs. The majority of the derivation of a program focuses on implementation decisions that are recorded in the coordination layer language. Due to this separation, the particular details of the computational language are of little interest to this thesis: many language designs would be suitable. The simple computational language used in the specification stage of the PEDL system possesses the common iteration and and looping constructs and a set of operations for creating, indexing and manipulat-

```
if (rank == 2)
 send(4, val);
else if (rank == 4)
 val' = recv(2);
```

(a) Processor view

```
val' = communicate(2, 4, val);
```

(b) Collective view

Figure 3.2: Comparison of message passing in different views

ing arrays. This language forms the basis of the computational languages used in later stages.

3.3 Collective-View Programming

The decision stages of PEDL present a two-level view of the parallel computation – the upper coordination layer manages the placement and scheduling of lower-layer computational blocks which execute on a single processor. Abstract data types are used to allow the two levels to manipulate distributed data in ways that are safe for each layer.

This approach separates the description of the algorithmic computation from that of scheduling and data movement. It provides a programming model where the implementation is built up as a sequence of parallel computations. Although this 'Seq of Par' view is not suitable for all classes of algorithm, it is easier for the programmer to understand and work with as each parallel operation is self-contained: it is easier to rearrange and manipulate the parallel behaviour of the implementation.

Parallel programming in conventional languages such as C+MPI is difficult for many reasons: one of these is that the language does not provide a clear view of the entire machine. Rather than providing constructs that define behaviour over the whole machine, these languages instruct the processors individually – they have a *processor-view* of the parallel machine. As well as making it difficult to understand the behaviour of the machine, this is a new cause of errors.

Figure 3.2 illustrates one of the problems of processor-view parallel programming. To communicate a message requires two procedure calls – a send and the corresponding receive. If the communication operations are not paired, the result will be deadlock or erroneous behaviour. It is common for the calls to the two procedures to be separated in the source code. A simplified example of this is shown in *(a)*; however it is often the

case that the two calls are separated by significantly larger blocks of code. Thus there are linkages and dependencies between branches of the code that are not made explicit in the language itself. Unfortunately the compiler cannot check for correctness because it has no information about the relationship between the two library calls.

A better solution would be to have a communication construct that extends over all the processors involved as in *(b)*. Linkage is no longer a problem as one construct performs the send and receive operations. As the call-sites for the two ends of the communication are in the same position the comprehension is also improved: it is easier to follow the flow and processing of data across the machine.

The idea of operations that scope over all processors in the machine can be extended from communication to other language constructs. The constructs of the language express behaviour in which all processors participate in some regular way. Programs written in such a language do not execute on any one processor, but present a collective view of the behaviour of a partition of processors. Such a style of language – where constructs scope over a set of processors – we call a *collective-view* language. Code that initializes the system or creates replicated data can more concisely be expressed in a language with a collective view.

Most algorithms will at some point need to escape from the collective view so that each processor can be instructed independently. For more task-parallel styles of implementation, each processor may be required to perform arbitrarily different computations. For this purpose the collective-view coordination language provides *hook* constructs that execute a block of processor-view computational code on every processor in the machine. This provides a uniform entry and exit point to the arbitrary computational code.

When calling a processor-view computation it is desirable to ensure that the code is safe and well-behaved: it must not alter the configuration of the parallel machine or attempt to communicate with other processors. To this end, the language used to express processor-view blocks has no parallelism-related constructs – it may only manipulate local data structures. However, the computational block may query the rank of the processor it is executing upon: this allows different routines to be executed on each processor.

This link between the two language levels fits well with the SPMD model. The processor-view code performs a computational step and an implicit synchronisation, while the collective view expresses the communication step and any initialization or management of the machine configuration.

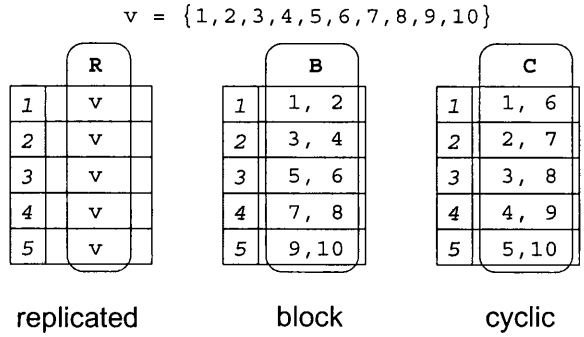

Figure 3.3: Example data distributions for a ten-element vector v over five processors

3.4 Replicated and Distributed Data

Parallel algorithms make use of two kinds of parallel data structure. The first is distributed data, where processors each have a local copy of a subset of the total data. The other is replicated data: here each processor has a copy of the entire data structure. Figure 3.3 illustrates different distributions of a vector of ten elements v over a partition of five processors. In a replicated distribution, all elements of the vector are resident in each of the processors' local memories. Unlike replicated data, distributed data can be divided between processors in many different patterns. Two of the commonest are the block and cyclic decompositions. A block decomposition allocates a continuous range of elements to each processor, while a cyclic decomposition 'deals out' the elements between processors so that adjacent elements are on adjacent processors.

While distributed and replicated data are used to achieve a speedup, replicated data is also used to control the behaviour of the parallel algorithm. Some of this control data is static: for example parameters such as problem size, number of processors and array bounds. Other control data is dynamic. For example a parallel search algorithm may terminate when a processor finds an acceptable solution. If a match is found by one processor the others must be informed. The simplest solution would be to broadcast a message to the other processors: this replicates the dynamic data throughout the machine.

In the two level model of our system, distributed data is generated and consumed by blocks of computational code but manipulated and marshalled in the coordination level. Replicated data may be used as parameters to coordination language constructs or within computational code.

```
for (1, ≤ x, +1)
  parallel{
    ⟨computational block⟩
}
```

(a) Coordination program

```
for (i=1;i<=x;i++){
 /* computational
    block */
}
```

(b) SAC+MPI translation

Figure 3.4: Example translation from the coordination language to C+MPI

3.4.1 A Problematic Example

As explained previously, a program written in the coordination language does not reside on any one processor, but instead manipulates a partition of processors. However the target language for the derivation, SAC+MPI, does not have a machine model suited to executing such a language. To produce the final implementation the coordination component of the program must be factored into the computational code for each processor. This results in a replication of the control structures and collective constructs of the coordination program across all processors in the partition.

An example of this is given in Figure 3.4. The coordination program *(a)* consists of a **for** loop which repeats x times. The loop body is a **parallel** construct parameterized by a computational block. This is one of the hook constructs mentioned in the previous section: **parallel** causes its parameter block of computational code to be executed in parallel on every processor in the partition.

The corresponding C+MPI implementation *(b)* would be a `for` loop which iterates x times. The coordination level has been combined with the computational code by transforming x iterations of a parallel computation into the parallel execution of a computation which iterates x times on each processor.

This illustrates a problem. *The transformation is only valid if x has the same value on every processor.* x must be a replicated, not a distributed value. To apply such transformations reliably it is essential that replicated and distributed data can be distinguished: in short we must control the occurrence of distributed data in the coordination language. Before explaining how this is done, we introduce the language constructs that generate distributed data.

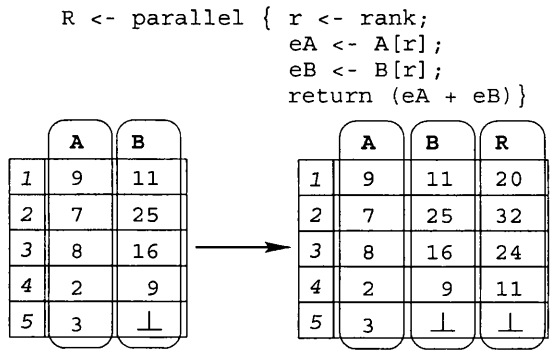

Figure 3.5: A parallel sum of decomposed values A and B on a partition of five processors

3.4.2 Generating Distributed Data

The main means of introducing parallel computation that generates distributed data is by using the **parallel** construct. Tthis is one of the *hook* constructs that link the collective-view coordination language and the processor-view computational language. It takes a block of computational code as a parameter and executes this in parallel on every processor in the current partition. The computational language provides two constructs to access environmental information: **size** returns the size of the partition, while **rank** gives the rank in the partition of the processor the computational block is executing on. The results of the processor computations are returned by the **parallel** construct back to the coordination language as a sequence indexed by *processor identifier* (PID).

Figure 3.5 illustrates a parallel computation that combines two distributed values A and B that are are simply distributed element-wise across the partition. In this example, the parallel computation block accesses the elements of the distributed values that are local to that processor and returns their sum. As the block is executed on every processor in the partition in parallel, this performs an parallel element-wise sum of the distributed values. The result returned in the coordination language is another distributed value R, which has the same distribution as A and B.

It is possible for a distributed value to be unevenly distributed over a partition, so that for some processors it is undefined. In the figure undefined elements are represented by $\perp$. Any processor computations involving undefined elements are themselves undefined: however this does not affect other element computations or the termination of the **parallel** construct.

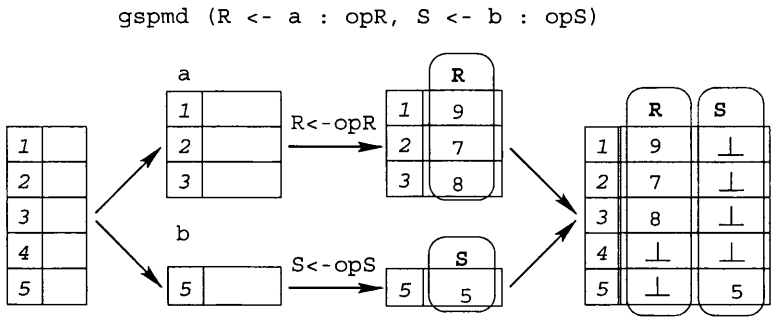

Figure 3.6: Concurrent computations on child partitions

3.4.3 Partitioning

The **parallel** construct performs a SPMD computation step. The other method of introducing concurrent computation in the Group-SPMD model is to divide the partition into independent child partitions. This is done by the **gspmd** construct. It is parameterized by descriptions of the partitions to create and a block of coordination code to execute on each.

The interaction of child partitions is restricted according to the Group-SPMD methodology – in particular no communication or synchronization is permitted between them. The child partitions execute independently of one another and each returns a PID-indexed sequence of results to the parent partition. The coordination program of the parent partition resumes once all child partitions have terminated.

Figure 3.6 illustrates the operation of the **gspmd** construct. In the example, two computations `opR` and `opS` are to be executed concurrently. The `a` and `b` parameters define which processors of the parent partition form each child partition. Each partition then independently executes its assigned coordination block; this produces an indexed set of results on the sub-partition. Once the computations of the child partitions have terminated, the partitions coalesce together again. The results computed on the child partitions are available for use in the parent partition program. These decomposed values have defined elements for those processors which participated in the child partition: the other elements are undefined.

3.4.4 An ADT for Distributed Data

Section 3.4.1 described the need to distinguish between replicated data and distributed data and to control where each may be used. Distributed data can only be introduced into the coordination language by the **parallel** and **gspmd** constructs. Our solution is to

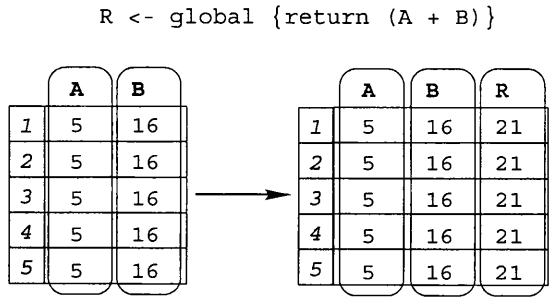

Figure 3.7: A global sum of replicated values A and B on a partition of five processors

encapsulate the sequence of results returned from such computations as an abstract data type. We call this type a *distributed value* (DVal).

The use of distributed data is controlled by restricting the operations available on the DVal type. In particular the coordination languages lack the ability to project elements from this abstract type: therefore elements of distributed data cannot be used as parameters to coordination language constructs.

Meanwhile the computational language does have operations to project elements of a DVal: this is necessary as the elements are the results of previous computations. The construct **use** accesses the element of a DVal resident on the current processor. Rewriting the code from Figure 3.5 so that it uses the DVal ADT gives:

$$
\begin{aligned}
R \leftarrow \textbf{parallel}\ eA &\leftarrow \textbf{use}\ A \\
eB &\leftarrow \textbf{use}\ B \\
&\textbf{return}\ (eA + eB)
\end{aligned}
$$

Although computational blocks may unpackage the elements of a DVal, this data cannot escape back into the coordination layer: the result of a **parallel** computation is encapsulated in a new DVal. In this way, the DVal ADT provides a safe interface between the different views of the two language layers. The coordination layer has a collective view of the distributed data structures while the computational language can access individual elements of these structures.

3.4.5 Generating Replicated Data

Distributed data is controlled by packaging it in an abstract data type and restricting the operations available on this type. However, coordination layer constructs require unpackaged data as parameters – this cannot be generated by a **parallel** block because the result will be inaccessible; while arbitrary computation is disallowed in the coordination language.

Such data can be generated using the other coordination language hook construct – **global** (Figure 3.7). This executes a block of computational code to produce a replicated result. The computational block may refer to previous replicated values, but cannot project elements from a **DVal** or query the index of the processor it executes on – that is the **use**, **get** and **rank** constructs are disallowed. As these are the only two ways in which a computational block may generate a result that differs over processors, a replicated result is guaranteed. Therefore a single result may be returned to the coordination language, without the need for encapsulation in a **DVal**. Such data can be safely used in later coordination layer or computation later code.

Sometimes it is necessary to use a particular element value of a **DVal** as a parameter in the coordination language. In these cases the coordination language construct **globalize** is used – this performs a broadcast communication to dynamically replicate the value and makes the representative element visible.

3.5 Decision-Making Stages

This section presents each of the the decision stages and their supporting languages.

3.5.1 Independent Computation Stage

The *Independent Computation* stage is the first decision stage of the system. It introduces a coordination language which is used to identify the potential parallelism of the algorithm. The language can express the computation of decomposed and replicated values and the execution of concurrent child partitions using the constructs **parallel**, **global** and **gspmd**. However, it does not specify the placement of computations on processors or communication.

A program in this coordination language executes on a partition described simply by an integer size – the number of processors in the partition. The processors are given ranks numbering from one to this size.

Distributed Values. Decomposed values are encapsulated in a **DVal** whose elements are indexed by processor ranks. This language has a shared memory machine model in which the computational result of one processor can be directly accessed by subsequent computations on other processors without explicit communication constructs. The local element of a distributed value can always be accessed in a computational block by the **use** construct. To access elements of a distributed value computed by another processors the **get** construct is used: this takes a **DVal** and the processor rank of the element to

project. This means that distributed values can be treated rather like arrays, where **get** is the indexing operation.

Partitioning. As this decision stage is intended to identify the maximum potential parallelism, child partitions are not limited to sub-dividing the processors of the current partition. Instead an unlimited number of processors are available; the partitioning construct **gspmd** can create partitions that are as large as required. Partitions are described by a single integer representing their size. A **DVal** is returned from a child partition with elements indexed up to this partition size.

3.5.2 Distribution Stage

The next decision stage starts to introduce the constraints of the target parallel machine. The *Distributed* coordination language has a machine model of a fixed number of processors and they have an identity: each is named by a *processor identifier* (PID). A partition is described by a new data type – the **Group**. Values of this type define a mapping between logical rank in the partition and absolute PID. This abstraction is more convenient than programming with PIDs directly, as it gives a contiguous ordering to the processors and also a degree of machine independence.

Rewriting the program in this language requires load balancing and granularity adjustments so that the processors of the machine are evenly utilized. Once the program is in this language the decomposition of data and computation is fully specified. However communication still remains implicit.

Distributed Data. The constructs from the previous language for creating and accessing decomposed data are retained unchanged. The **use** and **get** commands still work by processor rank. The ranks are mapped through the underlying group that describes the current partition. This converts ranks to PIDs which are then used to index the **DVal**.

Partitioning. In this language the processors in child partitions created by the **gspmd** construct are defined by a group value. To compute new values of this type, the computational language is extended with a construct **currentGroup** that returns the underlying group that defines the current partition. A set of operations on the Group type are also provided that manipulate and create new groups,

The most significant change to this language is the limitation to a fixed number of processors. Therefore child partitions must be located on a subset of the processors of their parent. Similarly, a processor can only be in one concurrently executing partition. Therefore the machine resources must be managed by balancing the load of the different SPMD computations.

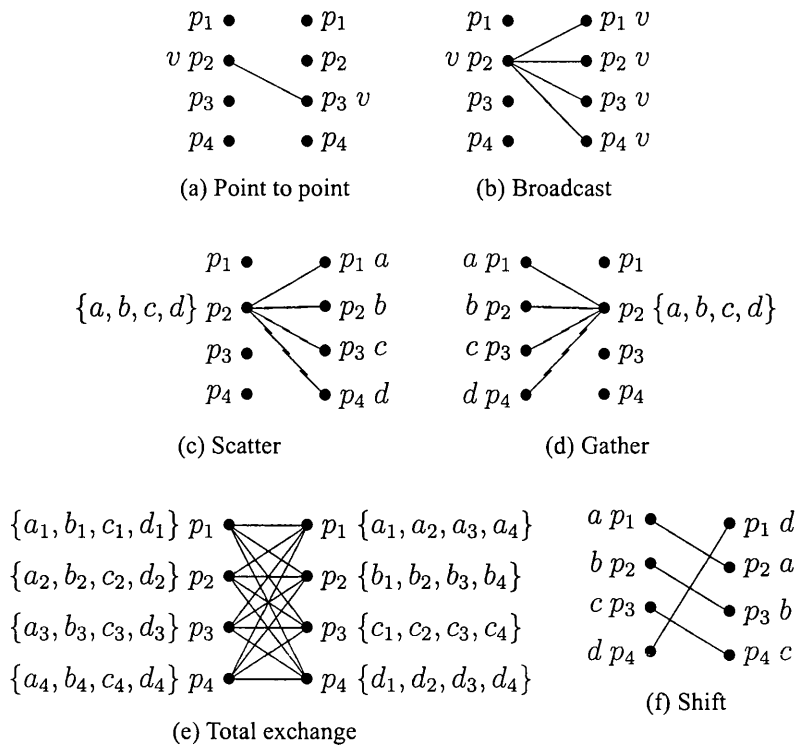

Figure 3.8: Collective Communications

A group defining a child partition may give a different mapping between ranks and PIDs than its parent. This can be used to describe different orderings and topologies for sub-computations. However, as the elements of a DVal are accessed using the rank ordering of the *current* partition, the parent partition needs no knowledge of the ordering used by the child partition to access elements of the DVal it returns.

3.5.3 Explicit Communication Stage

The final decision stage makes explicit the communication patterns required by the implementation. The supporting language disallows the get construct. Therefore the unstructured non-local DVal accesses previously expressed by this construct must now be written in a different way – by using collective communications.

The language provides a typical selection of communication primitives, based upon the informal consensus within the parallel programming community as to which primitives are necessary for a communication library. Many different systems, such as TwoL, MPI and iCC (Mitra et al., 1995) (a communication library for the Paragon), propose similar sets of collective operations. Figure 3.8 illustrates the behaviour of the primitives provided by PEDL for a partition of four processors p_1, p_2, p_3, p_4. The set comprises:

point-to-point One processor passes a single message to another processor.

broadcast One processor passes the same message to every processor in the partition.

scatter One processor passes a individual message to every processor in the partition. This primitive is typically implemented so that an array of messages is passed to it: the first element is intended for the first ranked processor in the partition, and so on.

gather The reverse of scatter. Every processor in the partition passes an individual message to a single destination processor. Implementations typically return an array of messages on the destination processor.

total exchange Every processor in the partition performs a scatter simultaneously. Therefore a unique message is passed between all pairs of processors in the partition.

shift Every processor in the partition passes an individual message to a processor whose rank is the specified offset from it. The offset can be positive or negative. This primitive can be periodic: where the shifting action 'loops round' from the end of the processor ranks to the start again; or non-periodic: where some processors will receive null values instead.

These collective communication operations reside in the coordination language. All have the same general form: in addition to some configuration parameters, they accept a *communicator* and a DVal of elements to communicate and return a new DVal where these elements have been reordered.

A communicator is a concept taken from MPI. It defines a communication context: messages from different communicators cannot be accidentally intercepted by communications from other communicators. It also defines the set of processors which participate in the communication and an ordering for them. In the PEDL system a communicator is an abstract data type, created in the computational language from a group. This group gives the ordering used within the communicator.

The choice of communication operations is shaped somewhat by those provided by MPI, the initial implementation target. However, they need not be restricted to this set – any communication between a set of processors can be encoded as a permutation function operating on a DVal.

Once this stage of the derivation is completed, the program contains details of the computations, their ordering and placement on named processors, and the communication operations required to redistribute data between them. All the information needed to produce an implementation has been provided.

3.6 Final Implementation

After all the implementation details have been fixed, the back end of the PEDL system takes the derived program and generates a SAC+MPI implementation which is then compiled to C+MPI.

Single Assignment C (SAC) (Scholz, 1994; Scholz, 1998) is a strict, purely functional language whose syntax in large parts is identical to that of C. It differs from C in that it disallows global variables and pointers to keep functions free from side-effects, and allows multiple return values from user-defined functions. One of the motivations for the SAC research is the efficient compilation of array-based computation. The language possesses a rich set of array operations that allow the programmer to write dimension-independent array expressions. The heavily optimising compiler generates code comparable in performance to FORTRAN (Scholz, 1997).

We use SAC as a bridging language between our system and C+MPI. This simplifies the transformations required to produce the final implementation as the SAC compiler converts from a functional to an imperative language model and generates conventional, portable C code. Although the SAC language model is closer to the PEDL languages than C, there are still differences. The most significant is that it only presents a processor-view model of parallel computation.

The next stage is to combine the coordination and computation layers of the program using the change of view transformation (Section 7.1) to produce a program in an intermediate language. This has a processor-view computational model and the functionality of a subset of SAC+MPI.

The intermediate language is an extension of the computational language used throughout the system. It provides the core computational operations and constructs that manipulate groups; communicators; and access information about the parallel environment. This language also has processor-view equivalents of the collective view communication operations. These mirror the functionality of the MPI calls available from SAC. As it is single-level language, there is no need for the encapsulation provided by the DVal ADT. Therefore the constructs that access elements of distributed values are not present in the language.

3.7 Summary

This chapter has given an informal overview of the PEDL system. It comprises a series of decision stages followed by a series of transformations that result in a SAC+MPI program, which is then compiled to C+MPI. At each stage the program is expressed in a different language with a level of abstraction appropriate for that stage.

During the decision stages, where the programmer is introducing parallelisation detail, the program is expressed in a combination of a coordination language and a computation language. Each of these layers present a different view of the parallel machine, appropriate for the purpose of that language. Combining different views of the machine requires a technique to control the movement of data across the boundary between the views. An abstract data type, the DVal, is used for this. It provides a different set of operations to each language level.

The following chapter gives a formal definition of the syntax and semantics of the PEDL languages. It describes the parallel behaviour of the coordination level constructs and then demonstrates that it is possible to reason formally about programs in these languages.

Chapter 4

Formal Definition of PEDL

Capsule

This chapter gives a definition of the syntax, semantics and parallel behaviour of the PEDL languages.

An operational semantics is defined for the languages of the decision stages of the system. The languages share a common core of features and constructs. These can be factored out of the languages and so presented only once. As well as simplifying the presentation, the commonality allows code and theorems to be produced that can be applied in many different contexts.

A different technique must be used for the intermediate language, as it does not fit well within the operational semantic framework used for the other languages. A transformational semantics is given that produces an equivalent collective-view program for an intermediate language program.

The parallel behaviour of the coordination layer constructs is described using Abstract Parallel Machines. In addition to documenting the behaviour of the constructs, this technique could be used to provide cost models for PEDL, and makes it possible for the PEDL system to be used as an implementation route for APM derivations.

The operational semantics can be used to reason about computations expressed in the same or adjacent languages. The chapter concludes with an example equivalence proof which uses evaluation functions defined by the operational semantics to show that two computations have the same effect on the machine.

Introduction

The PEDL system produces a parallel implementation by transforming a specification through a series of progressively more explicit languages. To prove that the transformations applied are correct, we must be able to reason about program fragments within the same language, and also equate program fragments in adjacent languages. This chapter gives a formal definition of the PEDL languages that can be used as a foundation for reasoning and performing proofs.

The first section defines the syntax and semantics of the languages. An operational presentation of the semantics is given for the languages of the decision stages, while the intermediate level language is described using a transformational semantics. Section 4.2 describes the parallel behaviour of the coordination-level languages by defining an Abstract Parallel Machine (O'Donnell and Rünger, 1997b) where each construct is represented as a ParOp. Finally Section 4.3 demonstrates how the operational semantics can be used to reason about programs expressed in the PEDL languages. Equivalence proofs can be performed by showing that the semantic evaluation functions for two program fragments have the same effect on the machine state.

4.1 Syntax and Semantics of the PEDL Languages

The PEDL languages are implemented by embedding them within a host language. A pure lazy functional language, Haskell, is used as the host; the PEDL language constructs are implemented within it as combinators. This implementation is described in detail in Chapter 5. Embedding within a host language simplifies the design, implementation and description of a language. Many of the features of the host language can be reused, so that it is only necessary to implement the novel features of the embedded language.

The language specification given in this chapter concentrates on the higher-level portion of the languages: mainly the constructs and concepts of the coordination layer. Other details – for instance the syntax and semantics of expressions and function definitions – are not handled here. Such details are as defined by the host language.

Furthermore, much of the static semantics and type system is omitted. For each construct presented, we indicate what types are expected for its parameters. How this is enforced and verified is dependent upon the host language. In our current implementation, the languages are strongly-typed and statically checked. This is achieved by representing the constructs in the Haskell type system. The type system is also used to ensure that the PEDL languages are composed together correctly.

The PEDL language syntax used throughout this thesis is a lightly-sugared version of the Haskell implementation of the languages. The syntax of the languages is presented in standard BNF. Language keywords are written in **bold** while non-terminals are capitalized in angle brackets like ⟨THIS⟩.

The operational semantics of the PEDL languages were first developed and checked by implementing an abstract interpreter within Haskell. The sources and description of this model are available from (Winstanley, 2000b). For clarity, the semantics are presented in this chapter in a more mathematical form. The main addition is an explicit set type $\{a\}$, with set comprehensions and other set operations. While such operations can be approximated using the list-handling capabilities of Haskell, a distinct type clarifies where there is no ordering or sequencing implied in the semantics.

This section first presents the supporting functions and types used within the definition of the languages. The common language structure and groups of constructs shared by classes of language are then presented (Section 4.1.2). This is followed by the specification of the constructs particular to each stage of the system (Section 4.1.5).

Sharing constructs between languages simplifies the design, implementation and specification of the languages. It also permits the programmer to write generic procedures and to formulate transformations and theorems that may be used in a range of settings. However it is still possible to distinguish between the different languages; the embedded implementation achieves this through the use of *phantom language types* (Section 5.3).

4.1.1 Language Basics

We start by defining the underlying type and functions used throughout the presentation of the semantics.

Environments

The machine state is modeled using the following types and function. These definitions manipulate tuples, for which we provide some primitive operations. The domain of a set of tuples s can be accessed using **dom** s. Similarly, the codomain, or range, of a set of tuples can be accessed using **codom** s. The notation $s(n)$ maps a value through a set of tuples. The value n must occur in the domain of s: the associated value in the codomain of s is returned.

$$
\begin{array}{ll}
\mathit{State} & \triangleq (\mathit{World},\ \mathit{Store}) \\
\mathit{Store} & \triangleq [\{(\mathit{Identifier},\ \mathit{Value})\}] \\
 & \\
\mathit{newState} & :: \mathit{World} \rightarrow \mathit{State} \\
\mathit{popframe} & :: \mathit{State} \rightarrow \mathit{State} \\
\mathit{pushframe} & :: \mathit{State} \rightarrow \mathit{State} \\
\mathit{bind} & :: \mathit{Identifier} \rightarrow (\mathit{Value}, \mathit{State}) \rightarrow \mathit{State} \\
\mathit{findValue} & :: \mathit{Identifier} \rightarrow \mathit{State} \rightarrow \mathit{Value} \\
 & \\
\mathit{newState}\ w & = (w,\ []) \\
\mathit{popframe}\ (w,\ (f : fs)) & = (w, fs) \\
\mathit{pushframe}\ (w,\ fs) & = (w,\ \{\} : fs) \\
\mathit{bind}\ n\ (v,\ (w,\ (f : fs)) & = (w,\ (f \uplus \{(n,\ v)\}) : fs) \\
\mathit{findValue}\ n\ (w, []) & = \mathit{undefined} \\
\mathit{findValue}\ n\ (w,\ (f : fs)) & = \textbf{if}\ n \in \textbf{dom}\, f \\
 & \quad \textbf{then}\, f(n) \\
 & \quad \textbf{else}\ \mathit{findValue}\ n\ (w, fs)
\end{array}
$$

The *State* of the parallel machine is represented as a tuple of an abstract representation of the external world state, and the store. The *Store* associates identifiers with values. It reflects the block structure of the language: it is a sequence of frames. Each frame is a set of bindings between identifier and value. A fresh frame is added to the store whenever a new block of the program is entered; this frame is removed once execution of the block is completed.

The program starts executing in a state with an empty store. The *popframe* and *pushframe* functions add and remove a new frame to the store. A new binding between a name and value can be added to the most recent frame using *bind*. Within the definition of *bind* the $\uplus$ symbol denotes overriding union – that is, the new binding overrides any previous binding involving that identifier in the frame.

The *findValue* operation finds the value associated with an identifier. Notice that when bindings for an identifier occur in more than one frame, the value from the most recent binding is returned. This corresponds to the lexical scoping of block-structured languages, where the innermost binding gives the current value of an identifier. If no binding can be found for an identifier, the value returned is undefined.

Worlds

Some computations do not return useful results – instead they perform IO that affect the state of the external world. To retain a referentially-transparent presentation of the semantics, this external state is modeled using an abstract type called 'World'. Provided the world state is never duplicated or stored the program will be single-threaded, which allows equational reasoning to be applied. This is similar to the approach used in

Clean (Brus et al., 1987) – where states are represented by variables whose single-use is enforced using uniqueness type attributes (Smetsers et al., 1994).

This type is merely a device for preventing errors during reasoning and transformation. It does not give any specification of the behaviour of IO-performing computations.

$$
\begin{array}{lll}
\mathit{World} & \triangleq & \ldots \\
\mathit{doIO} & :: & \mathit{World} \rightarrow \mathit{World} \\
\mathit{mergeWorld} & :: & \{\mathit{World}\} \rightarrow \mathit{World} \\
\mathit{splitWorld} & :: & \mathit{World} \rightarrow \{\mathit{World}\}
\end{array}
$$

$$
\begin{array}{ll}
\forall\, w :: \mathit{World} & \cdot\, w \neq \mathit{doIO}\, w \\
\forall\, \mathit{ws} :: \{\mathit{World}\} & \cdot\, \mathit{mergeWorld}\, \mathit{ws} \notin \mathit{ws} \\
\forall\, w :: \mathit{World} & \cdot\, w \notin \mathit{splitWorld}\, w
\end{array}
$$

The evaluation rules for computations performing IO consume the current world state and produce a new one. This is encapsulated in the *doIO* primitive; no definition is given, merely an axiom stating that it never returns the same world value.

We also define operations that split and combine the world value. This is necessary because of the change of view that occurs when control passes from the coordination layer to a block of parallel computational code. Such a block of code may cause each processor to perform IO. The world value is decomposed using *splitWorld* so that each processor has a portion of it on which to operate. After the parallel computation has completed the (possibly altered) processor world values are combined using *mergeWorld* to produce a new collective-view model of the world.

This splitting and merging is quite sensible provided that different processors do not attempt to access the same IO resource during a computation step. If this happens the outcome is undefined, as synchronisation is impossible in a single computation step. Furthermore, such behaviour breaks the SPMD programming model. An example of legal IO is where each processor writes to a different file simultaneously – these are disjoint parts of the collective world state that do not interact with one another. However, having one processor reading and another writing to the same file in a single step is nondeterministic – as there is no way to synchronize the two operations in the computational model.

For convenience the world values are threaded through the semantic equations as part of the *State* value, which also contains the environment of the executing program. To prevent extensive packing and unpacking later in the semantics, we define merge and split functions that operate over this state type as follows:

$$
\begin{array}{lll}
\mathit{mergeState} & :: & \{\mathit{State}\} \rightarrow \mathit{State} \\
\mathit{splitState} & :: & \mathit{State} \rightarrow \{\mathit{State}\} \\
\mathit{mergeState}\ \mathit{ws} & = & (\mathit{mergeWorld}\ (\mathbf{dom}\ \mathit{ws}),\ e) \qquad \textbf{where}\ e \in \mathbf{codom}\ \mathit{ws} \\
\mathit{splitState}\ (w, e) & = & \{(w', e) \mid w' \in \mathit{splitWorld}\ w\}
\end{array}
$$

These definitions also specify what happens to the store value when moving back and forth between the processor view and the collective view. Moving to the processor view (*splitState*) replicates the current environment on each processor. When recombining, the environment of an (unspecified) processor is selected to form the environment of the coordination level. This leads to non-deterministic behaviour unless the environment on every processor is identical. However, this is the case once the processor blocks have completed, as any bindings that had been introduced by them were stored in fresh frames, which are subsequently popped off the environment.

4.1.2 Program Blocks

All the PEDL languages are block structured. A ⟨PROGRAM⟩ is made up of a block of code, combined with an integer which gives the number of processors that the starting partition should contain. A ⟨BLOCK⟩ consists of a sequence of one or more computations, where the return value of the final computation forms the return value of the entire block. Each computation in the block is an action, whose result may be bound to a new identifier in the environment or simply discarded.

⟨PROGRAM⟩ ::= **run** ⟨BLOCK⟩ ⟨IDENT⟩
⟨BLOCK⟩ ::= { ⟨COMPUTATION⟩$^+$ }
⟨COMPUTATION⟩ ::= ⟨IDENT⟩ ← ⟨ACTION⟩ ;
| ⟨ACTION⟩ ;

The ⟨ACTION⟩ syntactic class contains all other language constructs: this varies from language to language. Therefore ⟨ACTION⟩ it is left undefined for now, and a separate definition given for each language in turn.

Semantics

We now present the semantic equations for the syntactic classes introduced so far. The evaluation function $[\![\cdot]\!]$ gives the result of evaluating a construct in the current machine state. In most cases, the equation of $[\![\cdot]\!]$ to choose can be determined by the concrete syntax passed to it. However, in cases where this is ambiguous or unclear, the function is annotated with the name of the syntactic class.

In addition to the syntactic clause, the evaluation function commonly takes two further parameters The first, d, is a parameter containing environmental information about the parallel machine the program is executing upon. This will is examined in detail when the execution rules for parallelism constructs are presented (Section 4.1.4). Until then, it can be ignored. The other parameter, e is the representation of the machine state introduced earlier.

The evaluation function typically returns a tuple of a result value, and a new machine state. The machine configuration parameter d is never returned: it consists of 'read-only' data that may only be accessed by evaluation functions but not modified. Some of the evaluation equations differ in their parameter or return types. Such cases will be indicated as they occur.

$$
\begin{aligned}
[\![\mathbf{run}\ \mathrm{B}\ i]\!] &= \lambda\, w \rightarrow [\![\mathrm{B}]\!]_{\mathbf{block}}\ (mkGroup\ i)\ (newState\ w) \\
[\![\{\mathrm{C}\}]\!]_{\mathbf{block}}\ d\ e &= \mathbf{let}\ (v, e') = [\![\mathrm{C}]\!]_{\mathbf{computations}}\ d\ (pushframe\ e) \\
&\quad \mathbf{in}\ (v, popframe\ e') \\
[\![\mathrm{A}]\!]_{\mathbf{computations}}\ d\ e &= [\![\mathrm{A}]\!]_{\mathbf{computation}}\ d\ e \\
[\![\mathrm{B}\ ;\ \mathrm{A}]\!]_{\mathbf{computations}}\ d\ e &= \mathbf{let}\ (v, e') = [\![\mathrm{B}]\!]_{\mathbf{computation}}\ d\ e \\
&\quad \mathbf{in}\ [\![\mathrm{A}]\!]_{\mathbf{computations}}\ d\ e' \\
[\![n \leftarrow \mathrm{A}]\!]_{\mathbf{computation}}\ d\ e &= \mathbf{let}\ (v, e') = [\![\mathrm{A}]\!]\ d\ e \\
&\quad \mathbf{in}\ (v,\ bind\ n\ (v, e')) \\
[\![\mathrm{A}]\!]_{\mathbf{computation}}\ d\ e &= [\![\mathrm{A}]\!]\ d\ e
\end{aligned}
$$

The first equation above states that the effect of executing a program on a set of processors is the effect of executing the top-level block B in the state $newState\ w$. This is a state formed from an empty store and the currently available world value. The parallel configuration is generated using the *mkGroup* function (defined in Section 4.1.4): this takes as input the number of processors required by the program.

A block has its own nested scope. Executing a block involves adding a fresh frame to the environment, executing the computations of the block within this new frame, and then popping the frame off the environment. Each computation in the block may bind its result to an identifier in the new stack frame. Alternatively, the result may be discarded; this is useful for computations that return no significant result and are called for their effect on the external world.

Any new bindings can be accessed by subsequent computations. The result returned from a block is the result computed by the final computation within it. As the temporary frame is removed on completion of the block, the result returned from it is the only property of the block observable from the parent program.

4.1.3 Control Structures

There is a common core of control structures that appear in all the PEDL languages with identical semantics. This section examines these constructs.

$$
\begin{aligned}
\langle \text{STRUCTURE} \rangle ::= \; & \textbf{return } \langle \text{EXP} \rangle \\
| \; & \textbf{if } \langle \text{EXP} \rangle \textbf{ then } \langle \text{BLOCK} \rangle \textbf{ else } \langle \text{BLOCK} \rangle \\
| \; & \textbf{do } \langle \text{BLOCK} \rangle \\
| \; & \textbf{repeat } \langle \text{BLOCK} \rangle \\
| \; & \textbf{repeatn } \langle \text{EXP} \rangle \; \langle \text{BLOCK} \rangle \\
| \; & \textbf{repeataccum } \langle \text{EXP} \rangle \; \langle \text{ABS1} \rangle \\
| \; & \textbf{repeatnaccum } \langle \text{EXP} \rangle \; \langle \text{EXP} \rangle \; \langle \text{ABS1} \rangle \\
| \; & \textbf{for } (\; \langle \text{EXP} \rangle \; , \; \langle \text{EXP} \rangle \; , \; \langle \text{EXP} \rangle \;) \; \langle \text{ABS1} \rangle \\
| \; & \textbf{foraccum } (\; \langle \text{EXP} \rangle \; , \; \langle \text{EXP} \rangle \; , \; \langle \text{EXP} \rangle \;) \; \langle \text{EXP} \rangle \langle \text{ABS2} \rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![\textbf{return E}]\!] \; d \; e &= ([\![\text{E}]\!] \; e, \; e) \\
[\![\textbf{if E then B1 else B2}]\!] \; d \; e &= \begin{cases} [\![\text{B1}]\!]_{\textbf{block}} \; d \; e, & \text{if } [\![\text{E}]\!] \; e = \mathit{true} \\ [\![\text{B2}]\!]_{\textbf{block}} \; d \; e, & \text{if } [\![\text{E}]\!] \; e = \mathit{false} \end{cases} \\
[\![\textbf{do B}]\!] \; d \; e &= [\![\text{B}]\!]_{\textbf{block}} \; d \; e
\end{aligned}
$$

The **return** construct binds the result of an expression to an identifier in the current store. This can be thought of as a let-binding or local definition. The evaluation equation for expressions is not specified here – it is part of the host language. As such, it cannot access the parallel-machine data, and so is only parameterized by an environment. Furthermore, as the expression must be pure, the environment is unchanged.

The conditional statement is quite straightforward. Depending on whether the boolean expression evaluates to True or False, one of the two alternative blocks is executed in the current state. Next, the **do** statement introduces a child block. This is useful to control the scoping of local variables. The child block is executed as described in the previous section.

$$
\begin{aligned}
[\![\textbf{repeat B}]\!] \; d \; e &= [\![\textbf{repeat B}]\!] \; d \; (\mathit{snd} \; ([\![\text{B}]\!]_{\textbf{block}} \; d \; e)) \\
[\![\textbf{repeatn E B}]\!] \; d \; e &= \textbf{let } \mathit{eval} \; 1 \; e = [\![\text{B}]\!]_{\textbf{block}} \; d \; e \\
&\qquad\qquad \mathit{eval} \; i \; e = \mathit{eval} \; (i - 1) \; (\mathit{snd} \; ([\![\text{B}]\!]_{\textbf{block}} \; d \; e)) \\
&\qquad \textbf{in } \mathit{eval} \; ([\![\text{E}]\!] \; e) \; e
\end{aligned}
$$

The other shared constructs are various forms of loop combinator; these mirror the different iteration patterns possible in an imperative language with loop constructs. The first, **repeat**, executes its loop body infinitely; a possibly more useful construct is **repeatn** which executes a block a fixed number of times, returning the result of the final iteration.

$$
\begin{aligned}
[\![\textbf{repeataccum}\ \text{E B}]\!]\ d\ e &= \textbf{let}\ \mathit{eval}\ (v, e) = \mathit{eval}\ ([\![\text{B}]\!]_{\textbf{abs1}}\ d\ v\ e\ ) \\
&\quad\ \textbf{in}\ \mathit{eval}\ ([\![\text{E}]\!]\ e,\ e) \\
[\![\textbf{repeatnaccum}\ \text{E E' B}]\!]\ d\ e &= \textbf{let}\ \mathit{eval}\ 0\ (v, e) = (v, e) \\
&\qquad\ \mathit{eval}\ i\ (v, e) = \mathit{eval}\ (i - 1)\ ([\![\text{B}]\!]_{\textbf{abs1}}\ d\ v\ e) \\
&\quad\ \textbf{in}\ \mathit{eval}\ ([\![\text{E}]\!]\ e)\ ([\![\text{E'}]\!]\ e,\ e)
\end{aligned}
$$

For expressing folds and scans, **repeataccum** and **repeatnaccum** can be used. These are similar to the previous looping constructs but add an accumulating parameter. A starting value for this must be supplied, which is then passed to the first iteration of the loop body. After that, the result of one iteration is passed as a parameter to the next. The loop body is an abstraction that binds an identifier to the value passed in to the loop. The evaluation rules for the abstraction syntactic classes are given presently.

$$
\begin{aligned}
&[\![\textbf{for}\ \text{(N, N', N")}\ \text{B}]\!]\ d\ e = \\
&\quad \textbf{let}\ \mathit{start} = [\![\text{N}]\!]\ e \\
&\qquad \mathit{stop} = [\![\text{N'}]\!]\ e \\
&\qquad \mathit{inc} = [\![\text{N"}]\!]\ e \\
&\qquad \mathit{eval}\ i\ e = \textbf{if}\ \text{abs}\ (i + \mathit{inc}) > \text{abs}\ (\mathit{stop}) \\
&\qquad\qquad\qquad \textbf{then}\ [\![\text{B}]\!]_{\textbf{abs1}}\ d\ i\ e \\
&\qquad\qquad\qquad \textbf{else}\ \mathit{eval}\ (i + \mathit{inc})\ (\mathit{snd}\ ([\![\text{B}]\!]_{\textbf{abs1}}\ d\ i\ e)) \\
&\quad \textbf{in}\ \mathit{eval}\ \mathit{start}\ e
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{foraccum}\ \text{(N, N', N")}\ \text{E B}]\!]\ d\ e = \\
&\quad \textbf{let}\ \mathit{start} = [\![\text{N}]\!]\ e \\
&\qquad \mathit{stop} = [\![\text{N'}]\!]\ e \\
&\qquad \mathit{inc} = [\![\text{N"}]\!]\ e \\
&\qquad \mathit{eval}\ i\ (v, e) = \textbf{if}\ \text{abs}\ (i + \mathit{inc}) > \text{abs}\ (\mathit{stop}) \\
&\qquad\qquad\qquad\ \ \textbf{then}\ [\![\text{B}]\!]_{\textbf{abs2}}\ d\ i\ v\ e \\
&\qquad\qquad\qquad\ \ \textbf{else}\ \mathit{eval}\ (i + \mathit{inc})\ (\mathit{snd}\ ([\![\text{B}]\!]_{\textbf{abs2}}\ d\ i\ v\ e)) \\
&\quad \textbf{in}\ \mathit{eval}\ \mathit{start}\ ([\![\text{E}]\!]\ e\ ,\ e)
\end{aligned}
$$

Finally, a general for-loop construct is provided, and a variant with an accumulating parameter. The range of loop indexes is described by a tuple of start, stop and increment values. The value of the loop index is passed to each iteration of the loop body through an abstraction. In addition the **foraccum** construct also passes an accumulating parameter through the iterations of the body.

Provided they terminate, all the loop constructs return the result of the final iteration of the loop body. The computational language has additional varieties of loop combinator that build arrays or vectors from the results of each iteration. These are useful for constructing data structures where the value of one element depends on previous elements. However, these features are not really relevant to the details of the coordination languages and management of parallelism, and so are not presented here.

Abstractions

$$\langle \text{ABS1} \rangle ::= \lambda \langle \text{IDENT} \rangle \rightarrow \langle \text{BLOCK} \rangle$$
$$\langle \text{ABS2} \rangle ::= \lambda \langle \text{IDENT} \rangle \langle \text{IDENT} \rangle \rightarrow \langle \text{BLOCK} \rangle$$

$$\begin{aligned}
[\![\lambda\, n \rightarrow \text{B}]\!]_{\text{abs1}}\ d\ v\ e &= \textbf{let}\ e' = \mathit{bind}\ n\ (v,\ \mathit{pushframe}\ e) \\
&\qquad (r, e'') = [\![\text{B}]\!]_{\text{block}}\ d\ e' \\
&\quad \textbf{in}\ (r,\ \mathit{popframe}\ e'') \\
[\![\lambda\, n\ n' \rightarrow \text{B}]\!]_{\text{abs2}}\ d\ v\ v'\ e &= \textbf{let}\ e' = \mathit{bind}\ n\ (v,\ \mathit{pushframe}\ e) \\
&\qquad (r, e'') = [\![\text{B}]\!]_{\text{block}}\ d\ (\mathit{bind}\ n'\ (v',\ e)) \\
&\quad \textbf{in}\ (r,\ \mathit{popframe}\ e'')
\end{aligned}$$

The bodies of the accumulating and **for** loops are abstractions over loop indexes or accumulating values. The syntax and evaluation equations for these abstractions are shown above. These evaluation functions are passed additional values: the actual parameters to be passed to the abstraction. These are bound to the formal parameter identifiers in a new frame. The block itself is then evaluated after which the temporary frame of parameter bindings is removed.

4.1.4 Parallelism Constructs

This section presents the constructs that introduce parallel computation and partition decomposition into the language. These are shared by all the coordination languages of the system. Before presenting the constructs themselves, some supporting types and definitions are given.

Groups

A group is a MPI type that is used to describe a set of processors and an ordering of the set. A group defines a mapping between an ordered sequence of ranks (integers) and the arbitrary processor identifiers (PIDs). Ranks, which are numbered contiguously from one, are more convenient to program with than the irregularly named processor identifiers. Groups are used within the PEDL languages to describe the partitioning of the parallel machine.

$$\begin{aligned}
&\mathit{Group} && \triangleq [\mathit{PID}] \\
&\mathit{mkGroup} && :: \mathit{Integer} \rightarrow \mathit{Group} \\
&\mathit{rankToPid} && :: \mathit{Group} \rightarrow \mathit{Rank} \rightarrow \mathit{PID} \\
&\mathit{pidToRank} && :: \mathit{Group} \rightarrow \mathit{PID} \rightarrow \mathit{Rank}
\end{aligned}$$

$$\begin{aligned}
&\mathit{mkGroup}\ n && = g,\ \mathit{length}\ g = n \\
&\mathit{rankToPid}\ \mathit{grp}\ \mathit{rank} && = \mathit{grp}\ !!\ (\mathit{rank} + 1)
\end{aligned}$$

$$
\begin{array}{lll}
pidToRank\ grp\ pid & = \textbf{let}\ count\ i\ (p : ps) = \textbf{if}\ pid = p & \\
 & \qquad\qquad\qquad\qquad\quad \textbf{then}\ i & \\
 & \qquad\qquad\qquad\qquad\quad \textbf{else}\ count\ (i+1)\ ps & \\
 & \quad \textbf{in}\ count\ 1\ grp &
\end{array}
$$

A group is modeled as a sequence of processor identifiers. The ordering of the PIDs in the sequence determines their ranks in the group. *rankToPid* and *pidToRank* give the mapping between these two naming systems. *mkGroup* is used to generate the group that defines the starting partition of a program, based on the number of required processors passed to the **run** construct. The naming scheme of the PIDs in the starting group is system dependent; all that can be stated about the *mkGroup* function is that it returns a group containing the required number of processors.

Distributed values

As described in the previous chapter, the distributed result of a parallel computation is encapsulated within an abstract datatype called a **DVal**. This ensures a clean separation between the coordination and computation layers of a program. A distributed value has a different element on each processor in the machine, where some of these elements may be undefined. A distributed value is represented as a set of bindings between PIDs and values. The operation *indexDval* projects the element of the distributed value associated with processor named *pid*.

$$
\begin{array}{lll}
Dval & \triangleq & \{(PID, Value)\} \\
indexDval & :: & PID \rightarrow Dval \rightarrow Value \\
indexDval\ pid\ dv & = & dv(pid)
\end{array}
$$

Parallel Machine Configuration

This section describes the machine configuration data that is passed to the evaluation equations – this is the d parameter introduced earlier.

$$
\begin{array}{lll}
CoordinationData & \triangleq & Group \\
GlobalData & \triangleq & (size :: Int,\ group :: Group) \\
ParallelData & \triangleq & (size :: Int,\ rank :: Int,\ pid :: PID,\ group :: Group)
\end{array}
$$

$$
\begin{array}{lll}
mkGlobalData & :: & CoordinationData \rightarrow GlobalData \\
mkParallelData & :: & CoordinationData \rightarrow \{(ParallelData,\ State)\}
\end{array}
$$

$$
\begin{array}{l}
mkGlobalData\ d = (size = length\ d,\ group = d) \\
mkParallelData\ d\ e = \\
\quad \{((size = length\ d,\ rank = i,\ pid = p,\ group = d), pe) \\
\quad\ \mid (i, p, pe) \in zip3\ [1\ ..\ ]\ d\ (splitState\ e)\}
\end{array}
$$

Within the coordination layer of the program, the configuration data is a value of type *CoordinationData* – which is simply a group. This group defines the partition of processors the coordination program is executing on.

When a parallel or global computation block is entered the configuration data is extended with extra information that may be required during its execution. Values of type *GlobalData* or *ParallelData* types are passed to the evaluation equations of such blocks. These types are records that contain the group defining the current partition along with extra fields. The *GlobalData* record type contains a size field – this gives the number of processors in the current partition. In addition the *ParallelData* type has fields that contain the rank and PID of the currently executing processor.

The functions *mkGlobalData* and *mkParallelData* construct the configuration data for a computational block from the data in the coordination layer. *mkGlobalData* returns a single record as the configuration data is identical for all processors participating in the computation. In contrast, the *mkParallelData* function returns a set of record and environment pairs. This is necessary because each processor must have an individual record that contains the correct values for the rank and PID fields. Furthermore, this function splits the world value contained in the coordination layer state, generating a partial world state for each processor.

Syntax

The syntax of the coordination language parallelism constructs is as follows:

⟨PAROP⟩ ::= **parallel** ⟨BLOCK⟩
| **global** ⟨BLOCK⟩
| **globalize** ⟨EXP⟩ ⟨IDENT⟩
| **gspmd** { ⟨PBINDING⟩$^+$ }
⟨PBINDING⟩ ::= ⟨IDENT⟩ ← ⟨PARTITION⟩
⟨PARTITION⟩ ::= ⟨EXP/IDENT⟩ : ⟨BLOCK⟩

The **parallel** construct executes a block of computational code on each processor in the current partition in parallel and returns a **DVal** of their results. The **global** construct does similar but, by constraining the possible operations in the computational body, the result is guaranteed to be replicated across processors. Therefore the result does not need to be encapsulated in the distributed value ADT. The **globalize** construct replicates an element of a distributed value across all processors in the partition. Its parameters are an expression which evaluates to a valid processor rank and the identifier of a distributed value from which to project the element.

The **gspmd** construct divides the current partition into concurrently executing child partitions. It accepts a list of partition bindings: these bind the result of a partition com-

putation to a new identifier. The partition syntactic class comprises a description of the processors in the new partition, and a block of coordination code to execute on these processors. The method used to describe a partition varies across the languages. In the earlier coordination language a single integer expression giving the size is sufficient. Later languages introduce processor identity and so a partition must be described by a group value.

Semantics

The purpose of the semantics presented here is to express the changes to state and the values computed by the coordination language constructs, rather than to model their parallel behaviour. Hence there is no explicit notion of a processor in any of these semantic equations. The parallel behaviour of these constructs is described later in Section 4.2.

$$[\![\textbf{global}\ \mathrm{B}]\!]\ d\ e = [\![\mathrm{B}]\!]_{\textbf{block}}\ (mkGlobalData\ d)\ e$$

$$[\![\textbf{globalize}\ \mathrm{E}\ dv]\!]\ d\ e = \textbf{let}\ p = rankToPid\ d\ ([\![\mathrm{E}]\!]\ e)\ \textbf{in}\ (indexDval\ p\ (findValue\ dv\ e),\ e)$$

The **global** construct generates data that is guaranteed to be replicated over all processors of the current partition. The equation for this construct is quite simple. A machine configuration for the global block is generated from the current group using *mkGlobalData*. The block of computational code is then executed in the current environment and is passed this machine configuration As the current environment is used, any previous bindings are visible within the computational block. However, only other replicated values may be accessed: distributed values are encapsulated within the *Dval* type, for which this language lacks a construct to project elements.

The **globalize** operation evaluates the expression to a rank and maps this to the corresponding PID using the group in the configuration data, which defines the current partition. The element of the distributed value associated with this PID is then projected.

$$\begin{aligned}[\![\textbf{parallel}\ \mathrm{B}]\!]\ d\ e = \textbf{let}\ f\ (pd, pe) &= \textbf{let}\ (v, pe') = [\![\mathrm{B}]\!]_{\textbf{block}}\ pd\ pe \\ &\quad\ \textbf{in}\ ((pd.pid,\ v),\ pe') \\ rs &= \{f\ s \mid s \in mkParallelData\ d\ e\} \\ \textbf{in}\ (\textbf{dom}\ rs,\ &mergeState\ (\textbf{codom}\ rs))\end{aligned}$$

The semantics of the **parallel** construct are a little more involved. A local function f describes the behaviour of a single processor. It accepts a tuple of a system configuration and a state. The computation body is executed in this configuration and state, and the resulting value labelled with the PID of the current processor.

The set rs is the result of applying the single-processor computation f to each element of the set of (*configuration*, *processor state*) pairs generated by *mkParallelData*. The

domain of rs is a set of $(PID, Value)$ pairs – the distributed value which is the result of this construct.

The state returned by the **parallel** construct is the merging of all the resulting processor states. As the temporary frame is removed after evaluating the computational block, the environment at this stage is identical in all processors. Therefore it does not matter which processor store is chosen to become the collective-view environment. All *mergeState* really does is to coalesce the processor world values into a collective-view world value.

$$
\begin{aligned}
[\![\mathrm{E} : \mathrm{B}]\!]_{\mathbf{partition}}\ d\ e &= \mathbf{let}\ d' = mkPartition\ [\![\mathrm{E}]\!]\ d\ e \\
&\quad\ \mathbf{in}\ [\![\mathrm{B}]\!]_{\mathbf{block}}\ d'\ e \\
[\![n \leftarrow \mathrm{P}]\!]_{\mathbf{pbinding}}\ d\ e &= \mathbf{let}\ (v, e') = [\![\mathrm{P}]\!]_{\mathbf{partition}}\ d\ e \\
&\quad\ \mathbf{in}\ ((n,\ v),\ e')
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathbf{gspmd}\ \{\mathrm{PS}\}]\!]\ d\ e\ = \\
&\quad \mathbf{let}\ f\ [\![\mathrm{P}]\!]\ e\ = [\![\mathrm{P}]\!]_{\mathbf{pbinding}}\ d\ e \\
&\qquad\ rs\ = map2\ f\ [\![\mathrm{PS}]\!]\ (splitState\ e) \\
&\qquad\ g\ (n, v)\ e\ = bind\ n\ (v, e) \\
&\quad \mathbf{in}\ (snd\ (last\ (\mathbf{dom}\ rs)),\ fold\ g\ (mergeState\ (\mathbf{codom}\ rs))(\mathbf{dom}\ rs))
\end{aligned}
$$

The evaluation function for a partition generates a new *CooordinationData* value through the use of the *mkPartition* function, and then executes the block in this new machine configuration. This returns a result and state pair. The definition of *mkPartition* is dependent on whether the language uses integers or groups to describe partitions. It is left undefined for now, and a definition given for each language.

The **gspmd** construct is similar to **parallel** but executes a set of coordination language blocks in parallel, rather than a set of computational blocks. The world value is divided between the child partitions; after they have terminated it is merged back together. The local function f describes the execution of one partition. It is mapped along the sequence of partition bindings and fragmented world values to produce rs, which is a set of partition binding results. Evaluating a partition binding executes the partition it contains and pairs the result with the identifier it is to be bound to. Therefore the domain of the set rs is a set of $(Identifier, Value)$ pairs, while the codomain is a set of states.

The value the **gspmd** construct returns is the result of the last partition. The environment returned is the merged state of the states produced by the different partitions, to which each of the $(Identifier, Value)$ pairs returned by the partition bindings is then added.

4.1.5 Computational Language

The block structure of the languages, the set of control constructs shared by all languages, and the parallelism constructs shared by the coordination languages have all

been described. The following sections will now examine the constructs particular to each language.

We start with the computation language. This language has a single processor machine model and is used to express the computational facets of the parallel program. It is the core on which the different parallel computation and global computation languages are based. The language has a rich set of array combinators. However, these can be defined in terms of a few primitives, the semantics of which are presented below.

$$
\begin{array}{ll}
\mathit{Array} & \triangleq \{(\mathit{Index}, \mathit{Value})\} \\
\mathit{indexSet} & :: a \rightarrow \{a\} \\
\mathit{indexArray} & :: \mathit{Index} \rightarrow \mathit{Array} \rightarrow \mathit{Value}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{indexSet}_1\ b & = \{i \mid i \in [1\,..\,b]\} \\
\mathit{indexSet}_2\ (b_i, b_j) & = \{(i,j) \mid i \in [1\,..\,b_i],\ j \in [1\,..\,b_j]\} \\
\mathit{indexArray}\ ix\ a & = a(ix)
\end{array}
$$

An array is modeled as a set of (*Index*, *Value*) pairs. The *Index* type for an n-dimensional array is an n-tuple of integers. The indexes of an array are contiguous and have an origin of 1 for a vector, $(1, 1)$ for a two-dimensional matrix, and so on. The *indexSet* function takes an upper bound and produces a set of all indexes from the origin up to this upper bound. It is difficult to give a clear definition of this functions that works for all dimensionalities of index. Instead two examples are given: definitions for other dimensionalities can be defined as needed by following this pattern.

$$
\begin{array}{lll}
\langle \text{ARRAYOP} \rangle & ::= & \textbf{genarray}\ \langle \text{EXP} \rangle\ \langle \text{ABS1} \rangle \\
& \mid & \textbf{modarray}\ \langle \text{IDENT} \rangle\ \langle \text{ABS2} \rangle \\
& \mid & \langle \text{IDENT} \rangle\ !\ \langle \text{EXP} \rangle \\
& \mid & \textbf{bounds}\ \langle \text{IDENT} \rangle
\end{array}
$$

$$
\begin{array}{lll}
[\![\textbf{genarray}\ \text{E}\ \text{B}]\!]\ d\ e & = & \textbf{let}\ f\ ix\ =\ \textbf{let}\ (v,\ e')\ =\ [\![\text{B}]\!]_{\textbf{abs1}}\ d\ ix\ e \\
& & \qquad\qquad\qquad\qquad\ \textbf{in}\ (ix, v) \\
& & \textbf{in}\ (\{f\ i \mid i \in \mathit{indexSet}\ ([\![\text{E}]\!]\ e)\},\ e) \\
[\![\textbf{modarray}\ a\ \text{B}]\!]\ d\ e & = & \textbf{let}\ \mathit{array}\ \ \ = \mathit{findValue}\ a\ e \\
& & \qquad f\ (ix, v)\ =\ \textbf{let}\ (v',\ e')\ =\ [\![\text{B}]\!]_{\textbf{abs2}}\ d\ ix\ v\ e \\
& & \qquad\qquad\qquad\qquad\ \textbf{in}\ (ix, v') \\
& & \textbf{in}\ \{f\ a \mid a \in \mathit{array}\},\ e) \\
[\![a\ !\ \text{E}]\!]\ d\ e & = & \textbf{let}\ \mathit{array}\ =\ \mathit{findValue}\ a\ e \\
& & \textbf{in}\ (\mathit{indexArray}\ ([\![\text{E}]\!]\ e)\ \mathit{array},\ e) \\
[\![\textbf{bounds}\ a]\!]\ d\ e & = & \textbf{let}\ \mathit{array}\ =\ \mathit{findValue}\ a\ e \\
& & \qquad \mathit{bnd}\ \ \ =\ \textbf{if}\ \mathit{array} \neq \{\}\ \textbf{then}\ \mathit{maximum}\ (\textbf{dom}\ \mathit{array}) \\
& & \qquad\qquad\qquad\qquad\ \textbf{else}\ 0 \\
& & \textbf{in}\ (\mathit{bnd}, e)
\end{array}
$$

A new array is created using **genarray**. This applies the abstraction body to each index in the index set to compute the corresponding element. The **modarray** operation is

similar, but performs a map over an existing array to form the new array. Notice that the order in which the abstractions are executed is not defined. This is necessary as the SAC equivalent of this construct does not define an execution order for the element computations either: this allows the SAC compiler to reorder the computations for the best efficiency. For safety the language does not permit IO to take place within the body of the construct. If IO does take place the result will be undefined as the world value is not threaded though the element computations, but instead is returned unchanged from this construct.

The ! operation indexes an array. This is straightforward, as is computing the **bounds** of an array, which returns the largest index in the array.

4.1.6 Independent Computation Stage

This is the first decision stage in the sequence. As all the common structure has already been presented, the syntax of the coordination language used at this stage can be completed by giving a definition of the missing ⟨ACTION⟩ syntactic class.

⟨ACTION⟩ ::= ⟨PAROP⟩ | ⟨STRUCTURE⟩

So an action in the independent coordination language is either: a parallel operation (for instance a **parallel** or **global**); or a control structure (such as a conditional or loop).

$$mkPartition\ [\![\mathrm{E}]\!]\ d\ e\ =\ mkGroup\ ([\![\mathrm{E}]\!]\ e)$$

Similarly, the semantics of the language can be completed by giving a definition of the missing *mkPartition* function. In this language, a partition is described by a single integer – the size. Hence *mkPartition* can be defined in terms of *mkGroup* given earlier.

Global Language

The syntax of the computational languages used at this stage of the system can be defined in the same way.

⟨ACTION⟩ ::= ⟨ARRAYOP⟩ | ⟨STRUCTURE⟩ | ⟨SIZE⟩

The actions permitted in the global computation language of this stage are array operations, common control structures, and a new construct ⟨SIZE⟩.

Parallel Language

⟨ACTION⟩ ::= ⟨ARRAYOP⟩ | ⟨STRUCTURE⟩
| ⟨RANK⟩ | ⟨SIZE⟩ | ⟨USE⟩ | ⟨GET⟩

Similarly, an action of the parallel language is either an array operation, a common structure, or one of four new constructs.

New Constructs

The following constructs are introduced by the computational languages of this stage of the PEDL system:

⟨RANK⟩ ::= **rank**
⟨SIZE⟩ ::= **size**
⟨USE⟩ ::= **use** ⟨IDENT⟩
⟨GET⟩ ::= **get** ⟨IDENT⟩ ⟨EXP⟩

$$
\begin{aligned}
[\![\textbf{rank}]\!]\ pd\ e &= (pd.rank,\ e) \\
[\![\textbf{size}]\!]\ pd\ e &= (pd.size,\ e) \\
[\![\textbf{use}\ dv]\!]\ pd\ e &= (indexDval\ (pd.pid)\ (findValue\ dv\ e),\ e) \\
[\![\textbf{get}\ dv\ \mathrm{E}]\!]\ pd\ e &= \textbf{let}\ p\ =\ rankToPid\ (pd.group)\ ([\![\mathrm{E}]\!]\ e) \\
&\quad\ \textbf{in}\ (indexDval\ p\ (findValue\ dv\ e),\ e)
\end{aligned}
$$

The **rank** and **size** constructs allow the programmer to query the rank of the current processor and the size of the current partition. Their evaluation functions project the corresponding field from the parallel configuration record, while the state is unchanged.

The other two operations access elements of a distributed value. **use** projects the element local to the current processor by using the *pid* field of the parallel configuration record. **get** accesses an element resident on another processor. The construct takes an expression which evaluates to a rank. This rank is then mapped through the current group to find a PID with which the distributed value is then indexed.

Notice that any constructs that allow access to distributed data, query the processor rank, or perform IO are disallowed in the global computational language. As using these is the only way that a different value can be computed on each processor, the result of a global computation is guaranteed to be replicated.

4.1.7 Distribution Stage

The next decision stage introduces a fixed number of named processors. From this stage groups are used to define partitions of processors.

⟨ACTION⟩ ::= ⟨PAROP⟩ | ⟨STRUCTURE⟩

$mkPartition\ [\![n]\!]\ d\ e\ =\ findValue\ n\ e$

The constructs permitted in this coordination language are unchanged from the previous stage. However, the definition of the *mkPartition* function is changed so that it is passed an identifier bound to a group in the current environment.

Global Language

⟨ACTION⟩ ::= ⟨ARRAYOP⟩ | ⟨STRUCTURE⟩ | ⟨SIZE⟩
| ⟨GROUP⟩

Parallel Language

⟨ACTION⟩ ::= ⟨ARRAYOP⟩ | ⟨STRUCTURE⟩
| ⟨RANK⟩ | ⟨SIZE⟩ | ⟨USE⟩ | ⟨GET⟩
| ⟨GROUP⟩

The ⟨ACTION⟩ syntactic class for the global and parallel computational languages contains the constructs of the previous stage and adds a new set of constructs for manipulating groups.

Group Constructs

These allow the construction and manipulation of group values. In addition to queries such as finding the size of a group, there are constructs to perform set-like operations on groups, compare groups, and create groups by including or excluding a set of ranks given as a vector.

⟨GROUP⟩ ::= **currentgroup**
| **groupSize** ⟨IDENT⟩
| **groupTranslateRanks** ⟨IDENT⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **groupCompare** ⟨IDENT⟩ ⟨IDENT⟩
| **groupIncl** ⟨IDENT⟩ ⟨IDENT⟩
| **groupExcl** ⟨IDENT⟩ ⟨IDENT⟩
| **groupUnion** ⟨IDENT⟩ ⟨IDENT⟩
| **groupIntersection** ⟨IDENT⟩ ⟨IDENT⟩
| **groupDifference** ⟨IDENT⟩ ⟨IDENT⟩

$$[\![\textbf{currentgroup}]\!]\ pd\ e = (pd.group,\ e)$$
$$[\![\textbf{groupSize}\ g]\!]\ pd\ e = (length\ (findValue\ g\ e),\ e)$$

The **currentGroup** operation returns the defining group of the current partition, by projecting the *group* field from the parallel configuration record. The current group is a base value from which other groups can be constructed.

$$\begin{array}{l}[\![\textbf{groupTranslateRanks}\ g\ g'\ a]\!]\ pd\ e = \\ \quad \textbf{let}\ fromGroup = findValue\ g\ e \\ \qquad\ toGroup = findValue\ g'\ e \\ \qquad\ array = findValue\ a\ e \\ \qquad\ array' = \textbf{let}\ pids = \{(ix,\ fromGroup\ !!\ (r - 1)) \mid (ix, r) \in array\} \\ \qquad\qquad \textbf{in}\ \{(ix,\ r) \mid (ix,\ p) \in pids,\ (r,\ p') \in zip\ [1\ ..]\ toGroup,\ p = p'\} \\ \quad \textbf{in}\ (array',\ e)\end{array}$$

The **translateRanks** operation allows a vector of ranks to be mapped through a group, which is useful, for instance, to convert between ranks of a parent and child partition.

$$\begin{array}{lll}[\![\textbf{groupCompare}\ g\ g']\!]\ pd\ e = & \textbf{let}\ group = findValue\ g\ e & \\ & \qquad group' = findValue\ g'\ e & \\ & \qquad n = \textbf{if}\ group = group'\ \textbf{then}\ 1\ \textbf{else} & \\ & \qquad\quad \textbf{if}\ sort\ group = sort\ group'\ \textbf{then}\ 2 & \\ & \qquad\quad \textbf{else}\ 0 & \\ & \textbf{in}\ (n,\ e) & \\ [\![\textbf{groupIncl}\ g\ a]\!]\ pd\ e = & \textbf{let}\ group = findValue\ g\ e & \\ & \qquad array = findValue\ a\ e & \\ & \textbf{in}\ ([group\ !!\ (r - 1) \mid r \in \textbf{codom}\ array],\ e) & \\ [\![\textbf{groupExcl}\ g\ a]\!]\ pd\ e = & \textbf{let}\ group = findValue\ g\ e & \\ & \qquad array = findValue\ a\ e & \\ & \qquad excl = \{group\ !!\ (r - 1) \mid r \in \textbf{codom}\ array\} & \\ & \textbf{in}\ ([p \mid p \leftarrow group,\ \neg(p \in excl)],\ e) & \end{array}$$

The **groupCompare** returns an integer indicating whether two groups are identical, similar (same PIDs, but different ranks), or different. **groupIncl** constructs a new group composed of the ranks contained in an array based on an existing group. The complement of this group can be created using **groupExcl**.

$$\begin{array}{lll}[\![\textbf{groupUnion}\ g\ g']\!]\ pd\ e = & \textbf{let}\ group = findValue\ g\ e & \\ & \qquad group' = findValue\ g'\ e & \\ & \textbf{in}\ (group' +\!\!+ [p \mid p \leftarrow group',\ \neg\ (p \in group)],\ e) & \\ [\![\textbf{groupIntersection}\ g\ g']\!]\ pd\ e = & \textbf{let}\ group = findValue\ g\ e & \\ & \qquad group' = findValue\ g'\ e & \\ & \textbf{in}\ ([p \mid p \leftarrow group,\ p \in group'],\ e) & \\ [\![\textbf{groupDifference}\ g\ g']\!]\ pd\ e = & \textbf{let}\ group = findValue\ g\ e & \\ & \qquad group' = findValue\ g'\ e & \\ & \textbf{in}\ ([p \mid p \leftarrow group,\ \neg(p \in group')],\ e) & \end{array}$$

Finally, the language provides a group generators that allow groups to be treated as sets.

4.1.8 Explicit Communication Stage

The final decision stage in the PEDL system introduces explicit collective communications.

⟨ACTION⟩ ::= ⟨PAROP⟩ | ⟨STRUCTURE⟩
| ⟨COMM⟩ | **communicator** ⟨IDENT⟩

$mkPartition\ [\![n]\!]\ d\ e\ =\ findValue\ n\ e$

The coordination language for this stage is extended with a set of collective communication operations ⟨COMM⟩ and a construct to create a new communicator. A communicator defines a separate namespace for messages, and ensures that messages cannot be intercepted by other communication operations. The *mkPartition* function is unchanged from the previous stage.

Global Language

⟨ACTION⟩ ::= ⟨ARRAYOP⟩ | ⟨STRUCTURE⟩ | ⟨SIZE⟩
| ⟨GROUP⟩

Parallel Language

⟨ACTION⟩ ::= ⟨ARRAYOP⟩ | ⟨STRUCTURE⟩
| ⟨RANK⟩ | ⟨SIZE⟩ | ⟨USE⟩
| ⟨GROUP⟩

While the global language is remains the same as in the previous stage, the **get** operation is removed from the parallel language. Therefore all data redistribution must now take place using the collective communication operations rather than by direct accesses.

Communicator

A communicator is a MPI type that provides a separate namespace for a communication and defines an ordered set of processors that participate in a communication operation. In the semantics a communicator is modeled by a group. The **communicator** operation creates a new communicator from a group.

$$\begin{aligned} &Communicator && \triangleq\ Group \\ &[\![\textbf{communicator}\ n]\!]\ d\ e && =\ (findValue\ n\ e,\ e) \end{aligned}$$

Although communicators and groups share the same representation, we differentiate between the two syntactic classes because of the different properties of these types. The creation of a group in MPI is a asynchronous operation that requires no communication. However, the process of creating a communicator involves synchronization and communication between the group of processors involved. Hence this construct is placed in the coordination layer language. All processors which call the creation procedure must do so with identical parameters. Placing the creation construct in the coordination layer ensures that only replicated data can be passed as parameters.

Collective Communications

The collective communication operations all have the same general form – they accept ranks indicating source and destination processors, a communicator and a distributed value. A new distributed value is returned with the indexes reordered.

⟨COMM⟩ ::= **pointToPoint** ⟨EXP⟩ ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **broadcast** ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **multiBroadcast** ⟨IDENT⟩ ⟨IDENT⟩
| **gather** ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **scatter** ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **totalExchange** ⟨IDENT⟩ ⟨IDENT⟩
| **shift** ⟨EXP⟩ ⟨EXP⟩ ⟨EXP⟩⟨IDENT⟩ ⟨IDENT⟩

The semantic functions have a common structure. The communication operation evaluates or finds the value of its parameters and then uses these to create a new distributed value that reorders the elements of the message data.

$$
\begin{aligned}
&[\![\textbf{pointToPoint}\ \text{E E' } \textit{comm dv}]\!]\ d\ e\ = \\
&\quad \textbf{let } \textit{commGroup} = \textit{findValue comm e} \\
&\qquad \textit{src} = \textit{rankToPid commGroup } ([\![\text{E}]\!]\ e) \\
&\qquad \textit{dest} = \textit{rankToPid commGroup } ([\![\text{E'}]\!]\ e) \\
&\qquad \textit{dval} = \textit{findValue dv e} \\
&\quad \textbf{in } (\{(\textit{dest},\ \textit{indexDval src dval})\},\ e)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\textbf{broadcast}\ \text{E } \textit{comm dv}]\!]\ d\ e\ = \\
&\quad \textbf{let } \textit{commGroup} = \textit{findValue comm e} \\
&\qquad \textit{src} = \textit{rankToPid commGroup } ([\![\text{E}]\!]\ e) \\
&\qquad \textit{dval} = \textit{findValue dv e} \\
&\quad \textbf{in } (\{(p,\ \textit{indexDval src dval}) \mid p \in \textit{commGroup}\},\ e)
\end{aligned}
$$

```
[[multiBroadcast comm dv]] d e =
  let commGroup = findValue comm e
      dval      = findValue dv e
      arr       = {(pidToRank commGroup pid, indexDval pid dval)
                   | pid ∈ commGroup}
  in ({(p, arr) | p ∈ commGroup}, e)

[[gather E comm dv]] d e =
  let commGroup = findValue comm e
      dest      = rankToPid commGroup ([[E]] e)
      dval      = findValue dv e
      arr       = {(pidToRank commGroup pid, indexDval pid dval)
                   | pid ∈ commGroup}
  in ({(dest, arr)}, e)

[[scatter E comm dv]] d e =
  let commGroup = findValue comm e
      src       = rankToPid commGroup ([[E]] e)
      dval      = findValue dv e
      arr       = indexDval src dval
  in ({(p, indexArray (pidToRank commGroup p) arr)
       | p ∈ commGroup}, e)

[[totalExchange comm dv]] d e =
  let commGroup = findValue comm e
      dval      = findValue dv e
      f destRank = {(pidToRank commGroup srcPid
                     , indexArray destRank (indexDval srcPid dval))
                    | srcPid ∈ commGroup}
  in ({(p, f (pidToRank commGroup p)) | p ∈ commGroup}, e)

[[shift E E' E" comm dv]] d e =
  let commGroup = findValue comm e
      dval      = findValue dv e
      periodic  = [[E]] e
      direction = [[E']] e
      disp      = [[E"]] e
      commSize  = length commGroup
      f pid     = let destRank = pidToRank commGroup pid
                      srcRank  = destRank - (direction × disp)
                  in if srcRank < 1 ∨ srcRank > commSize
                     then if periodic
                          then rankToPid commGroup
                               (1 + ((srcRank - 1) mod commSize))
                          else undefined
                     else rankToPid commGroup srcRank
  in ({(p, indexDval (f p) dval) | p ∈ commGroup}, e)
```

4.1.9 Intermediate Language

At this point of the a PEDL derivation all the parallelisation details have been fixed. The next stage in the process is to transform the program to an intermediate language which more closely resembles the target language. Unlike the previous languages, the intermediate language has a conventional single-processor view of the parallel machine. In place of using a coordination layer to express the communication and coordination of the parallel algorithm, the language provides constructs which request these services from an underlying communication system. This is similar to the interface provided by communication libraries to conventional programming languages.

As there is only a single layer to this stage, there is no need for the abstraction of distributed values and the operations associated with them. The language is comprised of the common computational core of structures, array operations and group operations. To this are added single-processor versions of the collective communication constructs. These differ in that they accept elements of data to communicate, rather than entire distributed values. The language has the following syntax:

⟨ACTION⟩ ::= ⟨STRUCTURE⟩ | ⟨ARRAYOP⟩
| ⟨GROUP⟩ | ⟨SIZE⟩ | ⟨RANK⟩
| ⟨COMMRQ⟩
| **communicatorRQ** ⟨IDENT⟩
| **gspmdRQ** { ⟨PBINDING⟩$^+$ }
| **globalizeRQ** ⟨EXP⟩ ⟨IDENT⟩

The language introduces constructs (**communicatorRQ**, **gspmdRQ**, **globalizeRQ** and ⟨COMMRQ⟩) that replace the collective operations of the previous languages. These have the same structure and parameters as their collective counterparts, but produce a result by calling the underlying communication system rather than manipulating data directly. The new syntactic class ⟨COMMRQ⟩ enumerates the communication requests. Each has the same structure as the corresponding collective communication.

⟨COMMRQ⟩ ::= **pointToPointRQ** ⟨EXP⟩ ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **broadcastRQ** ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **multiBroadcastRQ** ⟨IDENT⟩ ⟨IDENT⟩
| **gatherRQ** ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **scatterRQ** ⟨EXP⟩ ⟨IDENT⟩ ⟨IDENT⟩
| **totalExchangeRQ** ⟨IDENT⟩ ⟨IDENT⟩
| **shiftRQ** ⟨EXP⟩ ⟨EXP⟩ ⟨EXP⟩⟨IDENT⟩ ⟨IDENT⟩

Semantics

To represent the semantics of the intermediate stage in the same manner as the previous languages would require a description of the communication system and the interaction between processors. This would introduce concurrency and nondeterminism into the presentation, greatly complicating it.

Instead we use a technique proposed by (O'Donnell, 2000). The semantics of an intermediate language program are defined by a transformation which generates an equivalent collective-view program. The computational code is unchanged in this process, and so can be reasoned about before and after the transformation using the operational semantics defined already.

Transformational Semantics

Figure 4.1 defines a transformation C that generates a semantically equivalent collective-view program from an intermediate language program. The transformation converts communication requests into the corresponding collective communication constructs. These must occur in the coordination layer of the resulting program. Therefore, no matter how deeply nested a communication request occurs, the enclosing block forms part of the boundary between the two layers.

While any communication requests in a block of code are transformed to coordination-level constructs, any surrounding computational code must be placed either in a **parallel** or **global** block. A computation is determined to be parallel whenever it contains constructs specific to parallel blocks (such as **rank**) or references to identifiers bound to the result of previous parallel computations. Otherwise, a global computation block can be used.

The transformation relies on the source program being well-formed – in particular distributed data must not used where replicated data is required. While the collective languages enforce this by the use of different hook constructs and the *Dval* type, at this stage the responsibility lies with the programmer.

We now examine each group of transformation rules in turn:

1. **Blocks.** If the block contains constructs that will appear in the coordination layer, the algorithm must enter the block and independently transform each of the computations within it.

   If the block is comprised solely of computational constructs, then the block can be transformed as a whole without recursing further. If the block contains constructs specific to parallel computations, or refers to distributed values produced

Blocks

$\mathcal{C}[\![\textbf{do B}]\!]$
- | *isCoordination*[[B]] $\Longrightarrow$ **do** $\mathcal{C}[\![\text{B}]\!]$
- | *isParallel*[[B]] $\Longrightarrow$ **parallel** *transUse*[[B]]
- | *otherwise* $\Longrightarrow$ **global B**

Traverse a block

$\mathcal{C}[\![\{\text{B}\}]\!]_{\textbf{block}}$ $\Longrightarrow$ *map* $(\mathcal{C}[\![]\!]_{\textbf{computation}})$ B

$\mathcal{C}[\![\text{A}]\!]_{\textbf{computation}}$ $\Longrightarrow$ $\mathcal{C}[\![\text{A}]\!]_{\textbf{action}}$

$\mathcal{C}[\![n \leftarrow \text{A}]\!]_{\textbf{computation}}$ $\Longrightarrow$ $n \leftarrow \mathcal{C}[\![\text{A}]\!]_{\textbf{action}}$

Computation

$\mathcal{C}[\![\textbf{return E}]\!]$ $\Longrightarrow$ **return** E

$\mathcal{C}[\![\textbf{size}]\!]$ $\Longrightarrow$ **global size**

$\mathcal{C}[\![\textbf{rank}]\!]$ $\Longrightarrow$ **parallel rank**

$\mathcal{C}[\![\text{G}]\!]_{\textbf{group}}$
- | *isParallel*[[G]] $\Longrightarrow$ **parallel** *transUse*[[G]]
- | *otherwise* $\Longrightarrow$ **global G**

$\mathcal{C}[\![\text{A}]\!]_{\textbf{arrayop}}$
- | *isParallel*[[A]] $\Longrightarrow$ **parallel** *transUse*[[A]]
- | *otherwise* $\Longrightarrow$ **global A**

Communication requests

$\mathcal{C}[\![\textbf{gspmdRQ } \{\text{PB}\}]\!]$ $\Longrightarrow$ **gspmd** { *map* $(\mathcal{C}[\![]\!])$ PB }

$\mathcal{C}[\![\textbf{globalizeRQ E I}]\!]$ $\Longrightarrow$ **globalize E I**

$\mathcal{C}[\![\textbf{communicatorRQ I}]\!]$ $\Longrightarrow$ **communicator I**

$\mathcal{C}[\![\textbf{pointToPointRQ E E' I I'}]\!]$ $\Longrightarrow$ **pointToPoint E E' I I'**

⋮

Control Structures

$\mathcal{C}[\![\textbf{if E then B else B'}]\!]$ $\Longrightarrow$ **if E then** $\mathcal{C}[\![\text{B}]\!]$ **else** $\mathcal{C}[\![\text{B'}]\!]$

$\mathcal{C}[\![\textbf{repeat B}]\!]$ $\Longrightarrow$ **repeat** $\mathcal{C}[\![\text{B}]\!]$

⋮

Supporting Functions

- *isCoordination*[[A]]. Boolean. Returns True if A contains any request construct – i.e. **gspmdRQ**, **globalizeRQ**, **communicatorRQ** or a communication request.
- *isParallel*[[A]]. Boolean. Returns True if A contains the **rank** construct or references to any identifier bound to the result of a previous parallel computation or communication.
- *transUse*[[A]]. Transformation. Adds **use** constructs for all distributed values occurring in A. The following is an example of the transformation, where a, b are bound to the results of parallel computations:

  transUse[[*comp a b*]] $\Longrightarrow$ **do** $\{a \leftarrow$ **use** a; $b \leftarrow$ **use** b; *comp a b*$\}$

Figure 4.1: Transformational semantics for the intermediate langauge

by previous parallel computations or communications, then it too must be a parallel computation. Otherwise it can be safely transformed to a global computation.

2. **Traverse a block.** These rules recursively transform all the actions within a block.

3. **Computation.** The next set of rules transform computational code. There is a rule for each syntactic class of construct. For some of the primitives, a clear decision can be made on whether to place it in a parallel or global computation. For constructs that accept parameters, the context in which the parameter value was generated must be analyzed. This requires that a record be maintained of the identifiers currently in scope and the context in which they were generated – this is elided from the presentation.

   As with the rule for blocks, if a parameter to the computation under analysis was created in a parallel context, then this computation must also be placed in a parallel context. As well as adding the **parallel** construct, the transformation calls $transUse[\![]\!]$ to add **use** constructs that access the elements of these distributed values. To remove the need to rename identifiers, the **use** constructs name-shadow the original identifier. An example of this is given at the bottom of the figure.

   On the other hand, if the computation does not access the results of previous parallel computations, it can simply be transformed to a global execution context.

4. **Communication Requests.** A communication request is replaced with the corresponding collective-view operation. The rules for the other communication constructs follow the same pattern and are elided.

5. **Control Structures.** Control structures such as loops and conditionals are left unchanged, but the transformation is recursively applied to the bodies of these constructs. Parameters to a structure (such as the condition of the **if** statement) are unchanged. Provided that the source program is well-formed, such parameters will only contain references to values produced in an appropriate computational context. For conciseness we have omitted the similar rules for the other looping constructs.

An Example

The following intermediate language function creates a communicator which is then used to broadcast an array. The resulting array is then mapped across by a function f which takes the processor rank as an additional parameter.

```
p arr = do
          g ← currentGroup
          comm ← communicatorRQ g
          arr' ← broadcastRQ 1 comm arr
          r ← rank
          modarray arr' (f r)
```

Applying the $\mathcal{C}$ transformation gives the following collective-view program which defines the semantics of the intermediate program:

```
p arr = do
          g ← global currentGroup
          comm ← communicator g
          arr' ← broadcast 1 comm arr
          r ← parallel rank
          parallel arr' ← use arr'
                   r ← use r
                   modarray arr' (f r)
```

This example illustrates the conversion of requests to collective operations and the introduction of parallel and global computational contexts. The final **modarray** computation is determined to be a parallel computation because it refers to two values that have already been transformed into distributed values – the results of the communication and the **rank** construct. Constructs to access the local elements of these distributed values are added to the code.

The transformation produces programs which have a peculiar structure with small-grain parallel blocks and extensive unpacking of distributed values. Although perhaps not the most natural expression of the parallel algorithm, these generated programs give a faithful model of the semantics of the corresponding single-processor view program, which is their intended purpose.

4.1.10 Summary

This section has defined the syntax and semantics for all the languages in the PEDL system. This task was simplified by identifying sets of constructs that are shared by a group of languages and factoring out their definitions. As well as making the language definition more concise, this commonality also simplifies reasoning about the languages: theorems over these constructs can be applied to any language the constructs occur in.

4.2 Parallel Behaviour of the Coordination Languages

While the semantics presented in the preceding section define the values computed by PEDL programs, it says little about the parallel behaviour of these programs. This section gives a concrete specification of the parallel behaviour using *Abstract Parallel Machines*, which were introduced in Section 2.2.6. An APM consists of a set of ParOps which each define the parallel behaviour of an operation and the value it computes.

APMs can be used to structure extensive program derivations. By defining an APM with ParOps for each of the parallelism constructs in the PEDL languages it is hoped that this system can also be used as the 'back-end' for APM derivations intended for our target architecture.

4.2.1 The Definition of a ParOp

An APM consists of n sites of computation $P_1, \ldots, P_n$. The state of the machine is represented by a set of site states $S = \{s_1, \ldots, s_n\}$. The external view of a ParOp is a function that takes a set of site states and r inputs $X = \{x_1, \ldots, x_r\}$ and produces a new set of site states and t outputs $Y = \{y_1, \ldots, y_t\}$.

The definition of a ParOp describes the local computations performed on these sites and how data is communicated between them. It comprises:

- A *site function* f_i for each site P_i, $i = 1, \ldots, n$ with the following type:

  $f_i :: (\sigma_i, (\phi_1, \ldots, \phi_{m_i})) \rightarrow (\sigma_i, (\varphi_1, \ldots, \varphi_{n_i}))$

  A site function takes m_i inputs with types ϕ_j, $j = 1, \ldots, m_i$ and produces n_i result values with types φ_j. In addition, the site has an internal state with type σ_i . Thus f_i takes the old state and inputs and defines the new state and outputs.

- A *coordination function* for each site that describe the internal communications between sites. The function g_i, $i = 1, \ldots, n$ determines where the site function f_i obtains its inputs:

  $g_i :: (\Phi_0, \Phi_1, \ldots, \Phi_n) \rightarrow (\sigma_1, \ldots, \sigma_{s_i})$

  The selection of values required by site i is made from among the system inputs (of type Φ_0) and results produced by other sites; $\Phi_j = (\varphi_1, \ldots, \varphi_{n_i})$ is the type of the value produced by site j, for $j = 1, \ldots, n$.

$$
\begin{array}{lll}
\mathsf{ParOp}\ (s_1,\ldots,s_n)(x_1,\ldots,x_r) & = & ((s'_1,\ldots,s'_n),(y_1,\ldots,y_t)) \\
\quad \textbf{where}\ (s'_i,A_i) & = & f_i(s_i,g_i(V)) \\
\qquad\qquad (y_1,\ldots,y_t) & = & g_0(V) \\
\qquad\qquad V & = & ((x_1,\ldots,x_r),A_1,\ldots,A_n)
\end{array}
$$

Figure 4.2: Standard form of a ParOp

The general form of a ParOp is shown in Figure 4.2. A ParOp with n sites takes the old site states $(s_1,\ldots,s_n)$ and r inputs $(x_1,\ldots,x_r)$ and computes a new state $(s'_1,\ldots,s'_n)$ and the t outputs $(y_1,\ldots,y_n)$ using the local functions f_i and g_i. An additional coordination function g_0 selects the outputs from the values produced by the set of sites.

V consists of all available values – the A_i produced by every site i and the external inputs x_i. The function f_i in site i takes the old site state s_i and a selection of the available values, chosen by g_i, and it returns a new local site state s'_i and some number (possibly zero) of site results $A_i = (a_{i,1},\ldots,a_{i,n_i})$

There appears to be a circularity in the general form, as V is defined using the site outputs A_i, which are in turn defined using V. This is not really a problem – V is a container of all available values. Deadlock will only occur when the f_i and g_i functions cause a particular value (a site state s_i or site output $a_{i,j}$) to be defined in terms of itself.

4.2.2 An APM for PEDL

The parallel behaviour of the PEDL languages is described by the definition of an APM with ParOps for each of the coordination level constructs. The site states of this APM will be the environments of bindings maintained by each processor. We will not represent this explicitly – it is similar to the store used in the operational semantics of the previous section. Instead bindings are added and retrieved using two primitives:

$$
\begin{array}{lll}
\mathit{bind} & :: & (\mathit{Identifier},\mathit{Value}) \rightarrow \mathit{SiteState} \rightarrow \mathit{SiteState} \\
\mathit{lookup} & :: & \mathit{Identifier} \rightarrow \mathit{SiteState} \rightarrow \mathit{Value}
\end{array}
$$

All the language constructs operate solely over the site states – there is no other input or output from the ParOps. Therefore, we can use an abridged form where the input and output values have been removed as follows:

$$
\begin{array}{lll}
\mathsf{ParOp}\ (s_1,\ldots,s_n) & = & (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_i,A_i) & = & f_i(s_i,g_i(V)) \\
\qquad\qquad V & = & (A_1,\ldots,A_n)
\end{array}
$$

To start, we define ParOps for the communication operations of the language. The globalize construct performs a broadcast communication from the source site to all other sites in the machine. The definition is straightforward – the *src* site finds the binding

in the local site environment and makes it available as a result. The other sites return no result (denoted by ()). The other sites retrieve the result of the *src* site and store it in their own site local environment.

$$
\begin{array}{lll}
[\![r \leftarrow \textbf{globalize}\ src\ dv]\!]\,(s_1,\ldots,s_n) & = & (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_i, A_i) & = & f_i(s_i, g_i(V)) \\
\qquad V & = & (A_1,\ldots,A_n) \\
\qquad f_{src}(s,\_) & = & \textbf{let}\ v = lookup\ d\ s \\
 & & \quad \textbf{in}\ (bind\ (r,v)\ s, v) \\
\qquad f_{others}(s,v) & = & (bind\ (r,v)\ s, ()) \\
\qquad g_{src}(\_) & = & () \\
\qquad g_{others}(\ldots, A_{src}, \ldots) & = & A_{src}
\end{array}
$$

The **pointToPoint** communication is similar. The value to be communicated is retrieved from the store of the source site and made available as an output. The destination site takes this value and adds a binding in its local store. All other site states are unchanged. Therefore this identifier r is bound to a value only in the store of the destination site; on other sites the value of this identifier is undefined.

Communications use a *communicator* to provide a context and also a ranking scheme for the sites involved. A communicator can be represented as an array of site indexes. In the following ParOp definitions the site functions are subscripted by a site index given as an element of a communicator array.

$$
\begin{array}{lll}
[\![r \leftarrow \textbf{pointToPoint}\ src\ dest\ comm\ d]\!]\,(s_1,\ldots,s_n) & = & (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_i, A_i) & = & f_i(s_i, g_i(V)) \\
\qquad V & = & (A_1,\ldots,A_n) \\
\qquad f_{comm[src]}(s,\_) & = & (s, lookup\ d\ s) \\
\qquad f_{comm[dest]}(s,v) & = & (bind\ (r,v)\ s,\ ()) \\
\qquad f_{others}(s,\_) & = & (s, ()) \\
\qquad g_{comm[dest]}(\ldots, A_{comm[src]}, \ldots) & = & A_{comm[src]} \\
\qquad g_{others}(\_) & = & ()
\end{array}
$$

In the **scatter** operation, the source site provides an array of messages, where the i^{th} element of the array is to be communicated to the site index defined by $comm[i]$. There may be processors in the machine which are not part of the present group. Their site state is unchanged: this is captured by the *others* case.

$$
\begin{array}{lll}
[\![r \leftarrow \textbf{scatter}\ src\ comm\ d]\!]\,(s_1,\ldots,s_n) & = & (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_i, A_i) & = & f_i(s_i, g_i(V)) \\
\qquad V & = & (A_1,\ldots,A_n) \\
\qquad f_{comm[src]}(s,\_) & = & \textbf{let}\ v = lookup\ d\ s \\
 & & \quad \textbf{in}\ (bind\ (r, v[src])\ s, v) \\
\qquad f_{comm[j]}(s,v) & = & (bind\ (r, v[j])\ s, ()) \\
\qquad f_{others}(s,\_) & = & (s, ()) \\
\qquad g_{comm[j]}(\ldots, A_{comm[src]}, \ldots) & = & A_{comm[src]}[v_j] \\
\qquad g_{others}(\_) & = & () \\
 & & 1 \le j \le len(v),\ j \neq src
\end{array}
$$

The other collective communications follow a similar pattern, and are omitted.

Next we present the operations which coordinate computational code – the hook constructs. These are parameterized over a block of computational code. We use $\mathcal{E}[\![]\!]$ to denote the evaluation of these computational blocks. $\mathcal{E}$ is left unspecified – the operational semantics already defines how computational code is evaluated – but the ParOp definition does definedoes describe which site state and parallel configuration data is passed to it.

The **global** construct executes code in a restricted environment that guarantees the same value will be computed on all sites. No communication takes place between sites. The only information available about the machine configuration is the current group – an array of the site indexes.

$$
\begin{array}{lcl}
[\![r \leftarrow \textbf{global B}]\!]\,(s_1,\ldots,s_n) & = & (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_i, A_i) & = & f_i(s_i, g_i(V)) \\
\qquad V & = & (A_1,\ldots,A_n) \\
\qquad f_i(s,\_) & = & \textbf{let}\ (\_, v) = \mathcal{E}[\![\text{B}]\!]\ s\ grp \\
\qquad\qquad & & \quad grp = \{1,\ldots,n\} \\
\qquad\qquad & & \textbf{in}\ (bind\ (r, v)\ s, ()) \\
\qquad g_i(\_) & = & ()
\end{array}
$$

The **parallel** construct takes two forms. The first form presented is for the later stages of the PEDL system where all communication is by explicit collective construct. Therefore parallel computational code may only access bindings in the environment of the local site. The ParOp definition is similar to that for **global**, but the index of the current site is also made available to the computational code. This is used to provide a result for the **rank** primitive.

$$
\begin{array}{lcl}
[\![r \leftarrow \textbf{parallel B}]\!]\,(s_1,\ldots,s_n) & = & (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_i, A_i) & = & f_i(s_i, g_i(V)) \\
\qquad V & = & (A_1,\ldots,A_n) \\
\qquad f_i(s,\_) & = & \textbf{let}\ (\_, v) = \mathcal{E}[\![\text{B}]\!]\ s\ grp\ rank \\
\qquad\qquad & & \quad grp = \{1,\ldots,n\} \\
\qquad\qquad & & \quad rank = i \\
\qquad\qquad & & \textbf{in}\ (bind\ (r, v)\ s, ()) \\
\qquad g_i(\_) & = & ()
\end{array}
$$

In the earlier stages of the derivation, communication is unstructured and non-local accesses can be made within computational blocks using the **get** construct. The pattern of non-local data access is not apparent until the computational block B is evaluated. Therefore it is not possible to give a precise definition for the communication functions g_i for this ParOp. Instead, all site states are made available to the evaluation function $\mathcal{E}$ to be accessed as needed.

$$
\begin{array}{l}
[\![r \leftarrow \textbf{parallel B}]\!]\,(s_1,\ldots,s_n) = (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_i, A_i) = f_i(s_i, g_i(V)) \\
\qquad\qquad V = (A_1,\ldots,A_n) \\
\qquad\qquad f_i(s,S) = \textbf{let}\ (\_,v) = \mathcal{E}[\![\text{B}]\!]\ s\ grp\ rank\ S \\
\qquad\qquad\qquad\qquad\quad grp = \{1,\ldots,n\} \\
\qquad\qquad\qquad\qquad\quad rank = i \\
\qquad\qquad\qquad\qquad \textbf{in}\ (bind\ (r,v)\ s,\ ()) \\
\qquad\qquad g_i(V) = V
\end{array}
$$

We can describe how this data is used by the computation evaluation function by giving the evaluation rule for the get construct. The index is used to access the site state from that particular site, in which the requested identifier is looked up.

$$\mathcal{E}[\![get\ ix\ dv]\!]\ s\ grp\ rank\ (\ldots,s_{ix},\ldots) = (s,\ lookup\ dv\ s_{ix})$$

The final parallel construct of the languages is **gspmd**. This partitions the sites into independent machines that each evaluate a coordination block independently. This construct does not fit into the conventional ParOp form because it performs no computation itself. However, we can describe the way in which the sites are partitioned and then joined together again.

$$
\begin{array}{l}
[\![\textbf{gspmd}\{r_1 \leftarrow grp_1 : \text{B}_1;\ldots;r_p \leftarrow grp_p : \text{B}_p\}]\!]\,(s_1,\ldots,s_n) = (s'_1,\ldots,s'_n) \\
\quad \textbf{where}\ (s'_1,\ldots,s'_n) = \bigcup_{1\le j\le p} S'_j \\
\qquad\qquad S'_j = \textbf{let}\ S_j = (s_{grp_j[1]},\ldots,s_{grp_j[len(grp_j)]}) \\
\qquad\qquad\qquad\quad \textbf{in}\ \mathcal{E}[\![r_j \leftarrow \text{B}_j]\!]\ S_j
\end{array}
$$

For each partition $j = 1,\ldots,p$, a group grp_j is used to define the subset of sites S_j in the partition. A partition computation B_j is then evaluated independently on each subset of sites. The resulting site subsets for each partition are unioned together to produce the result set of sites.

4.2.3 Summary

We have used Abstract Parallel Machines to describe the parallel behaviour of language constructs. As well as providing a clear presentation of the operational details of the constructs this links the PEDL system into the APM hierarchy: APM derivations may use PEDL as an implementation route after the high-level exploratory algorithm design has been carried out within the APM system.

APMs also provide a structure in which to express cost models (O'Donnell et al., 2000), which could then be used to compare different implementation choices. A cost model can be defined for an individual machine in the APM hierarchy and from this cost measures generated for child- or parent-node APMs. It would be possible to produce a cost model for the PEDL system using the APM we have defined in this section. The cost model

could be either be derived from the internal descriptions of the ParOps or based on real timings of the underlying MPI operations.

APMs are usually used for algorithm derivation. As far as we know, this is the first use of APMs to describe language semantics. It was found that the standard form of ParOp in combination with semantic evaluation functions was sufficient to express most of the language constructs. An exception was the partitioning **gspmd** construct which does not fit into this form. However, it was found to be possible to describe the effect of this construct in a similar style to the rest of the ParOp definitions.

4.3 Reasoning about the Languages

The operational semantics of the languages can be used to prove equivalences between program fragments expressed in the same language (for horizontal transformations) and between fragments expressed in adjacent languages (for vertical transformations). The method used is to show that the corresponding semantic equations cause the same changes to the machine state and return the same value.

4.3.1 Validity

It is normal to state an equivalence between expressions that holds for all possible values of a particular variable. This is done by leaving free variables in the expressions which are then universally quantified by an explicit 'forall' notation. For instance, the theorem that composition distributes over map is commonly written:

$$\forall\, xs :: [a] \cdot map\,(f\,.\,g)\,xs \equiv (map\,f\,.\,map\,g)\,xs$$

Here the 'forall' symbol indicates that this theorem is true for all values of the type $[a]$.

However, some of the types used within the PEDL language semantics have values that are invalid or cannot occur in the model. For example, a group is represented as a sequence of processor identifiers. Each element of the sequence must be unique – as a processor may only belong to a group once. Therefore sequences containing duplicates are invalid models of a group.

When expressing a theorem involving free variables of these types, we wish to state that the theorem holds for all *valid* values of the type. This subset of the type is specified by defining a predicate *Valid*: like 'forall' this is then used as a quantifier for free variables. We now define the validity predicate for all types used in the semantics which have values that are not valid representations. First the definitions for types used to model data structures:

$$
\begin{array}{lll}
Valid(a :: Array) & \Longrightarrow & size\ (\mathbf{dom}\ a) = size\ a \\
 & & minimum\ (\mathbf{dom}\ a) = 1,\ a \neq \{\} \\
 & & maximum\ (\mathbf{dom}\ a) = size\ a,\ a \neq \{\} \\
Valid(dv :: Dval) & \Longrightarrow & size\ (\mathbf{dom}\ dv) = size\ dv \\
Valid(c :: Communicator) & \triangleq & Valid(c :: Group) \\
Valid(g :: Group) & \Longrightarrow & \forall\, i, j \in \{0 \ldots length\ g - 1\}\,. \\
 & & g!i = g!j \Longrightarrow i = j
\end{array}
$$

The first definition gives the properties of a well-formed array. For an array to be valid, the three conditions listed must be true. This datatype is represented as a set of $(Index, Value)$ pairs. An array is valid if each index is unique: that is, the set of indices (the domain) is the same size as the set of bindings itself. Furthermore, for a non-empty array the smallest index must be the unit of the index type and the set of indexes must be contiguous.[1]

A distributed value is a set of $(PID, Value)$ pairs. The only validity condition is that each processor identifier is unique. A communicator is represented as a group. Therefore, the validity condition for a communicator is defined to be the validity condition for a group. A group is a sequence of processor identifiers: each must be unique. This is expressed in the standard way by stating that, for any two indexes, if the elements the indexes project from the sequence are equal then the indexes themselves must be equal. Next we examine the validity conditions for the data structures used to models the parallel configuration of the machine.

$$
\begin{array}{lll}
Valid(d :: CoordinationData) & \triangleq & Valid(d :: Group) \\
Valid(gd :: GlobalData) & \Longrightarrow & Valid(gd.group :: Group) \\
 & & gd.size = length\ (gd.group) \\
Valid(pd :: ParallelData) & \Longrightarrow & Valid(pd.group :: Group) \\
 & & pd.size = length\ (pd.group) \\
 & & pd.rank = pidToRank\ (pd.group)\ (pd.pid) \\
 & & pd.pid = rankToPid\ (pd.group)\ (pd.rank)
\end{array}
$$

from which can be inferred

$$
\begin{array}{l}
pd.pid \in pd.group \\
pd.rank \in \{1 \ldots length\ (pd.group)\}
\end{array}
$$

The machine configuration in the coordination languages is represented as a group. Therefore, the configuration is valid if the group is valid. For the machine configuration types used in global and parallel computational blocks there are more restrictions: certain relationships must hold between the different elements of the record. A *GlobalData* record value is valid if the *group* field is a valid group and the *size* field corresponds to the size of the group. A valid *ParallelData* record has the constraints of the global configuration

[1] As it is difficult to give equations that express these constraints for all index types, the equations presented only apply to arrays indexed by single integers.

record, and also a relationship between the value of the *pid* and *rank* fields. Mapping from the rank field through the group must result in the value of the *pid* field, and vice versa. From the constraints on *ParallelData* records, some further lemmas can be deduced. For instance, the *pid* field must be an element in the group, while the *rank* field will be a value between one and the length of the group.

The only way that machine configuration records are created in the operational semantics is by the use of the *mkGlobalData* and *mkParallelData* functions. It can be shown that these functions always generate valid records.

4.3.2 An Example Proof

This section presents an example of a proof of equivalence between program fragments. The proposition we wish to prove is that for any distributed value *dv*, calling *use dv* to access the local element is equivalent to calling **get** *dv r*, where *r* is the rank of the current processor. That is:

$$\forall\, dv \cdot \mathbf{use}\ dv \Leftrightarrow \begin{array}{l}\mathbf{do}\\ \quad r \leftarrow \mathbf{rank}\\ \quad \mathbf{get}\ dv\ r\end{array}$$

We prove the proposition by showing that the corresponding semantic functions are equivalent.

$$\forall\, dv\ e,\ \mathit{Valid}(d :: \mathit{ParallelData}) \cdot \\ [\![\mathbf{use}\ dv]\!]\ d\ e \equiv [\![\mathbf{do}\ \{r \leftarrow \mathbf{rank};\ \mathbf{get}\ dv\ r\}]\!]\ d\ e \qquad (1)$$

This equation introduces new free variables: the environment and the machine configuration. As any value of the environment type is valid, the environment can be universally quantified. However, the *d* parameter is restricted to the set of valid parallel machine configurations.

Figure 4.3 shows the left hand side of the proof of this equation. Using the semantic equation for **use** the expression can be rewritten as an operation on the environment and the parallel configuration. As the value of these parameters are not defined, the reduction must stop here. The right hand side of the equation is a block of code, rather than a single construct. Due to this, the right hand side of the proof (Figure 4.4) has a few steps that merely unfold the syntactic block. The details of the proof, line by line, are as follows:

$$\begin{array}{ll} \mathit{LHS} & = [\![\mathbf{use}\ dv]\!]\ d\ e\\ & = \langle 1\text{: use}\rangle\\ & \quad (\mathit{indexDval}\ (d.\mathit{pid})\ (\mathit{findValue}\ dv\ e),\ e)\end{array}$$

Figure 4.3: LHS of proof of Equation 1

$$
\begin{aligned}
RHS &= [\![\textbf{do}\ \{r \leftarrow \textbf{rank};\ \textbf{get}\ dv\ r\}]\!]\ d\ e \\
&= \langle 1\text{: do}\rangle \\
&\quad [\![\{r \leftarrow \textbf{rank};\ \textbf{get}\ dv\ r\}]\!]_{\textbf{block}}\ d\ e \\
&= \langle 2\text{: block}\rangle \\
&\quad \textbf{let}\ (v, e') = [\![r \leftarrow \textbf{rank};\ \textbf{get}\ dv\ r]\!]_{\textbf{computations}}\ d\ (\mathit{pushframe}\ e) \\
&\quad \textbf{in}\ (v,\ \mathit{popframe}\ e') \\
&= \langle 3\text{: computations}\rangle \\
&\quad \textbf{let}\ (v, e') = \textbf{let}\ (v, e') = [\![r \leftarrow \textbf{rank}]\!]\ d\ (\mathit{pushframe}\ e) \\
&\qquad\qquad \textbf{in}\ [\![\textbf{get}\ dv\ r]\!]\ d\ e' \\
&\quad \textbf{in}\ (v, \mathit{popframe}\ e') \\
&= \langle 4\text{: computation}\rangle \\
&\quad \textbf{let}\ (v, e') = \textbf{let}\ (v, e') = \textbf{let}\ (v, e') = [\![\textbf{rank}]\!]\ d\ (\mathit{pushframe}\ e) \\
&\qquad\qquad\qquad\qquad \textbf{in}\ (v,\ \mathit{bind}\ r\ (v, e') \\
&\qquad\qquad \textbf{in}\ [\![\textbf{get}\ dv\ r]\!]\ d\ e' \\
&\quad \textbf{in}\ (v, \mathit{popframe}\ e') \\
&= \langle 5\text{: rank}\rangle \\
&\quad \textbf{let}\ (v, e') = \textbf{let}\ (v, e') = (d.\mathit{rank},\ \mathit{bind}\ r\ (d.\mathit{rank},\ \mathit{pushframe}\ e)) \\
&\qquad\qquad \textbf{in}\ [\![\textbf{get}\ dv\ r]\!]\ d\ e' \\
&\quad \textbf{in}\ (v, \mathit{popframe}\ e') \\
&= \langle 6\text{: factor out pushframe and bind}\rangle \\
&\quad \textbf{let}\ be = \mathit{bind}\ r\ (d.\mathit{rank}, \mathit{pushframe}\ e) \\
&\quad \textbf{in let}\ (v, e') = [\![\textbf{get}\ dv\ r]\!]\ d\ be \\
&\qquad \textbf{in}\ (v,\ \mathit{popframe}\ e') \\
&= \langle 7\text{: get}\rangle \\
&\quad \textbf{let}\ be = \mathit{bind}\ r\ (d.\mathit{rank}, \mathit{pushframe}\ e) \\
&\quad \textbf{in let}\ (v, e') = \textbf{let}\ p = \mathit{rankToPid}\ (d.\mathit{group})\ ([\![r]\!]\ be) \\
&\qquad\qquad\qquad \textbf{in}\ (\mathit{indexDval}\ p\ (\mathit{findValue}\ dv\ be),\ be) \\
&\qquad \textbf{in}\ (v,\ \mathit{popframe}\ e') \\
&= \langle 8\text{: expression evaluation}\rangle \\
&\quad \textbf{let}\ be = \mathit{bind}\ r\ (d.\mathit{rank}, \mathit{pushframe}\ e) \\
&\quad \textbf{in let}\ (v, e') = \textbf{let}\ p = \mathit{rankToPid}\ (d.\mathit{group})\ (\mathit{findValue}\ r\ be) \\
&\qquad\qquad\qquad \textbf{in}\ (\mathit{indexDval}\ p\ (\mathit{findValue}\ dv\ be),\ be) \\
&\qquad \textbf{in}\ (v,\ \mathit{popframe}\ e') \\
&= \langle 9\text{: findValue}\rangle \\
&\quad \textbf{let}\ be = \mathit{bind}\ r\ (d.\mathit{rank}, \mathit{pushframe}\ e) \\
&\quad \textbf{in let}\ (v, e') = \textbf{let}\ p = \mathit{rankToPid}\ (d.\mathit{group})\ (d.\mathit{rank}) \\
&\qquad\qquad\qquad \textbf{in}\ (\mathit{indexDval}\ p\ (\mathit{findValue}\ dv\ be),\ be) \\
&\qquad \textbf{in}\ (v,\ \mathit{popframe}\ e') \\
&= \langle 10\text{: Valid law for parallel data, subst for p}\rangle \\
&\quad \textbf{let}\ be = \mathit{bind}\ r\ (d.\mathit{rank}, \mathit{pushframe}\ e) \\
&\quad \textbf{in let}\ (v, e') = (\mathit{indexDval}\ (d.\mathit{pid})\ (\mathit{findValue}\ dv\ be),\ be) \\
&\qquad \textbf{in}\ (v,\ \mathit{popframe}\ e') \\
&= \langle 11\text{: subst for v and e'}\rangle \\
&\quad \textbf{let}\ be = \mathit{bind}\ r\ (d.\mathit{rank}, \mathit{pushframe}\ e) \\
&\quad \textbf{in}\ (\mathit{indexDval}\ (d.\mathit{pid})\ (\mathit{findValue}\ dv\ be),\ \mathit{popframe}\ be) \\
&= \langle 12\text{: popframe, bind, pushframe}\rangle \\
&\quad \textbf{let}\ be = \mathit{bind}\ r\ (d.\mathit{rank}, \mathit{pushframe}\ e) \\
&\quad \textbf{in}\ (\mathit{indexDval}\ (d.\mathit{pid})\ (\mathit{findValue}\ dv\ be),\ e) \\
&= \langle 13\text{: findValue when no binding in first frame}\rangle \\
&\quad (\mathit{indexDval}\ (d.\mathit{pid})\ (\mathit{findValue}\ dv\ e),\ e)
\end{aligned}
$$

Figure 4.4: RHS of proof of Equation 1

1–4. The block of constructs is unfolded using the semantic equation for do, then block, rhen computations and computation. this decompses the block into its constituent actions.

5. Rewriting using the semantic equations for **rank** results in a frame with a binding of r to the rank field of the parallel configuration being added to the environment.

6. The description of the new state is a little unwieldy. Rather than reduce it further using the definitions of *pushframe* and *bind*, it is factored out into a new let-binding and left in this non-normalized form.

7–8. The semantic equation for **get** is used to rewrite the return value of the computational block: this introduces a further let expression. The expression $[\![r]\!]$ is then evaluated. Evaluation rules for expressions are not defined, being left to the semantics of the host language. However, in this case it is clear that the expression r is evaluated by finding the corresponding binding in the environment.

9. The *findValue* function returns the binding of r to $d.rank$ that was recently added to the environment.

10. As d must be a valid configuration record, the following identity holds:

$$d.pid \equiv rankToPid\ (d.group)\ (d.rank)$$

This can be used to reduce the definition of p to $d.pid$. The associated let expression is then removed by substituting for p in the main expression.

11. The inner let binding is a tuple bound to a tuple pattern. Therefore, v and e' can be substituted by their definitions in the main expression.

12. By inspecting the definition of *bind*, *pushframe* and *popframe*, it can be seen that the following identity holds.

$$popframe\ .\ n\ .\ pushframe \equiv id$$
where
$$n = \langle \text{any number of applications of } bind \rangle$$

Using this identity, *popframe be* can be replaced by e. This shows that the computational block has made no lasting changes to the machine environment.

13. Finally, we can remove the definition of *be* and replace it by e in the main expression by observing that the following holds

$$findValue\ n\ (bind\ n'\ (v, pushframe\ e)) \equiv findValue\ n\ e$$
where
$$n \neq n'$$

The two sides of the semantic equation have been proven equal: therefore the two constructs have identical effects upon the machine, and are equivalent. It is noticeable that many little lemmas were introduced to simplify the stages of the main proof. If the PEDL system was to be used for real problems, libraries of these supporting lemmas and theorems would have to be provided to make proofs and reasoning practical.

4.4 Correctness

While the design of the PEDL languages prevents many errors occurring, for instance by differentiating between replicated and distributed data, it is still possible to write erroneous programs that will fail at runtime.

An arbitrary group value can be used to define a communicator. If the group refers to processors that are not in the current partition then the result of any communications performed with this communicator will be undefined. Similarly, indexing outside the bounds of an array will cause a runtime error. There are further circumstances where the languages do not prevent incorrect programs from being constructed. In fact, it may not be possible to design a language that prevents all these cases while still being flexible enough to conveniently express algorithms.

Therefore, as a program passes through the stages of the PEDL system, informal transformations may be performed that do not preserve the correctness of the program. Although the languages do not prohibit these erroneous programs, we claim that any transformation that generates such a program cannot be shown to be correct with respect to the operational semantics. It follows that if a transformation can be proven to be correctness-preserving, it will not generate invalid programs.

This claim assumes that the operational semantics we present are complete and consistent. No attempt has been made to prove this - to do so would require an induction over the set of all possible programs, which would be a lengthy undertaking. However we are reasonably confident that the semantics we present are correct, or at worse could be rectified with minor changes.

4.5 Summary

This chapter has given a detailed description of the syntax, semantics and parallel behaviour of the languages of the PEDL system. The semantics of the languages of the decision stages are described in an operational fashion; this presentation was developed and tested by modeling it within Haskell. The operational semantics is made more concise by the ability to factor out common classes of constructs and describe them once for all the languages. This commonality could also simplify reasoning about the languages.

An example equivalence proof was performed to demonstrate that the operational semantics could be used to reason about program fragments. Although this is possible, larger proofs will require libraries of supporting theorems.

The intermediate language has a different execution model than the previous languages in the system. A communication system which is outside the language is used to exchange

data between processors. The external nature of this system makes it difficult to give an operational semantics for this language that fits within the framework used by the other languages. Instead a transformational semantics is given which generates an equivalent collective-view program.

The parallel behaviour of the languages was described separately by defining an APM for the PEDL system, and a ParOp for each of the parallel constructs. Although this is an unusual application of APMs, it was found that the parallel behaviours could be expressed without much difficulty.

Chapter 5

Implementing a Series of Domain Specific Embedded Languages

Capsule

This chapter presents the implementation of the PEDL languages as an embedded set of combinators within Haskell, a lazy functional language.

The chapter first examines the development of the implementation of *Domain Specific Embedded Languages* (DSELs) in functional languages and explains the benefits of this method of implementation. The most significant of these is that the language may inherit many of its conventional features from the host: only the novel features need to be designed and implemented afresh. Furthermore, any existing tools supporting the host language can be used with the embedded language. This makes embedding a quick and lightweight implementation method and simplifies experimentation with different language designs.

After sketching the overall structure of the implementation, some of its more interesting features are presented. The PEDL languages have two unusual properties. The occurrence of hard-to-implement features of the host language must be controlled to simplify the final translation to the target language. In addition, a PEDL program is comprises two layers, each of which is expressed in a different language. *Phantom Types* are used to restrict difficult host language features and to ensure that program layers are composed correctly. These represent the constraints and structure of the PEDL languages within the type system of the host language, which allows errors to be detected during compilation. Furthermore, it was found to be possible and useful to extend this method to allow code from adjacent decision stages to be composed.

Introduction

There are a variety of techniques that can be used to implement a language. These include writing an inefficient but clear interpreter, an efficient conventional compiler, or a pre-processor that translates the source into another high-level language. For the languages of PEDL we chose to use the technique of embedding within a host language.

The first section traces the development of embedded languages and examines some of the techniques commonly used to model the semantics and type system of the embedded language in the host. In passing, this section also introduces some of the more advanced features of Haskell, a language commonly used for such applications.

The following sections present some of the interesting points of applying this implementation technique to our series of languages. On the way we describe two new applications of *Phantom Types*: representing the features of a language, and detecting the occurrence of constructs within a program block.

5.1 Embedded Languages

Implementation by embedding relies on the following observation: many of the decisions in the design of a programming language are not all that important. The design details of a language's arithmetic, scoping rules, pattern matching, type system and module system tend not to matter that much. There are often many perfectly reasonable solutions that could be chosen.

An embedded language is produced by implementing the significant constructs of the language as a library of functions, procedures and types in a host language. In this way, the implementor can concentrate on the design of the novel features, while the less important details are inherited from the host language.

Source programs are then written in a subset of the host language, making use of the embedded language's library of program constructs and types. The infrastructure of the host language – development environment, compilers, debuggers, profilers and runtime system – can all be utilized by the embedded language implementation. Similarly, there is no need to implement a parser or type-checker for the embedded language – the host language implementation provides these too.

At first glance, this technique appears to provide little more than an API to a code library. However, the choice of host language makes a substantial difference to this perception. By choosing a versatile, flexible language the 'feel' of coding in the embedded language is much closer to writing real programs than calling subroutines. Scheme was the language of choice for early embedded implementations (Abelson and Sussman, 1984),

while recently strongly-typed functional languages, such as Haskell (Peyton Jones, 1999) or ML (Milner, 1983), have gained in popularity.

5.1.1 Combinator Libraries

Within Haskell, the forerunners of embedded languages are libraries which tackled complex problems, such as parsing (Hutton, 1990; Hutton, 1992), pretty printing (Hughes, 1995), document manipulation (Wallace and Ranciman, 1999; Gill, 1999) and graphical user interfaces (Carlsson and Hallgren, 1993).

In place of an API of simple functions, the interface to these libraries is a collection of operators and higher-order functions which manipulate an abstract type. Primitive operations are also provided (such as displaying a string, parsing a single character, or displaying a single button.)

Applications are built by using the operators and higher-order functions to combine the primitive operations into larger expressions. These expressions are then passed to an evaluator or 'run' function to produce an observable result. Writing code using such a library resembles programming in a small self-contained language.

Applications which use combinator libraries are claimed to be quick to build; more likely to be correct; and simple to understand and modify. This is because no 'new' code is written: existing abstractions are simply composed together. There are also domain-specific benefits: for instance a parser implemented with combinators resembles the syntax it was designed to accept.

Analysis

Combinator libraries are versatile, safe and expressive. They mix easily with other program components, while the type system regulates how different elements may be combined. As combinator libraries are a set of functions and operators over an abstract type, they lend themselves towards formulating axioms and equational reasoning, especially when implemented in an underlying language with a simple semantics. Once a set of axioms has been formulated, and verified by appealing to the underlying implementation, the user of the library can reason directly within the domain semantics rather than within the semantics of the underlying language. From such axioms an algebra of laws and identities can be constructed about the domain.

There are three language features that simplify the production of a combinator library: higher-order functions, polymorphic typing and laziness. Higher-order functions enable the encapsulation and naming of programming idioms in a way not possible in first-order

languages. Polymorphic typing allows the same programming idiom to manipulate data of different types. Lazy evaluation allows the programmer to abstract away from the execution behaviour: the same idiom can be reused in circumstances where all, some or none of the result it computes is required.

Together, these three features allow the identification and capture of the core idioms of a problem domain. Applications can then be implemented in a component-based programming style with extensive code reuse. It may be possible to write a similar library in a language lacking some or all of these features. However the implementation and use of the library will be more difficult.

5.1.2 Monadic Combinator Libraries

Irrespective of the functionality provided by different combinator libraries many have similar implementation issues. For instance, how to propagate data through the combinators; maintain an mutable store; accumulate output; handle non-determinacy and error-recovery.

At one time, the author of a combinator library had to solve each of these problems from scratch, usually in an ad-hoc manner. Then it was realized that monads could be used to provide these facilities in a structured and reusable way. A monad defines the computational model of the combinator library. The value of monads is the modularity they provide. By using a monad the underlying machinery of the computational model can be hidden and new features added with ease.

Definition of a Monad

Monads were introduced to the functional programming community by Wadler (Wadler, 1990; Wadler, 1992). The literature is extensive, ranging from their theory to design and practical use. A good introduction to monadic support in Haskell is given in (Hudak et al., 1999) while (Winstanley, 1999b) gives an informal introduction to monadic programming.

A monad comprises a type and two primitive functions over it, called *return* and *then*. For a type T, these operations will have the following signatures:

$$\begin{aligned} &\mathit{return} :: a \rightarrow T\ a \\ &\mathit{then} \quad :: T\ a \rightarrow (a \rightarrow T\ b) \rightarrow T\ b \end{aligned}$$

A value of type $T\ a$ can be thought of as a computation which returns a result of type a. The *return* function produces a computation that simply returns a value without doing

anything else: it *lifts* its parameter into the monad. Computations are sequenced together using the *then* operator. This takes the result of the computation passed as its first parameter and applies the second (function-valued) parameter to it. The result is a computation whose result is the value returned by the second parameter.

To qualify as a monad, the combinators must satisfy the following requirements:

$$
\begin{array}{ll}
return\ a\ `then`\ (\lambda x \rightarrow f\ x) & \equiv f\ a \\
m\ `then`\ (\lambda x \rightarrow return\ x) & \equiv m \\
m\ `then`\ (\lambda x \rightarrow f\ x\ `then`\ \lambda y \rightarrow g\ y) & \equiv (m\ `then`\ \lambda x \rightarrow f\ x)\ `then`\ \lambda y \rightarrow g\ y
\end{array}
$$

These laws state that the *return* function must be a left- and right-unit, and that the sequencing operator *then* must be associative

Monads in Haskell

It was realized that the encapsulation of state provided by monads could be used as the basis of an IO system for Haskell which would be more versatile than previous approaches to IO, some of which are reviewed in (Hudak and Sundaresh, 1989). Rather than add a single special feature to the language, the language was adapted so that monads could be defined in a systematic way.

The core of this was the extension of type classes so that *constructor classes* could be declared. While a type class is parameterized over a *type*, a constructor class is parameterized over a *type constructor*. The language standard defines the following class which captures the notion of a monad: (notice that the *then* operator is written as $\gg=$ in Haskell)

$$
\begin{array}{l}
\textbf{class}\ Monad\ m\ where \\
\quad return \quad :: a \rightarrow m\ a \\
\quad m \gg= k :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

Furthermore, a rich library of generic monadic combinators is provided by the language standard. Any type which is an instance of the *Monad* class can make use of these combinators. An example is the *sequence* function, shown below. This takes a list of monadic computations and combines them into a computation that returns a list of the results.

$$
\begin{array}{lll}
sequence & :: Monad\ m \Rightarrow [m\ a] \rightarrow m\ [a] & \\
sequence\ [] & = return\ [] & \\
sequence\ (m : ms) & = m & \gg= \lambda x \rightarrow \\
 & \quad sequence\ ms & \gg= \lambda xs \rightarrow \\
 & \quad return\ (x : xs) &
\end{array}
$$

This illustrates a valuable point – although monadic values represent computations, they are still first-class values and so can be manipulated, stored in lists and returned from functions.

The final addition to Haskell was a syntactic sugar for monadic programming. Called the 'do-notation', it replaces the (*return*, $\gg\!=$) functions with a sequence of generators delimited by the offside rule. The details of the transformation between the two forms is given in the language definition (Peyton Jones, 1999). However, it is possible to get an intuition of the translation by comparing code in both forms. For example, here is the previous function expressed in do-notation:

$$
\begin{aligned}
&\mathit{sequence}\ [] &&= \mathit{return}\ [] \\
&\mathit{sequence}\ (m : ms) &&= \mathbf{do} \\
&&&\quad x \leftarrow m \\
&&&\quad xs \leftarrow \mathit{sequence}\ ms \\
&&&\quad \mathit{return}\ (x : xs)
\end{aligned}
$$

The do-notation is the basis for the sugared form used to present the PEDL code in this thesis.

Using Monads in Combinator Libraries

A monad can be used in a combinator library to implement the underlying runtime system required by the abstraction. By varying the definitions of the type constructor and the monadic primitives, computational models can be defined that possess any combination of the implementation issues mentioned earlier: state, back-tracking, exception-handling and so on. Depending on the features present in the computational model other primitives will be provided: for instance to update a value, or catch an exception.

After these primitives have been provided, all the functionality of the monadic combinator library is implemented using them. Therefore, the combinators return monadic values – representations of computations in the model defined by the monad. The computational model can be altered or extended by redefining the primitives and the monad type. By restricting access to the internals of the monad to a small set of primitives, modifications can be made without needing to rewrite the existing combinators.

The modularity provided by monads in combination with the syntactic and library support of Haskell has provided a strong incentive to library implementors. The success of this approach is indicated by the number of modern combinator libraries implemented in a monadic style. Examples are parsing (Hutton and Meijer, 1998), GUI programming (Finne and Peyton Jones, 1995; Scholz, 1999), and constructing SQL queries (Leijen and Meijer, 2000).

Another benefit of using monads is that it enables a greater understanding of the properties and limitations of a combinator library. The implementation fits into an existing framework, and so can use the results of general monad research. Examples are research into equational reasoning techniques for monads (Meijer and Jeuring, 1995), and methods for composing monads rather than adding new features by hand (King and Wadler, 1993; Liang et al., 1995).

5.1.3 Domain Specific Embedded Languages

A *domain specific language* (DSL) (Bentley, 1986) is a programming language designed for a particular problem domain. Such a language is not necessarily general purpose – it aims is to accurately capture the semantics of its target domain. A well designed DSL allows domain applications to be developed quickly and correctly.

A recent development is *Domain Specific Embedded Languages* (DSELs) (Hudak, 1998). This is a technique for implementing – or at least prototyping – a domain specific language as a combinator library. The implementation is a pure embedding in the host language: for clarity and simplicity no preprocessing, macros or the like are used.

For a new DSL, as with any language, it is likely that there will be many refinements made to the original language design. Furthermore, much of the design of a language is routine: the syntax and semantics of strings or arithmetic for example. Implementing by embedding in a host language, as a combinator library, assists with both these concerns. Such a lightweight implementation is no barrier to redesign or experimentation with the language semantics. The rapid prototyping this allows can give new insight into the domain. The second concern is removed: any feature that is not novel can be inherited from the host language.

As a DSEL is implemented as a library of combinators, it can be hard to draw a clear dividing line between what is a combinator library, and what is a DSEL. The difference is in the emphasis of the work. Combinator libraries are intended to be used as a subordinate component of a larger application. DSELs tend to be fuller libraries that subsume the host language; the majority of the code is written in the DSEL combinators. Where the host language is used, it is clearly a component of the DSL program, rather than the other way round.

A DSELs should be seen as a real language, rather than a library with convenient syntactic sugar. Many are intended for use by domain experts. To some degree, it is possible for the programmer to learn and use the DSEL without much knowledge of the host language.

Examples

An early published example of DSEL techniques is the experiment conducted by DARPA which compared the prototyping abilities of various programming languages (Hudak and Jones, 1994). The task was to implement a 'Geometric Region Server'; part of the Haskell solution was structured as an embedded language in which the domain expert could express properties of regions. The solutions and development metrics for the different languages were independently assessed: it was found that the DSEL prototype took significantly less time to develop, and was easier to understand and modify.

More recent examples include the *functional reactive programming* (FRP) family of languages. These include FRAN (Elliott and Hudak, 1997) for graphics and reactive animation; FranTk (Sage, 2000) for declarative GUI-intensive applications; and Frob (Peterson et al., 1999) for expressing robot controllers. These language designs share the same framework, in which there are two key abstractions: behaviours and events. A behaviour is a value which continuously varies over time, while an event is a stream of discrete $(time, value)$ pairs. The framework provides a range of combinators to combine primitive behaviours and events to create complex behavioural networks. The value of this approach is the abstraction provided by lifting functions over values to functions over behaviours, so that time becomes an implicit parameter in expressions.

Another notable system is Lava99 (Bjesse et al., 1999); a DSEL for hardware design. Compared to the FRP family, it has a simple implementation as a set of monadic combinators. The source programs, which describe electronic circuits, are expressed in the do-notation. By executing the source in different monads – and therefore different computational models – a circuit can be simulated; netlists or VHDL output generated; or a proof constructed that can then be verified by a theorem prover.

5.1.4 Conclusion

By embedding a new language in an established host the implementation is greatly simplified. The language designer need only be concerned with the novel constructs of the language; much of the language framework can be inherited from the host language. Furthermore, as the language is expressed as combinators, there is no need to implement a parser. Although this is convenient, it also means that the language design is constrained by the syntax of the host language.

Because of the low implementation cost, it is feasible for the language to go through many iterations of redesign – the cost to experiment with new constructs is minimal. This exploratory method is very powerful, and led to many refinements of the languages used in PEDL.

Another benefit is the inheritance of the properties of the host language. In this case of Haskell, this means that embedded languages have a good semantic underpinning and that equational reasoning and transformations are tractable.

One disadvantage of embedding is that the domain experts who are the intended users of the language may not have sufficient knowledge of the host language. Even in the case of a self-contained DSEL where no code has to be written in the host language the programmer will be exposed to error messages from the compiler that refer to the underlying implementation, rather than the source program. Due to the extensive use of types, classes and monads to structure the embedding, understanding error messages often requires deep knowledge of Haskell and the combinator implementation.

Another question is the efficiency of DSEL implementations: there is an additional level or two of interpretation in the final executable. For DSELs that are development tools for a different target system – for example Lava99 and the PEDL languages – efficiency is not so important, as programs are simply models. In the case of languages where programs are intended to be repeatedly executed – for instance the FRP family – Hudak proposes partial evaluation of the interpreter with respect to its source as a solution. However a useful partial evaluator for Haskell has yet to be been produced. Nonetheless, as embedded languages can make use of all the tools of their host, any advances in optimisation of the host will also benefit the embedded language.

5.2 Embedding the PEDL Languages in Haskell

Motivated by the reasons described in the previous section, it was decided to implement the PEDL languages by embedding them in a host language. The source code of this implementation is available from (Winstanley, 2000a).

5.2.1 Why Haskell?

Many higher-order functional languages would have been suitable as the host. Haskell is one of the prevalent functional languages: it was chosen because the author has extensive experience in this language. In addition, by using the same language as my colleagues it is hoped it will be easier to compare, combine and utilize the staged programming system with their research on parallel algorithm derivation (O'Donnell and Rünger, 1995; O'Donnell and Rünger, 1997a), derivation structuring (O'Donnell and Rünger, 1997b; Goodman et al., 1998) and proof tools.

If these cultural issues had been ignored, we might have decided to implement using one of the dialects of ML. There are always trade-offs, and many points for consideration,

when choosing a host language. The main benefit would be be that, compared to the lazy semantics of Haskell, the strict semantics of ML are closer to those of the target language. However, ML lacks the operator overloading provided by Haskell's type classes. Monads can be defined in ML, but due to the lack of overloading a library of generic monadic combinators and the syntactic sugar of the do-notation is not possible.

Type classes are also used to represent constraints on the values passed to language constructs. This technique was developed after the the host language had been chosen. It is unknown to the author if it it possible to construct a similar solution within ML: probably different techniques would have to be used.

5.2.2 Implementation Aims

In strongly-typed languages the type checker catches many programmer errors. This encourages the programmer to apply the heuristic 'if it compiles, my program is correct'. Although it is unable to detect all errors, strong typing is a valuable programmer aid.

It is very well to describe the valid syntax and static semantics of a language in a definition document, but if an implementation accepts programs that are not well-formed the user will become confused and disoriented. We wish to produce a embedding of our languages in which their static semantics are encoded in the type system of the host language. In this way the host type-checker will detect invalid programs, so that the heuristic can still be used.

The techniques to do this for standard DSELs are well understood. However, for our case it is made more difficult by two unusual features of our system. The first is that the result of a derivation is an implementation in a language which lacks many of the features of Haskell; to make this feasible the language forms that occur must be restricted to a subset of Haskell that has a direct translation to our target language. The following section describes the method used to do this.

The second problem is that a parallel program will be expressed in a combination of a computation and a coordination language. The properties of each language must be represented in the host type system so that constructs from different languages can not be combined incorrectly. Furthermore, it was discovered that the derivation process was simplified by being able to mix languages from *adjacent* derivation stages in the same program. The issues and techniques used to achieve this are discussed in Section 5.3.

5.2.3 Controlling the Host Language

The final step of the PEDL system produces an implementation by translating a program in the intermediate language to the target language. The language targeted a the moment by our system, SAC, lacks many of the features of rich languages such as Haskell. In particular the type system is much simpler and does not permit polymorphism, partial application or higher-order functions.

Although these language features are powerful and expressive – indeed, they are used throughout the implementation of the embedded languages – they have no simple translation to SAC. Therefore they must either be transformed away as part of the derivation process or prevented from occurring in the first place.

The motivation of this research is to use functional languages to model and derive parallel algorithms, not to find a new compilation technique for Haskell. Therefore it was decided that these difficult-to-translate features should be disallowed altogether.

The limitations of SAC are reflected in the design and semantics of the PEDL languages (Chapter 4). For example, the design of SAC favours iterative programs rather than recursive ones. A repeated computation may be expressed in Haskell as a directly recursive function or by using a combinator such as *fold* or *map* that encapsulates a recursion pattern. In contrast SAC possesses `for` and `while` loop constructs similar to C. The PEDL languages provide iteration combinators (Section 4.1.3) that can be directly translated to SAC loop constructs. By providing these, it is hoped that the programmer will not be tempted to write recursive code unnecessarily.

However it is still possible for the programmer to introduce arbitrary Haskell code into a PEDL program. The language embedding has been designed to constrain the use of Haskell to forms that can be translated to SAC simply. Although these mechanisms restrict some illegal forms, it it not possible to protect against all difficult features. For instance class and instance declarations are disallowed; however there is no way to enforce this as they are external to the type system.

The simplest technique used is to define a new prelude for use when writing PEDL programs. This is a very restricted version of the standard prelude that omits many types and functions: for instance, as lists are not supported in SAC we omit all the standard list combinators.[1]

The following sections describe two of the other techniques used. The first uses a monad to introduce sequencing and reduce the amount of laziness that may occur in the language, while the second section describes how we restrict polymorphism.

[1] It would be desirable to omit the list type too, but due to its special syntax this is not possible

Computational Model

The embedded languages are structured using an environment (or reader) monad[2] which was named *Action*:

```
data Action r a = Action (ProgramData → IO a)

instance Monad (Action r) where
        ⟨standard environment & IO monad⟩
```

From the monad type it can be seen that a computation is a function from some *ProgramData* to a result in the IO monad; the IO monad is used so that we can provide basic input and output functionality for the languages. The *Action* type is parameterized over a type variable r. This variable encodes the properties of a language in the type system; its use is described in Section 5.3 and can be ignored for now.

```
data ProgramData = ProgramData {
                     partition :: [PID],
                     rank :: Maybe Rank,
                     ...
                     }
```

The combinators of the monad propagate a value of *ProgramData* through the embedded program. The *ProgramData* type contains environmental information about the parallel machine that the program is executing on. Amongst other information, it records the current partition and, for computational languages, the rank of the current processor.

The embedded language constructs all return an *Action* computation. By hiding the constructors of this type, the only way to access the result of a computation is by binding it to a variable in a block of do-notation. Therefore a PEDL program is *monad-bound*: there is no way to escape from the *Action* monad. The serialization produced by the monad gives the program a defined execution order – something that is often lacking in vanilla Haskell programs. Furthermore, the serialization prevents the programmer writing recursive expressions which rely on laziness for termination. This is valuable considering that at the later stages of the derivation the implementation will be strict.

Another benefit of the monadification is that it encourages a simple single-assignment form of programming, rather than deep nesting of expressions and combinators. This simplifies the transformation process.

[2]This is a simplification. The monad used in the implementation also provides simple tracing and logging facilities. As monads provide modularity these features to not impact on the core functionality described here

Phantom Classes

In Haskell all types are first-class – a value of any type can be passed to a function, stored in a data structure or communicated via a channel. In SAC this is not the case: there is a discrimination between functions and the types that may be passed between them. SAC has five basic types: integers, single and double precision floating point, booleans and characters; and a single datatype, the array. Arrays may only contain basic types. Nested arrays are not permitted.

If PEDL is to be easily translated to SAC, these restrictions must be reflected in the type system of the embedded languages. We do this by introducing new constraints into the host type system.

Leijen and Meijer (Leijen and Meijer, 2000) describes how to use polymorphic type variables to ensure the type correctness of untyped embedded SQL queries. These variables only occur in type signatures - a value of that type is never physically present in the program: this explains their name, *phantom type variables*. In a similar vein, we use *phantom classes* to control the polymorphism present in the host language. A phantom class is simply a class with no member functions. Therefore it does nothing, but can still be used to constrain a type variable in the signature of a combinator.

We declare a phantom class *Data*. This class enumerates the set of types that may be passed between functions and communicated between processors. Instances of this class are provided for the five basic types and array type as follows:

```
class Data a

instance Data Int
⋮
instance Data a ⇒ Data (Array ix a)
```

A subset of types in the *Data* class may be stored in arrays. Another class, *Storable*, is declared and provided with suitable instances to represent this.

```
class Data a ⇒ Storable a

instance Storable Int
⋮
```

After setting up this framework, the legal types that may be passed to a combinator can be specified by adding a phantom class context to its type. No change is necessary to the definition of the combinator. For example:

```
genarray :: Storable a ⇒ ix → (ix → Action r a) → Action r (Array ix a)
get      :: Data a ⇒ DVal a → Rank → Action r a
```

We can constrain the array generator *genArr* so that it can only create arrays of a basic SAC type. Likewise *get*, which performs an implicit communication, is restricted to distributed values whose elements are of a type that can be communicated. In this way, if the programmer writes a PEDL program that breaks the constraints of the SAC type system, it will be detected by the Haskell type checker.

5.3 Distinguishing Between Languages

An interesting difference from typical DSELs is that PEDL comprises a a sequence of language pairs, rather than a single language. A parallel program is expressed in a combination of a coordination and a computation language. Derivation of the parallel implementation proceeds by transforming the language pair of one derivation stage to the language pair of the next stage in the sequence. The languages share common constructs and types, but each stage of the sequence extends this core with constructs that give greater control over the parallel algorithm.

This raises the question of how the implementation of the languages should be organized. We wish to keep each pair of languages distinct from the others so that the stages of the derivation process are well defined. Furthermore, constructs specific to the computation or coordination layers must be prevented from occurring in the other layer, where they are invalid. This is complicated further by sets of constructs, such as looping combinators, that are valid in both layers.

The simplest solution would be to use the module system to provide a separate interface for each pair of combination and computation language. Shared language features could be placed in modules that were imported by the interface modules of the languages where they were present. Different monad types could be used for the coordination and computation layer, while constructs that were valid in both layers would be overloaded using type classes.

As only one pair of combinator languages would ever be imported into a program at a time the stages would remain distinct, while the differing monad types would ensure the layers did not become muddled. The programmer could be confident the program was expressed in a single pair of languages, rather than some combination of derivation stages.

However this approach was unsatisfactory: we realized the importance of being able to mix languages from different derivation stages within a single program. This was not so that the programmer may write code in a mixture of languages: instead it is an aid for performing derivation steps.

The purpose of the PEDL system is to derive parallel algorithms by transforming between languages. When transforming a program from one derivation stage to the next, it is inconvenient to require the program to be entirely rewritten in the next stage before the user can verify that the transformations have been applied correctly. For real-world programs, having to monolithically transform the *entire* code would be unworkable.

For each of the derivation steps, the computational models and constructs of the adjacent language pairs are so similar that a program expressed in a mixture of the languages can be considered to be well-defined. By producing an implementation that accepts these mixed-language programs, the programmer can use the parser and type-checker of the host language to verify that the program is syntactically and type correct. Furthermore, it is possible to execute mixed-language programs.

Mixing languages gives the programmer the ability to perform a gradual transformation between stages. This is another example of staging and decision delaying that we believe gives more opportunity for insight and experimentation – the whole of a derivation step does not need to be thought through before preliminary steps can be made and verified.

We could permit mixed-language programs with a module-structured implementation. If designed carefully, it would be possible to import the language interfaces for adjacent derivation stages into a program that was being transformed between them. However, the clean separation of stages is then compromised: this increases the risk of confusion and making mistakes during the transformation process. The language implementation can provide no protection from these errors. The only way to determine if the program had been fully and correctly transformed would be to remove the import of the old interface and hope for the best.

For mixed-language programs all the combinator languages must be visible in the namespace at the same time. To preserve the distinction between derivation steps we require a method to determine which language a block of code is expressed in, and a way of ensuring that constructs may only occur in languages where they are valid.

5.3.1 Encoding Languages as Phantom Types

We devised a system which encoded language information as a 'chain' of phantom types. Through this the type checker can then determine the language used for each program block and ensure that only valid combinations of constructs are used.

Each group of constructs is valid in a different subset of the PEDL languages. The most general are the loop combinators, which occur in any language, computational or coordination, at any stage. The array operations may occur at any stage, but only in the

computational language. Then there are combinators common to all the coordination layers; and constructs specific to one or two languages, such at **get**, which is later replaced by explicit collective communications.

Figure 5.1 illustrates this hierarchy of constructs. The root node represents the constructs common to all languages. Coordination languages extend this core with constructs such as **parallel** and **global**; in the redistribution stage of the derivation, the coordination language is further extended with collective communications. Computation languages possess combinators that manipulate arrays. A subset of the computation languages execute upon a processor, rather than as a specification. These languages add constructs such as *size* that return information about the machine configuration. Computational code for a parallel machine may either be executed as a parallel or global block. Each of these add further constructs.

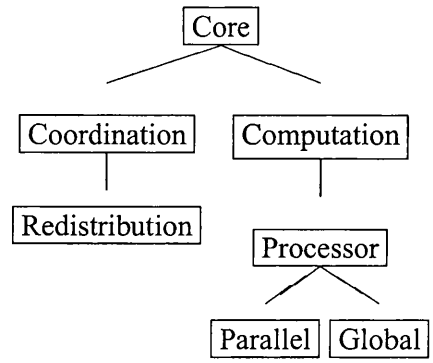

Figure 5.1: Hierarchy of constructs

If we view a language as a type, then each of the nodes in the tree defines a language subtype that supports the constructs of its parent nodes and adds constructs of its own. We have an inheritance hierarchy of languages.

Expressing inheritance using Phantom Types

The comparison with classes and inheritance gave an insight into how to represent languages within the type system. (Finne et al., 1999) use chains of phantom types to represent single-inheritance of interfaces to COM components. Their technique works as follows. For every kind of COM interface imported into Haskell, a new type is automatically generated. For instance, an address book interface could have a type like:

$$\begin{aligned} &\textbf{data}\ \mathit{IAddressBook}\ a\ =\ \mathit{IAddressBook} \\ &\mathit{openAddressBook}\quad ::\ \mathit{IO}\ (\mathit{Object}\ (\mathit{IAddressBook}())) \end{aligned}$$

These interface types are used to annotate untyped pointers to external COM objects. So when opening the address book component a pointer annotated by the *IAddressBook* interface type is returned. These types are also used in the signatures of the methods that are supported by an interface. In our address book example the method to add a new entry has the following type:

$$\mathit{addEntry}\ ::\ \mathit{Entry}\ \rightarrow\ \mathit{Object}\ (\mathit{IAddressBook}\ a)\ \rightarrow\ \mathit{IO}\ ()$$

	COM	PEDL
Carrier type	Object	Action monad
Chain of types represent	interface inheritance	inheritance of characteristics
Operations	interface methods	language constructs

Table 5.1: A comparison of inheritance in COM interfaces and PEDL languages

The interface type ensures that this method can only be called on objects that support the address book interface. However a value of the interface type never occurs in the program – they are phantoms that appear solely in the type system.

Imagine that an improved addressbook component is implemented. It supports the existing interface, but provides additional functionality for searching the entries. The new interface is represented by another type:

$$\begin{array}{l}\textbf{data}\ \mathit{ISearchable}\ a \qquad\qquad = \mathit{ISearchable} \\ \mathit{openSearchableAddressBook} :: \mathit{IO}\ (\mathit{Object}\ (\mathit{IAddressBook}\ (\mathit{ISearchable}\ ())))\end{array}$$

The object pointer returned by opening the improved address book component is now annotated by a chain of interface types. This chain represents the interface inheritance of the component. The additional methods provided by the of the extended address book have corresponding types:

$$:: \mathit{Query} \rightarrow \mathit{Object}\ (\mathit{IAddressBook}\ (\mathit{ISearchable}\ a)) \rightarrow \mathit{IO}\ [\mathit{Entry}]$$

The *findEntry* method can only be called on objects supporting the search interface. The methods of the simpler address book interface can also be called on this object: due to the polymorphism in the chain of interface types of their signature, methods of a parent interface can be used on objects of a subclass. However the reverse, which is incorrect, is prevented by the type system. The simpler addressbook interface type cannot be unified with the object type required by the *findEntry* signature, due to the use of the null tuple () to close off the inheritance chain.

Returning to the PEDL languages, we have observed that there is a inheritance relationship between the different classes of languages. We can represent this in the Haskell type system in a similar way to the COM interfaces using the mapping shown in Table 5.1. The following sections examine each part of the mapping in turn.

Carrier Type

The language constructs all return computations in the *Action* monad. By adding a parameter to the *Action* type that will contain a chain of phantom types each computation

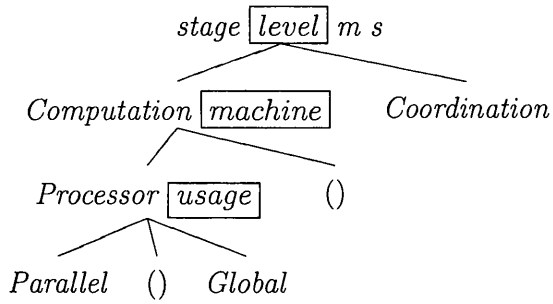

Figure 5.2: Hierarchy of language characteristic types

can be annotated with the language it was expressed in. This parameter, named r was shown in the definition of *Action* given earlier in Section 2.

Consider the type signature of the monadic sequencing operator when it is specialised to the instance for the *Action* monad:

$$(\gg\!=) :: Action\ r\ a \rightarrow (a \rightarrow Action\ r\ b) \rightarrow Action\ r\ b$$

It can be seen that although it can sequence computations that have different result types, the language type r must be consistent throughout. Therefore, the sequencing combinator may only combine computations expressed in the same language. Through this it can ensured that a block of code only contains compatible constructs.

Language characteristics

Unlike the COM interfaces, we have a fixed set of characteristics we wish to model. We can define phantom types for each of these, forming the tree structure shown in Figure 5.2. At the top level will be types representing the stage of the derivation. The stages are enumerated as follows:

```
data Specification level m s    = Specification
data Independent level m s      = Independent
data Distributed level m s      = Distributed
data Redistribution level m s   = Redistribution
data Intermediate level m s     = Intermediate
```

The *level* parameter in each type will contain further characteristics of the language. The m and s parameters have another purpose: in addition to describing the subtyping of a language, we will also record other information about a program block. Their use is explained in Section 5.3.2.

In many stages of the derivation, there are two language levels. These are represented with further types:

```
data Coordination          = Coordination
data Computation machine = Computation
```

No further information is required about coordination layer languages. However, there is a group of computation layer constructs that may only occur in languages that execute on a parallel machine. This gives another type *Processor*. Finally, there are some constructs that may occur only in a global block of processor computation, and some that occur only in a parallel block. The following types represent this:

```
data Processor usage = Processor
data Global          = Global
data Parallel        = Parallel
```

Using these types to describe language characteristics, each of the languages can now be defined as a type synonym. The first synonym below, for example, is the language type of the coordination layer of the independent decision stage. The second synonym is the language type of a parallel block of computational code in the distributed stage.

```
type IndependentProg m s a =
  Action (Independent Coordination m s) a
type DistributedParallelProg m s a =
  Distributed (Computation (Processor Parallel)) m s) a
```

The other languages of the two-level stages have similar types. We can also describe the specification language in this framework. It is a computational language, but does not execute on a parallel machine. We can disallow constructs that assume a parallel machine by plugging the subtyping 'hole' with the unit type ().

```
type SpecificationProg m s a =
  Action (Specification (Computation ()) m s) a
type IntermediateProg m s a =
  Action (Specification (Computation (Processor ())) m s) a
```

The intermediate language is more troublesome. It is a computational language on a parallel processor, but by this point in the derivation the differentiation between parallel and global blocks has been removed. We can express this by using () to prevent any constructs specific to these blocks.

This is correct for most cases. Unfortunately there are exceptions: some parallel-only constructs are carried through to the intermediate language. Section 5.3.1 demonstrates how these special cases are expressed without complicating the overall system.

Language Constructs

We have defined types that represent different language characteristics, and have a method that ensures that a block of code is expressed in a single language. It remains to define the language characteristics required by each construct. This is done by constraining the polymorphism of the language type chain sufficiently to restrict it to the subset of languages it is valid in. No changes are necessary to the definitions of the constructs.

Starting with the more general constructs first, we can state that array operations are permitted in any computation language by giving them signatures that constrain the language level to *Computation*. The other language characteristics are left fully polymorphic to allow these constructs to be used in any context. For example, the signature for the array indexing operation becomes:

$$(!) :: (Index\ i, Storable\ a) \Rightarrow Array\ i\ a \rightarrow i \rightarrow Action\ (l\ (Computation\ ll)\ s\ m)\ a$$

The **use** construct projects the local element of a distributed value. Therefore, it can only occur in a parallel block of computational code; its type states this.

$$\mathbf{use} :: Dval\ a \rightarrow Action\ (l\ (Computation\ (Processor\ Parallel))\ s\ m)\ a$$

Collective communications only occur in the coordination language of the redistribution stage. This more specific constraint is expressed by leaving less polymorphism in the phantom type:

$$\begin{aligned}\mathbf{broadcast} :: &\ Sendable\ a \Rightarrow Rank \rightarrow Communicator \rightarrow Dval\ a \\ &\rightarrow Action\ (Redistribution\ Coordination\ s\ m)\ (Dval\ a)\end{aligned}$$

We can also type the hook constructs that combine blocks in different languages. The **parallel** construct executes a block of parallel computation code and returns a distributed value containing the results back to the coordination layer. Notice that although the derivation stage l is left polymorphic, we require that the two language levels must agree to the stage.

$$\begin{aligned}\mathbf{parallel} :: &\ Data\ a \Rightarrow Action\ (l\ (Computation\ (Processor\ Parallel))\ s\ m)\ a \\ &\rightarrow Action\ (l\ Coordination\ s\ m)\ (Dval\ a)\end{aligned}$$

Typing programs

As the *Action* monad propagates the language type throughout a block of code, the constraints of each construct in the block are unified. Provided no direct conflicts occur –

which would indicate an incorrect program where languages have been combined illegally – the type of the block indicates the language, or class of languages, it is valid in.

For instance, although the following block of code contains an array operation, which may occur in any computational language, the possible language is further constrained by the *use* construct.

```
t1 dv = do
          a ← use dv
          a ! 3
```

Querying the type of this program in Hugs gives:

```
Hugs> :type t1
t1 :: (Index a, Storable b, Num a) => Dval (Array a b)
   -> Action (c (Computation (Processor Parallel)) d e) b
```

That is, this program is expressed in a computational language, executing on a processor in parallel. However it is not restricted to a particular derivation stage.

Correct programs can be passed to an interpreter for execution. The interpreter for each derivation stage uses the type synonyms shown earlier to describe the language it accepts. For instance, the interpreter for the independent stage only accepts programs written in the coordination language of that stage, as follows:

```
runIndependent :: Int    IndependentProg m s a    IO ()
```

Irregular Constructs

For the derivation stage, language or usage we can express either that a construct is valid only in a single alternative, or in all. We cannot express 'all but one'.

Some constructs require just that. For example the *get* construct may occur in a parallel computation block of any stage apart from the redistribution stage, where it is replaced by collective communications. Another problematic construct is *rank*, which may occur in any parallel computation block, and in the intermediate language, but not in a global computation block.

This may be a sign that the representation of languages we use is not the best possible design. However, adding types to represent these special cases causes the type model

to get more complex. There is a trade-off between modeling everything in a uniform manner and retaining usability and conciseness. Therefore we found a solution that was external to the type model: although maybe a little inelegant, our solution does not affect the constructs that fit into the type model.

We use phantom classes to express these 'all but' relations: we named them 'not-classes'. For instance to state that *rank* may appear anywhere but in a global block the following definitions are used:

```
class NotGlobal a
instance NotGlobal ()
instance NotGlobal Parallel
```

Instances are provided for all subtypes but the one we wish to prevent. Using these classes, we can express the restriction on the places where this construct may occur by adding a class context.

```
get  :: NotRedistribution l ⇒ Dval a → Rank
     → Action (l (Computation (Processor Parallel)) s m) a
rank :: NotGlobal ll ⇒ Action (l (Computation (Processor ll)) s m) Rank
```

This solution is a little counter-intuitive – it requires an instance definition for every subtype apart from the one we are really interested in. However, there is no increase on complexity other than the additional class context.

5.3.2 Another use of Phantom Types

Although the entire language is monad-bound, the majority of constructs are pure – they have no side effects. Furthermore, they cannot affect the state of the monad as it is a reader monad only, not a state transformer. It is safe to re-order such computations limited only by data dependencies.

However some computations do perform IO. The order of side-effecting computations cannot be swapped without altering the behaviour of the program. We could ensure this does not happen by unrolling the monadic combinators so that the IO actions are exposed. However this is inconvenient: we would like the user to be able to abstract away from the internals of the implementation. If program blocks that may perform IO could be detected, the user can ensure that their ordering is maintained without examining the underlying code. Fortunately phantom types can be used to achieve this.

As seen earlier, the types representing stages of the derivation system have two additional variables. The s variable is used to indicate side-effecting blocks. We define a new type that will denote a side-effecting computation.

$$\textbf{data}\ \mathit{SideEffect} = \mathit{SideEffect}$$

Side-effecting computations are expressed using the *doIO* primitive. This primitive lifts an IO computation into the *Action* monad. All side-effecting computations must be defined using *doIO*, as only this has access to the internals of the monad.

$$\begin{aligned}&\textbf{doIO} \quad :: \mathit{IO}\ a \rightarrow \mathit{Action}\ (l\ (\mathit{Computation}\ ll)\ \mathit{SideEffect}\ m)\ a\\ &\textbf{doIO}\ a = \mathit{Action}\ (\lambda\_ \rightarrow a)\end{aligned}$$

This primitive constrains the s variable to type *SideEffect*. Any side-effecting computations defined using it will share this behaviour. Constructs that do not perform IO leave the s variable fully polymorphic and unconstrained.

As the language types are unified within a block of code, the type of s for any block containing a side-effecting computation will therefore be *SideEffect*; for blocks of pure code, s will still be polymorphic. In this way the user can inspect the type of a block to determine whether special treatment is required.

This feature can also be used to prevent IO where it is not well defined. For instance, array generator bodies in SAC have no defined order of execution – this gives the compiler greater opportunities for optimisation. We can prevent the use of side-effecting computations in PEDL array generators by constraining the s variable to () for the generator body only.

This simple technique works well. Its one weakness is that a user may inadvertently give a type signature that over-constrains the s variable, so that a false positive is given. However this will not affect the semantics of the program – merely restrict the freedom with which the program can be manipulated. The dangerous false negative cannot occur. If the user does not over-constraint the s variable, type inference will not introduce false positives itself.

5.3.3 Mixing Languages

For flexibility we wish to allow constructs from other derivation stages to be embedded within a program whilst it is being transformed. However, we must ensure that these mixed language programs are still well-formed. For instance, while computational code from one stage can safely be substituted for computational code from another stage, we cannot allow coordination level constructs to occur where computational code is expected. Similarly, parallel processor constructs should not occur within a global computation block: this would allow access to the elements of a distributed value, breaking the barrier between the two language levels.

We can define a primitive that allows languages from different stages to be safely mixed.

```
data Mixed      = Mixed
mix             :: Action (l b s m) a → Action (l' b s Mixed) a
mix (Action m)  = (Action m)
```

The parameter to *mix* is a computation expressed in the language of a different derivation stage. This parameter is unpackaged and then repackaged in the *Action* monad: this destroys the existing type information and allows us to invent a new language type for the computation that fits with the language type of the enclosing code.

The type variable representing the derivation stage differs between the embedded and the enclosing program blocks – l, l'. This allows code from different derivation stages to be mixed. However the language level and usage – represented by the b variable – of the computations must be consistent. This means that two coordination blocks may be mixed, as can two global or two parallel blocks. However, different kinds of program block cannot be mixed.

The side effecting variable s is propagated out to the enclosing block – no matter what the language, a side effect is still a side effect. The final variable m is used to indicate which program blocks contain mixed code. This works in the same way as the side-effecting variable described in the previous section. We indicate that the enclosing block is mixed by constraining its type to *Mixed*. This annotation will propagate throughout any program that this block is used in, indicating to the user that the program is currently between derivation stages.

Below is an example of the use of *mix*. The outer block is coordination level code that, due to the occurrence of a collective communication, must be in the redistribution stage of the derivation. Within it is a block of computational code containing the *get* construct. This construct should not occur in the redistribution stage, However, the two stages can be combined using *mix*: this program will now compile and run correctly, while its type indicates that is expressed in a mixture of derivation stages.

```
t4 comm = do
  dv ← parallel rank
  dv' ← broadcast 1 comm dv
  parallel (mix ((get dv 1)
   :: Action (Distributed (Computation (Processor Parallel)) s m) Rank))
  ⋮
```

```
Hugs> :type t4
t4 :: Communicator
   -> Action (Redistribution Coordination b Mixed) (Dval Rank)
```

This example illustrates a disadvantage of *mix* – a lengthy type annotation is required to resolve any not-class constraints on the derivation stage variable of the language type. We can define a new operator that provides a more convenient way to describe the derivation stage a computation is expressed in.

```
infix 0 ⌈
(⌈)              :: Action (l b s m) a → l b s m → Action (l' b s Mixed) a
act ⌈phantom     = mix act
```

This operator takes the foreign-language computation and flags it with the constructor of the derivation stage type the computation is expressed in. As these constructors have the same name as the type itself, it appears as an abbreviated type annotation. Only the derivation stage needs to be specified – the other language characteristics are constrained by the enclosing program block. Using this operator, the previous example can be rewritten as:

```
t4 comm = do
  dv ← parallel rank
  dv' ← broadcast 1 comm dv
  parallel (get dv 1 ⌈Distributed)
  ⋮
```

This notation is more concise and less error prone than using **mix**. Another benefit is that the user must now consciously assert which derivation stage a block of code is in.

5.4 Summary

This chapter described the development of the method of implementing languages by embedding them in a host. This is an economical and versatile way to develop a language design. Monads were presented as a technique for structuring computations, which could be used as the underpinning for an embedded implementation. We justified our choice of Haskell as a host language and then sketched the method used to implement the PEDL languages as Haskell combinators.

There are two features peculiar to the PEDL languages. The first is that the features of the host language must be constrained to a subset that is easy to translate to our target language. The second is that a program is expressed in a combination of two languages. We found that the type system of the host language could be used to control the occurrence of hard-to-implement host language features, and that it could impose a structure on the combination of languages.

A hierarchy of phantom types were used to represent the classes of related language features. Through this, the type of a block of code indicates which class of languages it

is valid in. The allows the type checker to detect malformed programs. Phantom types are also used to record other program properties, such as the presence of side-effects.

It was found that the phantom language type system was flexible enough to allow code blocks from different decision stages to be mixed in a single program, but only in combinations that were well defined. This allows mixed language programs to be checked and executed, which was found to be a great advantage when performing program derivations.

At present the languages of the derivation stages are simulated on a uniprocessor. A extension to the current PEDL implementation would be to execute programs in these languages in parallel. This could allow the programmer to gain insight into the parallel behaviour and performance. For the more abstract stages of the system the constructs of the coordination language could be implemented in terms of the *seq* and *par* combinators of GpH. The implicit paralleism and dynamic nature of GpH is well suited to executing such programs, whose parallelisation is only partially specified. Furthermore the GpH simulator, GRANSIM (Hammond et al., 1994), could be used to visualise the parallel behaviour. The intermediate level language does not have a coordination layer and so would not be suitable for this method. However the communication requests of this language can be implemented using a Haskell binding to MPI such as hMPI (Weber, 2000). These intermediate level programs would have the same parallel behaviour as the final implementation, but the executable on each processor would be generated by a Haskell compiler rather than a C compiler.

Chapter 6

Using PEDL

Capsule

This chapter illustrates the use of the PEDL system. Two case studies are presented, from which the different stages of the derivation process can be seen. Each case study commences with an abstract specification and progresses though the user directed decision stages until a full specification of the parallel implementation is produced. Once the derivation stages of the process are completed, we informally illustrate the code generation transformations that produce the final implementation – the details of this process are formalised in the following chapter. The generated SAC programs are clear and well structured and have been executed in parallel.

Introduction

This chapter demonstrates the use of the PEDL system. We start with a case study that illustrates the main features and ideas of the system. The algorithm studied is the simple one-dimensional wave equation. Then Section 6.2 produces a parallelisation of the maximum segment sum problem using parallel scan and fold operations.

Each case study starts with an abstract specification and proceeds through the decision stages until a fully specified parallelisation results. Once the derivation stages of the process are completed, the process of simplification and transformation that generates the final implementation is illustrated. The details of this process are presented in Chapter 7. The result of the case studies are clear, conventional SAC programs: these have been executed in parallel on a network of workstations.

6.1 One Dimensional Wave Equation

A range of physical systems can be simulated by an algorithmic pattern commonly referred to as the 'wave equation'. We will develop a parallel implementation for the one-dimensional version of this algorithm. The algorithmic pattern simulates how the state of an area of space varies through time. The area of space is represented as a sequence of evenly-distributed points. Likewise, time is divided into small increments.

Let $\mathbf{w}^t$ be the sequence of point states at time t, where $\mathbf{w}_i^t$ denotes the state of the i^{th} point in the sequence at time t. The state of a point p at time $t+1$ is described by an element function f of the previous state of p and the points adjacent to it. That is:

$$\mathbf{w}_p^{t+1} = f(\mathbf{w}_{p-1}^t, \mathbf{w}_p^t, \mathbf{w}_{p+1}^t)$$

The simulation proceeds by calculating a new sequence of point states for each time step in turn. This is achieved by applying the function f to each element in the sequence of the previous time-step. For the elements at the start and end of the sequence constant values are used in place of the values of the missing neighbours.

The inputs to the algorithm are a sequence of starting states for the points, constant values for the left and right boundaries, an element function f, and the number of time steps n to run the simulation. The result of the simulation is the sequence of states after n time steps.

By varying the definition of the element function this algorithmic pattern can model, for example, transmission of heat along a one-dimensional solid; or propagation of a wave though space. Further systems can be simulated by altering the parameters to the element

function. For instance, the value of the local element at the previous two time steps, or the values of additional neighbours on each side may be passed to the function. However the principle of the algorithm remains the same.

The following sections illustrate how this algorithm is parallelised as it progresses through the stages of the PEDL system.

6.1.1 Specification Stage

The obvious way to implement this algorithm is to represent the sequence of point states as a vector. For each time step, the element function is applied to every point in the previous vector, resulting in a new vector. This suggests an external loop over time steps, and an internal loop over the elements of the vector. The following code is an executable specification of the algorithm. The computation *waveEq* is the loop over time steps; for modularity the inner loop is defined as a separate function *oneIter*.

```
waveEq f n bl br v = do
  b ← bounds v
  loopnaccum n v (λ vect → oneIter f bl br b vect)
```

The parameters to *waveEq* are the element function f; left and right boundary values bl,br; a vector v of the starting state of each of the points; and n, an integer giving the number of time steps that the system is to be simulated for.

The body of *waveEq* is a loop combinator **loopnaccum**. This executes its loop body n times, threading an accumulating value through the iterations. That is, the result of one loop iteration becomes the input to the next. The vector of starting states v is passed as the initial value of this accumulating value. The result of the loop computation is the value computed by the final iteration of the loop body: this forms the result of the *waveEq* computation.

The loop body is an abstraction over a call to the *oneIter* function. This accepts the state vector for one time step and returns the vector for the next time step. This function takes similar parameters to *waveEq* function: however the bounds (or size) of the vector is also passed in.

```
oneIter f bl br b vect = genarray b (λ i → do
  l ← if i > 1 then vect ! (i − 1) else return bl
  c ← vect ! i
  r ← if i < b then vect ! (i + 1) else return br
  f l c r)
```

The **genarray** construct computes a vector of point states. Its parameters are the upper bound of the vector and an abstraction to execute for each point in the vector. This

abstraction computes the inputs to the element function f, and then calls this function to return the new state for this point. The inputs to the element function are called *l*eft, *c*entre, and *r*ight. The conditional expressions for the left and right values test whether the element being computed is at either limit of the vector. If this is the case then one of the constant values is returned; otherwise the neighbouring element in the previous vector is indexed.

These two definitions give a complete description of the wave equation algorithm. By defining a harness program that initialises some input data, calls the wave equation computation and prints the result, the wave equation program can be executed and tested. The same applies for successive stages of the derivation. The full sources for all the programs in the derivation, and their execution logs, are given in Appendix A.1.

6.1.2 Independent Computation Stage

The first decision stage requires the programmer to identify the maximum amount of useful parallelism. The program is transformed so that it is expressed in a combination of the independent coordination language and the associated computation language. At this point in the derivation an unlimited number of processors are available. Therefore the partition size is determined by the problem size, rather than any limits of the parallel machine.

Inspecting the algorithm, it can be seen that there is a data dependency between iterations of the external loop, as the element vector for time $t + 1$ depends on that for time t. However, the computation of point states within a single time step are independent from one another: they only depend on values from the previous time step. Exploiting this independence produces a parallelisation where the outer loop remains the same, but the loop body now computes each new point state in the sequence in parallel. This is illustrated in the following code. Note that the definitions of *waveEq* and *oneIter* have been combined in this program – this simplifies the presentation of the later stages.

```
waveEq f n bl br iv = do
  b ← global size
  loopnaccum n iv (λ ivect → parallel
      i ← rank
      l ← if i > 1 then get ivect (i − 1) else return bl
      c ← use ivect
      r ← if i < b then get ivect (i + 1) else return br
      f l c r)
```

The sequence of point states passed into the wave equation function is now represented as a distributed value rather than a vector. This allows each element to be computed in

parallel. The function changes to accommodate this – instead of querying the bounds of a vector, we perform a global computation to find the size of the partition the program is executing on. As we have unlimited processors in this computational model, the size of the partition is therefore also the number of point states in the distributed value iv.

The loop body is a parallel computation, rather than an array generator. Each processor computes a single point state in the sequence. First the rank of the current processor is found – this is analogous to the index value i in the previous program. The values for l, c, r are then calculated and passed to the element function f. The computation of the parameters to the element function is similar to that in the previous program, but now must access elements of a distributed value rather than elements of an array. Therefore **use** is used to access the local element of the distributed value, while the indexes resident on adjacent processors are accessed using **get**.

It can be seen that the transformed program is in essence unchanged: all that has been done is to replace one data type, the array, and it corresponding operations with another, the distributed value.

6.1.3 Distribution Stage

The next decision stage in the series tailors the idealized parallel program to the limitations of the target parallel machine. The number of available processors is fixed; therefore the parallel data structures must be distributed between the processors such that good performance results.

For the target machine, it is likely that there will be a large factor fewer processors than simulation points. This means that points must be 'doubled up' so that each processor computes more than one point. The best data distribution to use for this algorithm is a block-wise decomposition. The access pattern for the distributed sequence of points is such that only values of neighbouring points are required by each computation. A block-wise decomposition ensures that for the majority of points the required data will be located on the same processor.

```
waveEq f n bl br block iv = do
  b ← global size
  loopnaccum n iv (λ bvect → parallel
      i ← rank
      l ← if i > 1 then do
                          a ← get bvect (i − 1)
                          a ! block
          else return bl
      c ← use bvect
      r ← if i < b then do
                          a ← get bvect (i + 1)
                          a ! 1
          else return br
      oneIter f l r block c)
```

We can represent the block-wise data pattern as a distributed value of vectors. The distributed value has one element per processor: each element is a vector containing the block of points assigned to that processor.

In the distributed computation version of the program above, the *waveEq* function has an additional integer parameter: the size of the blocks. It is assumed the blocks on each processor are of an equal size. Each processor now computes a block of point states for each time step. Therefore the body of the parallel computation returns a vector – the new block for this processor. This block of point states is computed sequentially by the method used in the specification stage. In fact, we reuse the *oneIter* function defined in the specification here to compute the block.

The rest of the loop body calculates the parameters to pass to the *oneIter* function. The center value is the block resident on the current processor. The left and right boundary values for the block computation are the end values of the blocks on the adjacent processors. Provided the current processor is not at one of the limits (in which case the limit values are used), the values of l and r are computed by **get**-ing the block vector from the adjacent processor, and then indexing to return either the first or last element of this block.

6.1.4 Explicit Communication Stage

The final decision stage of PEDL removes the implicit communications expressed using the **get** construct. Instead of processors accessing non-local data directly, collective communications must be used to redistribute the distributed values so all elements are local to the processors where they are required. To this end the **get** construct is removed from the computational language of this stage. However, the **use** construct is retained.

This construct performs no communication and is necessary to access the local element of a distributed value.

```
waveEq f n bl br block bv = do
  b ← global size
  grp ← global currentGroup
  comm ← communicator grp
  loopnaccum n bv (λ bvect → do
      lastVals ← parallel {a ← use bvect; a! block}
      firstVals ← parallel {a ← use bvect; a ! 1}
      ldata ← shift NonPeriodic Inc 1 comm lastVals
      rdata ← shift NonPeriodic Dec 1 comm firstVals
      parallel
          i ← rank
          l ← if i > 1 then use ldata else return bl
          c ← use bvect
          r ← if i < b then use rdata else return br
          oneIter f l r block c)
```

By inspecting the wave equation program of the previous stage it can be seen that the only data communicated is the limit elements of each block vector. The pattern is an exchange of these elements between adjacent processors: this can be implemented by a shift to the right followed by a shift to the left.

The communication stage program above first defines a communicator to use in the shift operations. A communicator is an MPI type that provides a context for a communication and defines the group of processors that participate in the communication and their ordering. A communicator is created from a group in a collective operation across all processors.

The ordering of processors in the partition is used to define which are neighbours and so exchange data. As all the processors in the partition will participate in the shift communications, the communicator in the program can be created directly from the group that describes the current partition. This group value is part of the information provided to the program about the machine environment and is accessed by the **currentGroup** construct.

The first two lines of the loop body define the values to communicate. In parallel, each processor accesses its local block vector and projects either the first or last element in the block. The result of these parallel computations are distributed values of the leftmost or rightmost elements on each processor.

Two shift operations are then performed to exchange these elements between adjacent processors. This results in two new distributed values (*ldata*,*rdata*) where the leftmost and rightmost values are resident on the processors where they are required.

The shift collective communication shifts elements of a distributed value according to the processor ordering defined by the provided communicator. In addition to the communicator and data, this operation takes three further parameters – a periodicity, a direction and a displacement value. The periodicity indicates whether the shift operation 'loops around' from the last ranking processor back to the first. If, as in this program, the shift is *NonPeriodic*, then the element of the resulting distributed value for the first ranking processor is undefined. The direction specifies whether to shift elements in an increasing or decreasing order along the processor ranks, while the displacement is the number of processors in that direction to shift the data.

After the data has been redistributed the main computation can be performed. This is similar to the loop body of the previous stage. In parallel the boundary values for the block (l,r) are computed by either accessing the local element of the shifted distributed values, or by using the constants if the processor is at one of the limits. As before, the *oneIter* computation is then called to compute the new block vector.

6.1.5 Intermediate Form

At this point all the parallelisation and algorithmic details required by the final implementation have been specified. The following stages prepare for the translation to the target language. The first step in the process is to factor out the coordination language layer to leave a processor-view program that matches the computational model of the target language. This is achieved by the *change of view transformation* which is explained in Section 7.1.

Applying the transformation results in the following program. It can be seen that the basic structure is retained, although the transformation introduces redundant computations. These inefficiencies can be removed as part of the translation to SAC, or they can be left for the optimising SAC compiler to eliminate The operations of the collective layer are replaced with requests to the underlying communication system to perform the operation.

```
waveEq f n bl br block bv = do
  b    size
  grp    currentGroup
  comm ← communicatorRQ grp
  loopnaccum n b (λ bvect → do
      lastVals ← do{a ← return bvect; a ! block}
      firstVals ← do{a ← return bvect; a ! 1}
      ldata ← shiftRQ NonPeriodic Inc 1 comm lastVals
      rdata ← shiftRQ NonPeriodic Dec 1 comm firstVals
      do
        i ← rank
        l ← if i > 1 then return ldata else return bl
        c ← return bvect
        r ← if i < b then return rdata else return br
        oneIter f l r block c)
```

6.1.6 Simplification

The generation of the target language code splits into two steps. The majority are source-to-source transformations that are applied to the intermediate code, resulting in another intermediate level program. These simplify the program to remove any language features that are not supported by the target language, in this case SAC. Problematic features include function-typed parameters, nested blocks and function definitions and anonymous functions. The details of the simplification process are presented in Section 7.2. The result of the simplification process is the following code:

```
cBLOCK      = 4 :: Int
f l c r     = ...

oneIter bl br vect = do
  result ← genarray cBLOCK (λ i → do
  l ← if i > 1 then vect ! (i − 1) else return bl
  c ← vect ! i
  r ← if i < cBLOCK then vect ! (i + 1) else return br
  element ← f l c r
  return element)
  return result
```

```
waveEq n bl br bv = do
  b ← size
  grp ← currentGroup
  comm ← communicatorRQ grp
  vect ← loopnaccum n bv (λ bvect → do
      lastVals ← bvect ! cBLOCK
      firstVals ← bvect ! 1
      ldata ← shiftRQ NonPeriodic Inc 1 comm lastVals
      rdata ← shiftRQ NonPeriodic Dec 1 comm firstVals
      i ← rank
      l ← if i > 1 then return ldata else return bl
      c ← return bvect
      r ← if i < b then return rdata else return br
      result ← oneIter l r c
      return result)
  return vect
```

At this stage, further optimisations can be performed, such as removing dead bindings; inlining functions; and factoring constant code out of loop bodies. Further research is required to ascertain which optimisations would improve the code generated, and which are already performed by the optimisation pass of the SAC compiler. For clarity, no optimisations will be applied to the example program.

6.1.7 Translation to SAC+MPI

The final step is to translate to SAC. Details of the translation rules are given in Section 7.4. For many constructs the translation is purely syntactic. However, here are three areas of the translation that are a little more complex: loops and array combinators; communications; and array indexes.

One of the decisions taken during the design of the PEDL languages of the system was that arrays would have an origin of 1. However, SAC follows the C tradition of indexing arrays from 0. This means that the values used in indexing and conditionals often have to be adjusted by subtracting 1. This ad-hoc approach is temporary: a better solution would be to redesign the PEDL array combinators to more closely resemble those of SAC.

```
#define BLOCK 4

inline float f(float l, float c, float r){
...
}

float[] oneIter(float bl,float br, float[] vect){
  result = with ( . <= i <= . ){
    if (i[[0]] > 0) {l = vect[i-1];} else {l = bl;}
    c = vect[i];
    if (i[[0]] < BLOCK - 1) {r = vect[i+1];} else {r = br;}
    element = f(l,c,r);
  } genarray([BLOCK],element);

  return (result);
}

float[] waveEq(int n, float bl, float br, float[] bv){
  b = size();
  grp = currentGroup();
  comm = communicator(grp);
  _loopcount = n;
  bvect = bv;
  while (_loopcount > 0){
    lastVals = bvect[[BLOCK-1]];
    firstVals = bvect[[0]];
    ldata = Shift_float(false,1,1,comm,lastVals);
    rdata = Shift_float(false,-1,1,comm,firstVals);
    i = rank();
    if (i > 0)   {l = ldata;} else {l = bl;}
    if (i < b-1) {r = rdata;} else {r = br;}
    result = oneIter(l,r,bvect);
    bvect = result;
    _loopcount--;
  }
  vect = bvect;
  return (vect);
}
```

The translated wave equation program is presented above. Notice that the structure is very similar to the intermediate language program, although the syntax is quite different in places. This program compiles and executes successfully in parallel: it has been tested on a network of workstations.

6.2 Maximum Segment Sum

We now present a PEDL parallelisation of the *Maximum Segment Sum* (MSS) problem (Bentley, 1984). This algorithm has been studied extensively in the literature. The purpose of this section is not to perform a novel parallelisation of this algorithm, but to demonstrate that PEDL allows algorithms to be parallelised in a clear, concise and manageable manner. The full sources for all the programs in this derivation, and their execution logs, can be found in Appendix A.2.

Given a sequence of numbers $X_i \ldots X_n$ the task is to find the largest possible sum of a continuous segment within X. For example, $mss(\{2, -4, 2, -1, 6, -3\})$ would compute 7. The maximum segment sum can be computed in the following way:

1. compute $S_i = \sum_{j=0}^{i} X_j$ for $1 \leq i \leq n$
2. compute $M_i = \max_{i \leq j \leq n} S_j$ for $1 \leq i \leq n$
3. compute $B_i = M_i - S_i + X_i$ for $1 \leq i \leq n$
4. compute $mss = \max_{1 \leq i \leq n} B_i$

This algorithm can be implemented in conventional Haskell as follows. The sequence of numbers is represented by a list. Each intermediate result is computed using one of the standard list combinators. The first two follow the pattern of a scan (or multiprefix operation). The element-wise combination of the three lists can be expressed using *zipWith3*. The final result is then computed by a fold (or reduce) to find the maximum. That is:

```
mss    :: [Int] → Int
mss x  = let s   = scanl1 (+) x
             m   = scanr1 max s
             b   = zipWith3 (λ mi si xi → mi − si + xi) m s x
             mss = foldr1 max b
         in mss
```

6.2.1 Specification Stage

The MSS algorithm can be expressed in the specification language of PEDL in a form very similar to the conventional Haskell program. The main difference is that arrays rather than lists are used as the underlying data structure. The computational language provides libraries of whole-array combinators defined using the looping constructs and

array primitives. These have similar functionality to their corresponding list combinators. The specification stage program is as follows:

```
specMss :: Vector Int → SpecificationProg m Int
specMss x = do
              s   ← scanlArr1 sumOp x
              m   ← scanrArr1 maxOp s
              b   ← mod3Arr (λ mi si xi → return (mi − si + xi)) m s x
              mss ← foldArr1 maxOp b
              return mss
```

6.2.2 Independent Computation Stage

This algorithm can be simply parallelised by placing each element of the sequence on a separate processor. The entire sequence now forms a distributed value, rather than an array. The type of the program changes to reflect this:

```
independentMss :: Dval Int → IndependentProg m (Dval Int)
independentMss x = do
                     s   ← scanlDval1 sumOp x
                     m   ← scanrDval1 maxOp s
                     b   ← parallel
                               mi ← use m
                               si ← use s
                               xi ← use x
                               return (mi − si + xi)

                     mss ← foldDval1 maxOp b
                     return mss
```

It remains to define the recursion combinators – scan, fold and map – for distributed values. The element-wise map across the three sequences can be expressed trivially using the **parallel** construct, as seen above. For every processor the local elements of the three sequences are projected, and then combined to produce the resulting distributed value.

We define the reduction operations separately. All have the same general form, a loop whose body is a parallel operation over the elements of the distributed value. An intermediate result is accumulated through the iterations of the loop. At each iteration, an offset is calculated. This is used to access the element resident on a distant processor, which is then combined with the local element to produce the next intermediate value. The pattern of non-local accesses over the iterations form a tree structure.

```
scanlDval1 op dv = do
  n ← global size
  rs ← parallel rank
  forAccum (0, ceiling (log (fromInt n)), 1) dv
      (λ j b → parallel
                  i ← use rs
                  bi ← use b
                  if i > 2^j
                    then do
                           xi ← get b (i − 2^j)
                           xi `op` bi
                    else return bi)

scanrDval1 op dv = do
  n ← global size
  rs ← parallel rank
  forAccum (0, ceiling (log (fromInt n)), 1) dv
      (λ j b → parallel
                  i ← use rs
                  bi ← use b
                  if i < (n + 1) − 2^j
                    then do
                           xi ← get b (i + 2^j)
                           bi `op` xi
                    else return bi)
```

The fold operation is implemented in the same way as the scan operations, but with an extra clause in the conditional so that only the intermediate values required for the final result are computed, rather than all processors participating at all stages. The result of the reduction is left in the first element of the distributed value – the other elements are intermediate values used in the computation.

```
foldDval1 op dv = do
  n ← global size
  rs ← parallel rank
  forAccum (0, ceiling (log (fromInt n)), 1) dv
      (λ j b → parallel
                  i ← use rs
                  bi ← use b
                  if i < (n + 1) − 2^j && i  mod 2^(j+1) == 1
                    then do
                           xi ← get b (i + 2^j)
                           bi `op` xi
                    else return bi)
```

The next stage in the PEDL system maps the idealised parallelisation onto a machine with a limited number of processors. This is done by introducing a data distribution where each processor possesses more than one element of the sequence. In short, a distributed value of elements is replaced by a distributed value of vectors of elements. The operations over the distributed value are then adjusted to map across these vectors.

However, in this case study we will omit this stage – it would introduce a block-wise decomposition which has already been seen in the previous case study, and only serve to complicate the presentation of later stages. Instead, we make the simplifying assumption that the number of processors in the target partition is equal to the size of the input sequence to the MSS algorithm.

6.2.3 Explicit Communication Stage

The final decision stage replaces the implicit communication performed by the get construct with explicit collective communications. The fold and scan operations must be transformed; however, the main algorithm remains essentially unchanged. The only addition is the creation of a communicator which is then passed to the new versions of the reduce operations.

```
redistMss :: Dval Int → RedistributionProg m (Dval Int)
redistMss x = do
                grp  ← global currentGroup
                comm ← communicator grp
                s    ← redistScanlDval1 sumOp comm x
                m    ← redistScanrDval1 maxOp comm s
                b    ← parallel
                         mi ← use m
                         si ← use s
                         xi ← use x
                         return (mi − si + xi)

                mss  ← redistFoldDval1 maxOp comm b
                return mss
```

The reduction operations are implemented in a similar way, and therefore exhibit similar communication patterns. For brevity, the communicating version of only one is presented. The access pattern at each loop iteration is regular: processors use the data from the processor a fixed offset along the sequence. The data redistribution required can be realized by a shift operation at each iteration of the loop.

```
redistScanlDval1 op comm dv = do
  n ← global size
  rs ← parallel rank
  forAccum (0, ceiling (log (fromInt n)), 1) dv
      (λ j b → do
                 x ← shift NonPeriodic Inc 2^j comm b
                 parallel
                     i ← use rs
                     bi ← use b
                     if i > 2^j
                       then do
                              xi ← use x
                              xi `op` bi
                       else return bi)
```

The shift operation involves all processors in the partition. However, not all processors use this data to compute the result: at iteration j, ranks above 2^j have already computed their final result. It may seem that this is inefficient – more messages are being passed through the communication network than are required. The alternative would be to define a new communicator at each iteration which contained only the processors that required the data. However, as creating a communicator within MPI is a synchronous operation, which requires communication between processors to negotiate a new context, it is doubtful that an improvement in efficiency would be seen.

6.2.4 Intermediate Form

All the parallelisation details have now been provided by the programmer. The process of generating target language code begins. After removing the coordination layer from the program and simplifying a little, the following intermediate stage program results:

```
intermediateMss :: Int → IntermediateProg m Int
intermediateMss x = do
                      grp  ← currentGroup
                      comm ← communicatorRQ grp
                      s    ← intermediateScanl1 sumOp comm x
                      m    ← intermediateScanr1 maxOp comm s
                      b    ← return (m − s + x)
                      mss  ← intermediateFold1 maxOp comm b
                      return mss

intermediateScanl1 op comm dv = do
  n ← size
  i ← rank
  forAccum (0, ceiling (log (fromInt n)), 1) dv
      (λ j bi → do
                  xi ← shiftRQ NonPeriodic Inc (2^j) comm bi
                  if i > 2^j
                    then xi `op` bi
                    else return bi)
```

The other scan and fold operation are similar and are omitted.

6.2.5 Translation to SAC+MPI

The intermediate stage program can be simplified a little further, and then straightforwardly translated to SAC to yield the final implementation.

```
int intermediateScanl1(Mpi_Comm &comm, int dv){
  n = size();
  i = rank();
  bi = dv;
  for (j=0;j < ceiling(log(n)); j++){
    xi = Shift_int(false,1,pow(2,j),comm,bi);
    if (i > pow(2,j))
      {bi = xi + bi;}
  }
  result = bi;
  return(result);
}
.
.
.
int intermediateMss(int x){
  grp = currentGroup();
  comm = communicator(grp);
  s = intermediateScanl1(comm,x);
  m = intermediateScanr1(comm,s);
  b = m - s + x;
  mss = intermediateFold1(comm,b);
  return(mss);
}
```

6.3 Summary

This chapter has presented two case studies that used the PEDL system to produce parallel implementations from mathematical descriptions of an algorithm. The first stage in the process is to express the computation in the PEDL specification language. The specification is then transformed through each decision stage in turn. This incrementally introduces parallelisation detail until a fully specified implementation results.

The second half of the PEDL system was then sketched. This generates a target language implementation from the PEDL program. A process of transformation, simplification and translation is used. This code generation process is described in detail in the next chapter.

It can be seen that the SAC programs that are the result of the derivation process are conventionally structured and readable; no additional runtime system or implementation techniques for functional languages are required.

Furthermore these programs have been constructed by a systematic decision process that allows proofs of correctness where necessary. The SAC programs have been compiled and executed in parallel on a network of workstations with no further modification.

Chapter 7

Code Generation

Capsule

This chapter describes the process of producing a conventional implementation from a PEDL program in which all parallelisation details have been specified. There are three stages to the process.

First the *change of view transformation* is applied to the two-level program. This removes the coordination layer and produces an equivalent processor-view program. This is expressed in the *intermediate language* which has a language model and capabilities similar to the target language. Any communication and scheduling operations that occur within the coordination layer are replaced with equivalent processor-layer requests to these services.

The next stage is to simplify the intermediate program so that it is in a form that can be trivially translated to the target language on a construct-by-construct basis. While some simplification can be automated, much must be done by inspection.

The simplified program can be translated to the target language. Translation rules for some of the language constructs are given. Many of these are obvious, and involve little more than converting syntax. The translation is simplified by the provision of a set of supporting libraries in the target language; we describe the design and implementation of these libraries.

The chapter concludes with an evaluation of the quality of the generated code and the suitability of using SAC as the target language.

Introduction

This chapter presents the back-end of the PEDL system. The previous chapters have shown how parallelization details can be incrementally added to an algorithm specification so that a fully-specified parallel program results. We use a sequence of progressively more detailed and explicit parallel computational models where each model is encapsulated within a language; implementation decisions are added to the program in the course of transforming the program from one language to the next.

The resulting parallel program is expressed in a combination of a coordination and a computation language; both of which are implemented as an embedded language of combinators within Haskell.

The final stage in the process is to take the Haskell-based description of the parallel program and generate a stand-alone implementation from it. This is the focus of this chapter. Unlike the incremental derivation stages, this process is entirely mechanistic. There is no scope for applying insight or 'eureka' steps: although the transformations are currently performed by hand, they can be described by a simple set of rules and much of it could be automated.

We had originally intended to generate a stand-alone implementation expressed in C, using the MPI library for communication. As the computational model and level of abstraction provided by C is far removed from that of the PEDL languages we found that we spent more effort describing the translation of sequential code than focusing on our field of interest, the parallel structure.

We decided to target a language whose abstractions and model were closer to our transformation languages. After a little searching and testing, Single Assignment C (Scholz, 1994) was settled upon. This purely-functional first-order language was chosen for a number of reasons. The translation of sequential code was simplified by the functional semantics and implicit allocation and deallocation of variables provided by SAC. One of the main research directions of SAC is the efficient compilation of functional code. Therefore, we can leave much of the optimisation to the SAC compiler. Another benefit is that the compiler generates standard ANSI C – this means that our code can be executed on parallel machines for which only a C compiler is provided.

Choosing SAC as the target language caused a redesign of the PEDL languages. Powerful features of SAC, such as whole-array combinators, were added as primitives in the PEDL languages. In short, the design was altered to minimise the impedance gap between the two languages. However, we do not believe that the transformation system is inextricably tied to using SAC as a back end; many other languages could be used. Another single-assignment language would be especially well suited.

The Haskell-based description of the parallel program is expressed in a two-level model where the coordination layer provides a collective view of the entire parallel machine. In contrast, SAC has a conventional single-processor computational model. The first step towards producing a SAC implementation is to derive a single-processor view program from the collective-view implementation. This process is described in Section 7.1. The program produced by this transformation is expressed in a small set of constructs, which we refer to as the intermediate language

The next stage is to transform the intermediate language program so that it closer resembles a SAC program. This involves substituting hard-to-implement constructs with more convenient equivalents; expanding and inlining some definitions, flattening nested blocks and rearranging the code. We call this process simplification, and it is described in Section 7.2.

Once it has been simplified, the program is really beginning to resemble a SAC program – it has quite modest features. All that remains is to translate the program. To support this process we implemented a library of SAC code that provides equivalents of some of the intermediate language constructs. This involved adding a binding to MPI and building wrapper libraries which abstracted over the details of the MPI primitives. The design and implementation of the supporting libraries are described in Section 7.3.

The translation proceeds construct by construct. Much of the translation is purely syntactic – we outline the typical features without giving an exhaustive set of rules or algorithm for this. However, translation of some of the constructs is a little more difficult due to the differences between the languages. Section 7.4 presents the interesting parts of the translation process. The following section (Section 7.5) assesses the quality of the SAC code generated and analyzes the weaknesses of this technique.

7.1 Change of View Transformation

The first step in the process of translating to SAC is to rework the program so that it is in a language with a similar computational model to SAC. In particular, SAC along with most other languages provides a single-processor view of execution. This is in contrast to collective view of the entire parallel machine provided by the coordination layer of the PEDL languages. The change of view transformation removes the coordination layer from the program to leave a residual single-processor view program. This program is expressed in a subset of the PEDL constructs; we call it the intermediate stage language.

Since the coordination layer is to be removed, the protection provided by the DVal ADT against improper use of distributed data in this layer is unneeded. Any reference to a distributed value can be replaced by a reference to the element of the distributed value

$$
\begin{array}{ll}
\mathcal{I}[\![\textbf{parallel B}]\!] & \Longrightarrow \textbf{do B} \\
\mathcal{I}[\![\textbf{global B}]\!] & \Longrightarrow \textbf{do B} \\
\mathcal{I}[\![\textbf{use } dv]\!] & \Longrightarrow \textbf{return } dv \\
\mathcal{I}[\![\textbf{communicator } n]\!] & \Longrightarrow \textbf{communicatorRQ } n \\
\mathcal{I}[\![\textbf{globalize E } dv]\!] & \Longrightarrow \textbf{globalizeRQ E } dv \\
\mathcal{I}[\![\textbf{gspmd PS}]\!] & \Longrightarrow \textbf{gspmdRQ PS} \\
\mathcal{I}[\![\textbf{pointToPoint E E' } comm\ dv]\!] & \Longrightarrow \textbf{pointToPointRQ E E' } comm\ dv \\
\mathcal{I}[\![\textbf{broadcast E } comm\ dv]\!] & \Longrightarrow \textbf{broadcastRQ E } comm\ dv \\
\vdots &
\end{array}
$$

Figure 7.1: The change of view transformation

local to that processor. Similarly there is no need to distinguish between computational code that executes in a global or a parallel context. The blocks of computational code embedded in the coordination layer are combined to produce a program for a single processor.

The change of view transformation removes the model of a communication system from within the two-level program: in the processor-view program these services are provided by the underlying runtime system. The coordination layer may contain collective communications which directly permutate the elements of a distributed value. They are replaced by processor-view requests to the external communication system to perform the communication. The requests are similar to the message passing bindings typically found in processor-view languages.

7.1.1 The Transformation

The transformation consists of the rules shown in Figure 7.1, applied exhaustively throughout the program.

- The constructs which introduce blocks of computation into the coordination language (**parallel** and **global**) are replaced with **do**. This preserves the scope of identifiers in the computation block but removes the embedding of one language within another. The program is now composed only of computational blocks, with no coordination layer code.

- As the collective-view layer is being removed, there is no need for distributed values. These values are replaced with the representative element on the processor. This can be simply achieved by replacing calls to **use** with **return** – this introduces a new binding for an existing value, leaving the rest of the code unchanged. These redundant bindings are removed during the following simplification stage.

- The other constructs that appear in the coordination layer are replaced by their intermediate language equivalents. These coordination layer constructs simulate the behaviour of the communication system by manipulating distributed values and other data structures directly. They are substituted by requests to the underlying system to perform the operation.

  In particular, collective view communications are replaced with single-processor view counterparts. These are similar to conventional MPI calls: instead of accepting and producing distributed values, the data communicated is directly passed to and from the procedure. Apart from the change in type of these parameters, the code is unchanged. The figure shows the transformation for two of the communications: the transformation rules for the other communication rules have been elided for conciseness.

- Any type signatures in the program must be adjusted to reflect the change in language and the type of parameters. In particular, any parameters which were previously of type *Dval a* will now have type *a*.

7.1.2 An Example

This following two-level function creates a communicator and uses it to broadcast an element of the distributed value *dv*. This results in another distributed value *dv'* which is subsequently used in a parallel computation *comp*.

```
p    :: Dval Int → Action (RedistributionProg s m) (Dval Int)
p dv = do
         grp  ← global currentGroup
         comm ← communicator grp
         dv'  ← broadcast 1 comm dv
         parallel
           i ← use dv'
           comp i
```

Generating an intermediate language program for this function illustrates all the features of the change-of-view transformation. The following program is the result: it consists of a single computational block which initializes communicators, performs communication and then computes a result.

```
p    :: Int → Action (IntermediateProg s m) Int
p dv = do
         grp  ← do currentGroup
         comm ← communicatorRQ grp
         dv'  ← broadcastRQ 1 comm dv
         do
           i ← return dv'
           comp i
```

The collective operations are replaced with processor-view requests to the communication system. Constructs that embed blocks of computational code are replaced by do blocks. Type signatures are adjusted to reflect the removal of distributed values and the change of language.

7.1.3 Correctness

The change of view transformation $\mathcal{I}$ is the reverse of the transformation $\mathcal{C}$ presented in Section 4.1.9 that defines the semantics of the intermediate language. However, unlike the application of a change-of-view transformation to a simple language that is presented in (O'Donnell, 2000), these transformations are not the inverse of one another. Applying one and then then reverse does not result in the same program. Formally:

$$\forall \mathrm{P} \cdot \mathcal{C}(\mathcal{I}[\![\mathrm{P}]\!]) \neq \mathrm{P}$$
$$\forall \mathrm{P} \cdot \mathcal{I}(\mathcal{C}[\![P]\!]) \neq \mathrm{P}$$

This is because the transformation from collective to individual view ($\mathcal{I}$) results in a loss of structure which cannot be reconstructed by the $\mathcal{C}$ transformation.

If the result program generated by the change-of-view transformation is simply compared to the source, it can be seen that this transformation is not correctness-preserving. It takes a program that operates on distributed values and produces a program that manipulates single values. However, the overall computation performed by a group of processors executing the intermediate program is equivalent to the computation described by the collective-view program. While the program texts are unequal, the behaviour of the systems they describe are equivalent.

The following equation holds (where $\mathcal{S}$ denotes the semantics of the collective-view languages):

$$\forall \mathrm{P} \cdot \mathcal{S}(\mathcal{C}(\mathcal{I}[\![\mathrm{P}]\!])) = \mathcal{S}[\![\mathrm{P}]\!]$$

That is, for any collective program P, the change of view transformation $\mathcal{I}$ preserves the semantics as defined by $\mathcal{C}$. The parallel behaviour of the program is also preserved.

7.2 Simplification

The next stage is to simplify the structure of the intermediate level program so that it is in a form more easily translated to our target language.

This section outlines the steps taken, and the justification for each. Some of the steps are a consequence of using SAC as the target language, while others are more general and

would apply for a range of target languages. We have attempted to devise an ordering that ensures that every step has to be performed at most once. However, it is possible that for some programs later steps could introduce situations that require further application of previous steps.

The result of the simplifications is a program with a flat structure, ordered so that each function is defined at the top level, and before its first usage, as is required by SAC. Partial applications and function-valued parameters are removed while uses of many of the language constructs are inlined.

1. **Rewriting & Expansion**
   The first step is to inline the definitions of some of the constructs and operators of the intermediate language. At the same time, syntactic forms that have no equivalent in SAC are rewritten.

   - The **gspmd** construct, should be replaced by its definition – a conditional statement testing on the rank of the current processor.
   - Case statements and guards are convenient for defining programs in the transformation languages. However, these syntactic forms have no translation to SAC, and so strictly speaking should be disallowed in the transformation languages. At this point they must be rewritten as blocks of nested **if** expressions.
   - The result produced by a sequence of computations (i.e. a **do** block) is the value returned by the last computation in the sequence. However, function definitions in SAC require an explicit `return` statement. An explicit **return** statement is therefore appended to each function definition.

2. **Handle limitations of SAC**
   The next step reforms the program so that it fits within the limitations of SAC.

   - Although higher-order functions are convenient for program development, they are not supported by SAC. Therefore all function-valued parameters must be eliminated. If the value of a function parameter is known at compile-time, it can be substituted into the body of the function. Multiple uses of a higher order function will result in a set of specialized versions of the function.

     If the value of the function parameter cannot be determined at compile-time, a system of enumeration types and conditionals can be used to allow a particular function to be selected from a set at run-time.

- SAC does not allow nested definition of functions. Therefore any local function definitions must either be inlined or moved to the top level. This may require adding additional parameters for values that were previously in scope.
- The current implementation of the SAC compiler requires a compile-time constant value for the bounds of an array generator. If array generators are passed a variable for the bounds, its value must be inlined. This reduces the generality of the program; for instance block sizes cannot be determined at runtime. The code can be made a little more size-independent by using preprocessor definitions to inline the value.

  This is not part of the SAC language specification, but a limitation of the current compiler. Hopefully, a new release of the compiler will remove this constraint.

3. **Flattening and Ordering**
   The program is now in a form where much of the code has been inlined and rewritten in a form that can be translated to SAC. The final step is to flatten the structure of the program – SAC does not permit nested blocks of code.

   - Block structures may only occur in a few places in SAC: as a loop body; as the alternatives in a conditional; and as the body of an array combinator. All other nested blocks must be flattened out, taking care to rename any identifiers that clash with identifiers in the outer scope.
   - In common with C, every function must be defined before it is used. Therefore the function definitions must be sorted into such an order.
   - Finally, we can examine the code and rationalize it by removing dead bindings introduced at earlier stages. Due to the extensive inlining, it may also be possible to factor out repeated computation. This results in a tidier program, but it is unclear if this leads to better code generation by the heavily-optimising SAC compiler.

7.3 SAC Supporting Libraries

A set of libraries were implemented in SAC to support the translation process. Before presenting the translation itself, this section will describe the libraries and some of the issues involved in designing and implementing them. The libraries fall into three parts: the interface and marshaling code required to link to the MPI library; abstractions built upon the MPI primitives; and code to provide the information about the parallel machine environment that is available in the PEDL languages.

7.3.1 Developing an MPI binding

SAC is a relatively new language and does not have a binding to the MPI library. However it has the ability to call C code and has a foreign interface system where APIs can be annotated to indicate the presence of side effects, updates and the desired behaviour of the garbage collector.

Referential transparency is maintained in SAC through uniqueness types (Grelck and Scholz, 1995) similar to those in the language Clean. Side effecting computations operate on objects, which maintain a mutable state. Object types have a uniqueness annotation which allows the type checker to ensure that objects are used in a single-threaded manner throughout the program.

MPI provides an interface of about 140 functions, many of which manipulate data in peculiar ways to maintain a strict separation between the application program and the communication system. Using a combination of classes, objects and code interfaces, we produced a binding to much of the functionality of MPI.

Implementing most of the binding was quite straightforward. However, in some places the strong type system and functional nature of SAC became quite a hindrance. The most noticeable example of this was passing message data to and from communication operations. In the C binding, the locations for the data sent and received from a communication are passed to communications as untyped pointers (`void *`). The size and type of the data is described by further parameters to the MPI call.

However, SAC is a functional language – data is immutable and bound to identifiers rather than stored in variables. Furthermore SAC provides no pointer type, and there is no way to take the address of an identifier. Because of this, it is not possible to pass data directly to MPI communications.

We implemented a buffer abstraction in C, with operations to create new buffers and to copy data to and from them. This abstraction was introduced into SAC as a new class of mutable object. In this way a buffer can be created in a SAC program, data copied into it, and then passed to an MPI communication. This adds the inefficiency of copying data before and after communicating a message, but seems to be the only solution.

A further difficulty is that the shape – the number of elements and dimensions – is part of the type of a SAC array. In the language specification, arrays with unspecified shapes are permitted. However, the current compiler does not support this – arrays passed to and from function calls must have their shape fully specified.

This makes it impossible to provide generic pack and unpack operations for the buffer object. We currently circumvent this by generating dummy pack and unpack functions for every array shape communicated in the program. These all call the same C primitive,

but have differing type annotations. The inconvenience of this is lessened slightly by using macros to declare a new function, instantiated to a particular size.

7.3.2 PEDL Wrapper Functions

Around the MPI primitives, we implemented a set of wrapper functions that more closely resemble the interface of the PEDL communication, group and communicator operations. The wrapper functions for communications create buffers into which the message data is packed. They also construct the data required for the additional parameters of the primitive operation, such as a representation of the message data type.

Due to the size and type system of SAC, these wrappers must also be specialised for each combination of size and type communicated. We use macros, parameterized over the size and type, to define the specialised communication wrapper functions required for a particular application.

The communication operations that are to be used are statically declared using these macros at the start of the program text. This is not a further limitation on the generality of the program: due to the restriction that the compiler must statically know the size of every array generated, this information must already be present at compile time.

Declaring a new wrapper for each size and type introduces some inefficiency, as new buffers and MPI type values are created every time a communication is performed. This could be lessened by expanding the macros and then factoring out common code so that buffers and other data could be reused. However, we hope that an improved version of the SAC compiler which accepts partially-shaped arrays will make this unnecessary.

7.3.3 Runtime Environment

The PEDL system provides a computational model where environmental information about the parallel machine, such as the current partition, is always available. This information is updated whenever the execution context changes. For instance, when the program executes a **gspmd** construct the current partition is sub-divided: the parallel environment information is updated for the duration of the independent computations, and then reverted back to the previous context once the processors unify again in the parent partition.

Within Haskell, this information is propagated throughout the program within the internals of the computation monad. An equivalent system that records the current machine environment must be provided for the SAC translation.

This consists of a few global objects; these are initialized at the start of execution by querying MPI primitives to find information about the current partition and the rank of the particular processor the program is executing on. The definition of the **gspmd** construct updates these objects when entering a new execution context. Primitives such as **size** and **currentGroup** are implemented as projection functions on these objects.

7.4 Translation to the Target Language

The majority of the translation process is purely syntactic – the abstract syntax of the intermediate stage language is very similar to the abstract syntax of SAC. Such details as where to replace ← by = and the syntax for function definitions are elided. Of more interest is the translation of the constructs themselves.

The constructs available in the intermediate language are described in Section 4.1.9 and reproduced here for convenience.

⟨ACTION⟩ ::= ⟨STRUCTURE⟩ | ⟨ARRAYOP⟩
| ⟨COMM⟩ | ⟨GROUP⟩
| ⟨SIZE⟩ | ⟨RANK⟩
| ⟨COMMUNICATOR⟩

The operations of the group, communicator and comm(unication) syntactic classes are translated to calls to wrapper functions around MPI primitives that take the same parameters. Similarly the ⟨SIZE⟩ and ⟨RANK⟩ classes, which access information about the parallel environment, are translated to projection functions over the global objects that maintain the parallel environment. These translations are trivial and are not pursued further. This leaves the ⟨STRUCTURE⟩ and ⟨ARRAYOP⟩ syntactic classes.

7.4.1 Translating Structures

⟨STRUCTURE⟩ ::= **return** ⟨EXP⟩
| **if** ⟨EXP⟩ **then** ⟨BLOCK⟩ **else** ⟨BLOCK⟩
| **do** ⟨BLOCK⟩
| **repeat** ⟨BLOCK⟩
| ⟨*other varieties of loop*⟩ ...

Above are the different forms of structure. The **return** construct can be translated directly. A similar case is **do** – as the block structure of the program has been flattened, this construct can only appear at the start of a function definition, or within the body of a conditional or loop. It allows a sequence of computations to be placed somewhere where

a single computation is expected. As such, it can be translated into the {} braces used for the same purpose in SAC and C.

The conditional is more interesting. While this construct is an expression in the PEDL languages, and so returns a value, it is a statement in SAC. Therefore the translation requires the movement of the binding operation into each alternative block of the conditional, as follows:

$$r \leftarrow \textbf{if}\ p \quad \textbf{then}\ a \quad \textbf{else do}\{\ldots;\ldots;b\} \xrightarrow{T} \texttt{if (p)}\ r = a; \texttt{else}\ \{\ldots;\ldots;r = b;\}$$

This will require complex expressions containing conditionals to be simplified so that the conditionals are directly on the right hand side of a binding.

Loop combinators are translated to the `while` and `for` loops provided by SAC. Although more involved, the transformation of loops is still template-based; it does not depend on the context the loop occurs in, or the operations within the loop body. Extra bindings are introduced for accumulating values and loop counters: these simulate the behaviour of the lambda abstractions of the intermediate program. It must be ensured that these are given fresh names that do not clash with existing bindings.

For example, the **loopnaccum** construct, which iterates a fixed number of times accumulating a result, is translated as follows:

$$r \leftarrow \textbf{loopnaccum}\ n\ a\ (\lambda\ acc \rightarrow \textbf{do}\ \ldots\ v) \xrightarrow{T} \texttt{acc = a; for (index = 0; index < n; index++)\{ ...; acc = v; \} r = acc;}$$

The other loop combinators have similar translations.

7.4.2 Translating Array operations

The other syntactic class that merits examination are the array operations.

⟨ARRAYOP⟩ ::= **genarray** ⟨EXP⟩ ⟨ABS1⟩
| **modarray** ⟨IDENT⟩ ⟨ABS2⟩
| ⟨IDENT⟩ **!** ⟨EXP⟩
| **bounds** ⟨IDENT⟩

The indexing and bounds operations can be translated directly to SAC primitives. The array generation and array map can be expressed as forms of SAC's generic `with` array construct. This allows the specification of a range of indexes to compute over and

provides a limited form of lambda abstraction for the index computation. The array generator operation is translated as follows.

$$
\begin{array}{l}
r \leftarrow \textbf{genarray}\ b\ (\lambda\, ix \rightarrow \textbf{do} \\
\quad x \leftarrow E\ ix \\
\quad \ldots \\
\quad E'\ x)
\end{array}
\quad \overset{T}{\Longrightarrow} \quad
\begin{array}{l}
\texttt{r = with(. <= ix <= .)\{} \\
\quad \texttt{x = E ix;} \\
\quad \ldots \\
\quad \texttt{\} genarray([b], E' x);}
\end{array}
$$

The array map operation is translated in a similar way:

$$
\begin{array}{l}
r \leftarrow \textbf{modarray}\ a\ (\lambda\, (ix, v) \rightarrow \textbf{do} \\
\quad x \leftarrow E\ ix\ v \\
\quad \ldots \\
\quad E'\ x)
\end{array}
\quad \overset{T}{\Longrightarrow} \quad
\begin{array}{l}
\texttt{r = with(. <= ix <= .)\{} \\
\quad \texttt{v = a[ix];} \\
\quad \texttt{x = E ix v;} \\
\quad \ldots \\
\quad \texttt{\} modarray(a, ix, E' x);}
\end{array}
$$

7.5 Discussion

This chapter has presented the back-end of the PEDL system. This takes a two-level program whose parallelisation is fully specified and generates an implementation in the target language.

The first step is to apply the change of view transformation. This produces a processor-view program from the two-level specification. The collective view coordination layer is removed and the coordination code rewritten using processor-view constructs that perform requests to an underlying communication system. This transformation is simply formulated as a set of rules and could be automated.

The processor-view program is expressed in an intermediate language which is similar to the target language. A process of simplification removes constructs that are difficult to translate and reforms the code so that it may be translated construct-by-construct to the target language. Parts of the simplification process are specific to SAC, while others are generally applicable to a range of possible target languages.

The target implementation is then produced by a translation. Most of the translation rules are purely syntactic and trivial. Translating array and loop constructs require a little more: the introduction of new variables. Communication operations are translated to calls to the SAC binding to the MPI library. The production of the final program uses no complex implementation techniques, but instead maps directly to the target language. In particular there are none of the runtime features typically associated with higher-order functional languages, such as graph reduction, garbage collection and heap allocation.

The simplicity of the final translation is due to the PEDL intermediate language being so close to SAC: it presents the same level of abstraction as SAC. As no additional abstractions are imposed by the PEDL computational languages compared to SAC, both systems provide the programmer with the same degree of control over the sequential execution.

This has two benefits. The first is that the generated code is clear, conventionally structured and bears a resemblance to the PEDL sources. If needed, it would be much easier for a programmer to read and modify this compared to, say, the C output of the GHC Haskell compiler. The second benefit is that there is little sequential execution overhead imposed by using the system rather than coding in SAC directly. Furthermore, as SAC has been shown to produce performance comparable to FORTRAN, we can fairly confidently extend this statement and claim that array-based computation in PEDL has little sequential overhead compared to C.

However, the most important factor in determining the performance of a parallel implementation is not the sequential code but the parallelisation and communication code.

The communication constructs of PEDL each map to a single MPI primitive. Therefore the system presents the same level of abstraction as coding against the MPI binding to SAC or C. The programmer has the same degree of control over the machine and can express parallel behaviours in the same level of details in all these languages. This is not the case for more abstract models of parallelism such as skeletons or evaluation strategies.

Although PEDL does not reduce the available control over parallelism, it does restrict the form parallelism may take. It is limited to a strict Group-SPMD model of parallelism; it does not provide access to MPI primitives that do not fit this model. This makes it unsuitable for expressing more dynamic or irregular algorithms. However, for algorithms that can be efficiently expressed in Group-SPMD, we claim that using PEDL imposes little overhead on communication or restrictions on the form of parallelism compared to hand-coding in C+MPI.

The overhead that is incurred is due to excess copying of data into temporary buffers before and after communication. However this is an implementation issue, and is a computational penalty – it does not have a significant impact compared to the time taken for communication itself. Further effort could be expended on producing a more efficient SAC binding to the MPI library that eliminates this copying. However, due to the functional semantics of SAC it may not be possible to remove this copying altogether: although perhaps transformations could be formulated that optimise the intermediate program so that buffers are reused and copying minimised.

The result of a PEDL derivation is an imperative program calling a communication library. We claim that little computational or communication overhead is incurred from using PEDL compared to hand coding. What our system adds to this are levels of ab-

straction from the detail of the parallel algorithm, and from the communication library, at the earlier stages of program design. Additional degrees of control are introduced in a systematic and structured way, while reasoning about the correctness of code is simplified. This does not limit the expressiveness or degree of control available within the later stages of the system, but may permit the programmer to formulate a better solution.

7.6 Further Work

Currently the code generation process is quite simple. Further optimisation steps could be added to produce more efficient target language programs. This optimisation could be assisted by modeling the features of the target language more fully within PEDL. For instance, optimisation of the `with` array loop construct of SAC has been the subject of extensive research. By adding this construct to the PEDL computational language, the programmer could express algorithms in a form that is more likely to be efficiently implemented. If the language was redesigned to be closer to SAC, small irritations such as the difference in array origin could also be removed.

The SAC research group have also investigated the automatic parallelization of `with` loop array operations (Grelck, 1998). Their parallelisation model is a shared-memory task-farm architecture quite different from that in this thesis. However, it would be interesting to experiment with methods to combine the two forms of parallelisation. Possibly PEDL could be used to describe the large-grain problem decomposition, while SAC manages the small-grain system parallelism. This could produce implementations suitable for a distributed machine where each node is a shared-memory multiprocessor.

Another consideration is whether another language would be better suited as the target for PEDL derivations. A lower-level language such as C, `C--` (Peyton Jones et al., 1998) or assembler, would require a much more complex code-generation system. As there are languages that provide a computational model closer to PEDL and produce efficient implementations there seems little point to target a lower level language directly. However, there may be other first-order functional or single-assignment languages which are better suited than SAC.

Chapter 8

Conclusions

8.1 Summary

Parallel programming is a difficult task. An efficient parallel implementation must manage a set of features that are not present in a similar sequential implementation. Many programming models have been proposed to tackle the complexity this introduces. These range from almost totally implicit techniques, where the compiler or runtime system makes most of the parallelisation decisions, to explicitly-parallel models where the programmer has full control.

Implicit approaches remove the need to precisely specify all details of the desired parallel behaviour. This is acceptable when the application is very irregular or where the programmer has little experience of parallel implementation. However for problem domains whose parallelisation is well understood, a skilled practitioner can produce better implementations in a programming model which permits more low-level control.

Unfortunately, explicitly-parallel programs tend to be poorly structured; the management of parallel features is tangled with computational components throughout the program. This leads to code that is difficult to understand, debug and maintain, while the presence of machine specific details reduces the portability of the code.

With effort, the programmer can overcome these drawbacks. A more serious problem is that explicit parallelism makes it more difficult to produce good implementations in the first place. The programming model is so cumbersome that prototyping, reasoning about correctness, and performance prediction is often seen as not being worthwhile.

The thesis proposes a *staged* programming model that accommodates both abstraction and low-level control of parallelism. We claim that such a model possesses many of the benefits of both the implicit and explicit treatments of parallelism. It allows specification,

prototyping and concretization of implementation decisions to take place within a single unified system.

The design of this programming model was guided by the analysis of a group of existing systems that present more than one level of abstraction. It was found that each of these systems had some of the following qualities:

- Implementation decisions are made incrementally.
- The system is divided into a series of discrete design stages.
- There is a fixed number and ordering of these stages.
- Movement from one stage to the next is by some form of program transformation, which may be either formal or informal.
- The system supports reasoning about the computational correctness and parallel behaviour of code.
- Programs at all stages of the system can be executed or simulated.
- The programmer is firmly in control of all implementation decisions: any supporting software is subordinate to the user.
- New primitives that extend the system may be added in a straightforward manner.

However, none of the systems reviewed possesses all these qualities. Many use informal or poorly defined notations; the result of other systems is a specification of the implementation, rather than the implementation itself; while others have no well defined derivation route and so provide less guidance for the programmer.

This thesis introduces a prototype staged programming system called PEDL which possesses all the properties identified above. It comprises a fixed series of stages. Although this is more restrictive than a system with an arbitrary number and order of stages, a fixed series provides more structure for the programmer and simplifies the production of software to support and analyze the process.

Each stage introduces details of another aspect of the parallelisation, A different language is associated with each stage: the languages capture the implementation decisions of that stage at an appropriate level of abstraction. The languages are well defined and executable. This permits equational reasoning on programs and the simulation of intermediate programs.

Once all the parallelisation details have been provided, a series of transformations and translation results in an SPMD program implemented in a conventional imperative language.

8.2 Contributions

8.2.1 The PEDL System

This thesis demonstrates it is possible to produce a programming system containing different levels of abstraction that is based around a series of well-defined executable languages. The system consists of a fixed series of stages which each introduce more implementation detail. At every stage the programmer can focus in isolation on one parallelisation issue.

The concreteness of the stages and languages give a clear implementation route. At each stage the programmer can focus on a facet of the parallelisation in isolation, while other concerns are abstracted by the stage language. This guides the programmer from a specification to a full description of the parallel algorithm. From this a conventional final implementation can be produced by a sequence of semi-automatic transformations.

The thesis also goes some way towards showing that a concrete staged programming system is also productive. The PEDL prototype system is limited to a particular programming model and has a limited range of datatypes and computational features. This restricts it to a particular class of algorithm: mostly data-parallel array based computations. However, the small case studies presented in this thesis demonstrate that algorithms which fit within these constraints can be clearly and concisely expressed and that their final implementations have little overhead compared to hand-coded equivalents.

In the initial stages of the system many of the implementation details are abstracted away from: programs in these stage languages are compact and easy to understand. This makes experimentation, modeling and reasoning about parallel performance much more feasible. The incremental introduction of detail structures the implementation process, while the well defined derivation route allows libraries of commonly used abstractions, equalities and transformations to be built up and reused. So it appears that this staged programming system is effective in combining the benefits of high-level specification and low level implementation within the same system.

Our only reservation is over the amount of code manipulation and rewriting necessary during a program derivation. Each stage of the process requires the transformation of the program from one language to the next. As the languages share a common base of constructs much of the code remains unchanged. Nonetheless, while transformation by hand is manageable for the small case studies presented here, clearly when working with larger real-world applications it could become unwieldy. The PEDL system still needs to be tested against larger case studies to ascertain the significance of this problem.

A usable staged programming system for real-world applications would have to be sup-

ported by a suite of tools for program editing, analysis and transformation. The concrete and constrained nature of the staged system would simplify the design and production of this suite. This further work is discussed in Section 8.3.3.

A related issue is the guidance provided to the programmer during the derivation. Cost models for performance prediction are increasingly being seen as a necessary component of a useful parallel programming system. The metrics they provide allow the programmer to compare candidate implementation choices.

Cost models would be straightforward to add to the PEDL system. One possible technique would be the cost modelling system proposed for the APM methodology. As the operational behaviour of PEDL parallel constructs are already described in the APM methodology this would pose few problems. This cost modeling system permits costs to be derived for ParOps of internal nodes of the APM hierarchy from the known costs of ParOps of leaf node APMs. In the case of PEDL, this would mean that costs for constructs of the stage languages could be derived from experimental measurement of the corresponding target language constructs.

Another possibility is to use a cost model similar to that of TwoL. TwoL and PEDL share the same model of parallel computation – Group-SPMD. The earlier stage programs correspond to partially-specified TwoL frame programs. In these cases a cost function is derived which is parameterised over the unspecified implementation decisions.

8.2.2 Staged Programming Models

The PEDL system presented in this thesis is only a first prototype of a concrete staged programming system. One question that this prototype raises is the optimal size and number of stages in the methodology. PEDL has a few large stages, each of which produces a significant change to the level of parallel implementation detail. This means that significant leaps are made during a derivation – although this is mitigated somewhat by being able to execute programs that are expressed in a mixture of languages.

It may be better to have a system with a finer granularity: one with a longer series of stages where each stage introduces a smaller amount of implementation decisions. This would reduce the distance between steps and make each more manageable. However, it is unclear whether it is possible to decompose the parallelisation process into more stages.

The staged programming model may have applications outside the field of parallelism. Other complex problem domains could benefit from the multiple levels of abstraction that this model provides. The ability to make significant decisions first and later fix the smaller details may be productive in, for example, hardware design or distributed systems.

For the staged programming model to be appropriate for a particular problem domain, it must be possible to decompose the domain into a set of implementation concerns. Each of the concerns must be to some extent independent, so that each can be tackled in isolation. Furthermore, it must be possible to order the concerns in such a way that freezing the implementation details of one concern does not overly constrain the choice for later concerns. If freezing one concern constrains another concern so much that useful solutions are not possible it suggests that it may be better to reverse the order in which decisions are made on the two concerns. It is possible that in some problem domains there will be cyclic dependencies between concerns. Such loops could be tackled by backtracking through stages, but a large amount of circularity would lead to a system where no progress was made until the entire solution was found.

Parallel implementation is amenable to the staged programming system, at least for the Group-SPMD model, because it is possible to serialize the fixing of concerns without requiring any backtracking. Freezing one concern restricts the possibilities for later concerns, but not so much as to prevent useful solutions. For instance, in PEDL decisions are made on the redistribution of data before choosing collective communication operations that satisfy these requirements. Therefore redistribution, the earlier concern, constrains the possible communication patterns chosen as a later concern. This is not a problem because in this case the latter concern is subordinate to the redistribution requirements.

8.2.3 Implementing Embedded Languages

The PEDL languages are implemented by embedding them within a pure lazy functional language. This simplifies the design of the languages, provides a common semantic base for reasoning and translation, and makes the implementation and modification of the languages easier. It also serves as a case study for the benefits and practicality of this implementation technique.

An unexpected contribution was the development of techniques to represent the static semantics of stage languages and the legal combinations of languages within the type system of the host language. This enables us to use the type checker of the host language to verify that programs are well-formed. Previous projects have used phantom classes and types. However this work extends their application to the use of trees of phantom types to represent languages and the legal contexts for constructs. Phantom types are also used to record whether a block of code performs side-effecting computations and whether it is composed of constructs from more than one language. This indicates to the programmer when a block has been fully transformed to the next stage of the derivation, and when extra care should be taken in applying transformations.

Chains of phantom types could be used in other situations to differentiate between related languages or other forms of hierarchy. For example, (Leijen and Meijer, 2000) describes a set of Haskell combinators for expressing SQL queries which can then be passed to an external database server. Many database vendors use different dialects or extensions of standard SQL. Therefore a query accepted by one server may be considered malformed by another. Phantom language types could ensure that each server is only passed queries in its own dialect, while allowing standard SQL constructs to be passed to all servers. The dialect and sub-dialects of SQL to be modeled can be arranged into a hierarchy, which is represented by a tree of phantom types similar to that used in PEDL. Each SQL combinator is parameterised by an additional phantom language type variable. This is left fully polymorphic in the type signature of standard SQL constructs; server-specific constructs constrain the language type to the extent in the hierarchy in which the construct is legal. Finally the call to the SQL interface is passed two values: the SQL expression to execute and a value of the phantom language type that represents the language that particular server accepts. By requiring that these two types be unified the type-checker can detect any incorrect use of dialects.

8.3 Further Work

8.3.1 Extending the Parallelism Model

PEDL enforces a strict Group-SPMD model of parallelism. While this makes programs amenable to analysis and is suitable for a wide class of applications, not all algorithms can be efficiently expressed in this model. Highly irregular computations are better suited to a more dynamic programming model, where the parallel behaviour of the program adapts to the input data.

A further development of the PEDL system would be to generalize it to other forms of parallelism – in particular task parallelism with unstructured point-to-point communications. This would require a redesign of the languages and would weaken the reasoning ability. However with care it should be possible to design language constructs and extensions to the system so that the the strengths of the Group-SPMD model are retained for algorithms that fit within it, while still allowing more difficult algorithms to be expressed.

Another thing to investigate is whether the non-blocking communications provided by MPI can be accommodated in our model. These are useful in achieving higher performance but again would complicate reasoning about programs.

Designing extensions to the system will be made easier by the nature of the implementation: the embedding technique simplifies experimenting and refining new language constructs. Furthermore, phantom language types can be used to record more information about the behaviour of a block of code – for instance the presence of asynchronous parallelism or non-blocking communications that require greater care in transformation.

8.3.2 Developing the System

There are a number of developments and extensions that could be made to the current system to improve its ability to handle real-world problems.

In PEDL it is possible to reason about programs expressed in the same language and in adjacent languages. Currently this reasoning is done by appealing to the underlying semantics. A developed staged programming system should provide a library of equalities and theorems. This would allow the programmer to relate and reason about constructs directly, instead of working from first principles. Producing a library of such theorems would allow concise high-level proofs to be produced.

Similarly, the system should supply catalogues of transformations that describe the usual implementation routes for common patterns of parallelism. Candidates for such a

catalogue would include transformations that introduce block-wise or cyclic data distributions for independent computations.

Another improvement would be to enrich the computational language with other datatypes, such as trees and lists. This would not affect the coordination layer of the system, but would provide more freedom to express computational components. Adding a new data type and operations over it would require an implementation as combinators in the PEDL computational language, the addition to the semantics of a characterisation of its behaviour, and rules to describe the translation of these combinators to supporting libraries in the target language.

The PEDL system currently uses only a small range of the facilities provided by MPI. The languages could provide bindings to some of the more powerful features of this library, such as processor topologies. Upon the processor topologies could then be added further abstractions, for example data distribution types (Rauber and Rünger, 1995) and distributed arrays. This would allow parallel array computations to be expressed more easily, without the need to map between global and local array indices. This would require research into the design of an interface for this abstraction that was convenient and still gave the programmer sufficient control.

A weakness of the design of the PEDL collective communications is the point-to-point operation. This causes a synchronisation across all processors in the current partition. Therefore a sequence of these communications is very inefficient. A better formulation would be a BSP-like primitive (Valiant, 1990), in which a set of arbitrary point-to-point communications are performed simultaneously, followed by a synchronisation across the partition.

Adding new communication primitives to the language is straightforward, but could be simplified by a more generic treatment of communication within the operational and parallel semantics. Instead of the full definition currently given for each operation, a general definition could be provided that is then parameterised by a permutation function which defines the characteristics of a particular communication. Hence the semantics of new primitives could be specified with the definition of a new permutation function.

This research could be extended to the design of libraries of higher-level abstractions that capture frequently used parallelisation idioms. Although similar to skeletons, these idioms would not be black-box constructs whose implementation details were hidden. They would be implemented using the primitives provided by the stage languages. Each encapsulation of a parallel idiom would be accompanied by a set of theorems and transformations that describe its properties, equivalences and possible implementation routes. An idiom would have one or possibly more realizations for each stage of the system. For instance, reducing the elements of a distributed value in the independent language could

be encapsulated as an idiom function. At the next stage it could be transformed to performing a reduction over various different distributions of data. Each alternative would be implemented as a separate library function in the distribution stage languages.

These idiom libraries would allow a much more structured form of parallel derivation without sacrificing any control. Alternate realizations of the idiom can be chosen at each stage; while if further control is required their definition in terms of the language primitives can be manipulated directly.

8.3.3 Tool Support

It can be seen that the back end of the PEDL system is mechanical and that much of it could be automated. Of more interest is the design of software to support the programmer during the earlier stages of the process by providing guidance and performing transformations. These decision stages require insight and inspiration, so any tool must be directed by the programmer rather than be fully automated.

In its minimal form, the support software would be an interactive editor that applied transformations to the program text. It should verify at each step that the transformation being applied was valid, and record any side-conditions or assumptions made. However, there are much richer design possibilities for a truly useful, powerful tool suite. In addition to the core transformation system, the following features would be desirable.

- **Journaling.** The application should maintain a record of the transformations applied to the program. This acts as documentation and justification for the implementation, It also allows the programmer to roll back implementation decisions when: seeking better solutions to the problem; porting a program to a different architecture; or reworking a component for a different usage or data distribution.

- **Program Analysis.** If the application could perform program analyses (for example cost analysis) the results could be used to narrow the set of applicable transformations, or to rank them according to an estimation of the likelihood of good results.

  One way to provide such functionality would be to use multiple interpretations of the PEDL language combinators. This technique is used in the hardware description language Hydra (O'Donnell, 1987) to perform profiling, proof checking and symbolic evaluation. The technique works by providing a set of different implementations for the language combinators: evaluating the source program with respect to a different implementation of the underlying language produces a different kind of result.

- **Scripting.** A powerful feature would be to allow the programmer to record a sequence of transformation applications, so that it can be reused at other points in the derivation. This would require a form of macro or transformational language in which the programmer defines new transformations by composition. Such a language should also be able to express higher-order operations over transformations, for instance: applying a parameter transformation exhaustively over a program text; or applying a transformation selectively according to a condition.

  The provision of a transformation control language, combined with the ability to perform program analysis, could lead to the development of a tactics-based transformation system. Here the user chooses from a selection of high-level parallelisation tactics. Once a tactic is chosen, the low-level details of the transformation are performed automatically, after which the result is presented to the programmer for evaluation.

- **User Interface.**

  Such an inherently complex application would require research into the design of a productive user interface. The user will be working on a sizeable program text, possibly divided into many different versions and branches. They must choose transformations from a large catalogue and then direct where they are to be applied to the code. Performing the transformation adds to the derivation history and generates proof requirements for any side conditions that occur. The interface must be able to shield the user from the complexity of the system by eliding any details that were not currently important.

  As well as managing the complexity, the interface must also support rather than hinder the user. Many proof assistants and structured editors have a modal method of operation, where the task to be performed is strictly serialized and formalized. This can stifle inspiration and frustrate the user. Even though the application is primarily a verification system that checks the correctness of transformations, it must allow the programmer to go with a hunch and temporarily break the rules: for example apply transformations to any portion of the code, leave side conditions unsatisfied, make unproven assumptions, or directly manipulate the code. The application should record the conditions and assumptions made, so if an exploratory transformation appears promising the programmer can return to tie up the loose ends by formalising the informal transformation steps they have performed.

  A related feature is that the tool should allow the development tree to be forked at arbitrary points of the derivation. Often the programmer will not know which of some alternatives is the best way to proceed. It would be convenient to be able to develop a few alternatives in parallel for a while, until it become clear which is the

best approach. Furthermore, this forking should only scope over the areas where the sibling derivations differ; the tool should allow unaffected transformations to be mapped across all sibling programs.

There are some existing systems that could provide inspiration or a starting point for the proposed tool suite. (Tullsen and Hudak, 1998) describes work in progress on the design of a *Programmer Assistant for Transforming Haskell* (PATH). This provides a meta-language in which to express the application of *transformation templates* (Huet and Lang, 1978) to an object program. Subsets of Haskell are used for both the meta-language and object-language. The design of the meta-language and supporting libraries ensure that only correctness-preserving transformations can be performed.

The *Haskell Equational Reasoning Assistant* (HERA) is an ongoing project in Glasgow to develop a system that assists in the production of inductive proofs over Haskell programs. It provides a point-and-click interface to select theorems and expression points at which they are to be applied. The applicability of the chosen theorem is checked, and then the expression rewritten as another line in the proof.

Meanwhile there are many examples of tools that perform automated transformation, for example the Glasgow Haskell Compiler (Peyton Jones, 1996), which could inform the design of an implementation of the back-end of the PEDL system.

This thesis has shown that a concrete staged programming method effectively manages complexity while still permitting detailed specification of the parallel behaviour. By relaxing the computational model of the system and adding libraries of higher level abstractions and corresponding sets of theorems, the programmer will be able to concisely express a much broader class of parallel algorithm. With automation of the back end of the system and tool support for program analysis and transformation, a staged programming system would be truly useful for real-world applications.

Appendix A

Sources for the Case Studies

A.1 Wave Equation

A.1.1 PEDL Codes

Specification Stage

```
seqHarness :: SpecificationProg m ()
seqHarness = do
        sz ← return (20 :: Int)
        iterations ← return (20 :: Int)
        boundsLeft ← return (100 :: Float)
        boundsRight ← return (0 :: Float)
        vect ← genArr sz (\_ → return (50 :: Float))
        let f l c r = return ((l/2 + c + r/2)/2)
        r ← seqWaveEq f iterations boundsLeft boundsRight vect
        putOut "result"
        printOut r
        return ()


seqWaveEq op n bl br v =  do
        b ← bounds v
        final $ loopNAccum n v (\vect →
                oneSeqIter op bl br b vect )


oneSeqIter op bl br b vect = genArr b (\i → do
        l ← if i >1 then vect ! (i−1)
                    else return bl
        c ← vect ! i
        r ← if i < b then vect ! (i+1)
                     else return br
        op l c r
        )
```

Independent Stage

```
indepHarness :: IndependentProg m (Dval ())
indepHarness = do
        sz ← global size
        iterations ← return (20 :: Int)
        boundsLeft ← return (100 :: Float)
        boundsRight ← return (0 :: Float)
        ivect ← parallel (return (50 :: Float))
        let f l c r = return ((l/2 + c + r/2)/2)
        r ← indepWaveEq f iterations boundsLeft boundsRight ivect
        one $ do putOut "result"
                 nulLoop $ for (1::Int, (≤sz), (+1))
                         $ \i → do v ← r 'get' i
                                   printOut v

indepWaveEq op n bl br iv =  do
        b ← global size
        final $ loopNAccum n iv (\ivect → parallel $
                do  i ← rank
                l ← if i >1 then ivect 'get' (i−1)
                             else return bl
                c ← use ivect
                r ← if i < b then ivect 'get' (i+1)
                             else return br
                op l c r
                )
```

Distributed Stage

```
distribHarness :: DistributedProg m (Dval ())
distribHarness = do
      sz ← global size
      blocksize ← return (20 'div' sz)
      iterations ← return (20 :: Int)
      boundsLeft ← return (100 :: Float)
      boundsRight ← return (0 :: Float)
      bvect ← parallel (genArr blocksize (\_ → return (50 :: Float)))
      let f l c r = return ((l/2 + c + r/2)/2)
      r ← distribWaveEq f iterations boundsLeft boundsRight blocksize bvect
      on 1 $ do putOut "result"
                nulLoop $ for (1::Int, (≤sz), (+1))
                        $ \i → do v ← r 'get' i
                                  printOut v

distribWaveEq op n bl br block bv =  do
      b ← global size
      final $ loopNAccum n bv (\bvect → parallel $ do
              i ← rank
              l ← if i > 1 then do a ← bvect 'get' (i−1)
                                   a ! block
                           else return bl
              c ← use bvect
              r ← if i < b then do a ← bvect 'get' (i+1)
                                   a ! 1
                           else return br
              oneSeqIter op l r block c
              )
```

Redistribution Stage

```
redistHarness :: RedistributionProg m (Dval ())
redistHarness = do
        sz ← global size
        blocksize ← return (20 `div` sz)
        iterations ← return (20 :: Int)
        boundsLeft ← return (100 :: Float)
        boundsRight ← return (0 :: Float)
        bvect ← parallel (genArr blocksize (\_ → return (50 :: Float)))
        let f l c r = return ((l/2 + c + r/2)/2)
        r ← redistWaveEq f iterations boundsLeft boundsRight blocksize bvect
        grp ← global currentGroup
        comm ← communicator grp
        r' ← gather 1 comm r
        on 1 $ do putOut "result"
                  vect ← use r'
                  nulLoop $ for (1,(≤sz),(+1))
                          $ \i → do v ← vect ! i
                                    printOut v

redistWaveEq op n bl br block bv = do
        b ← global size
        grp ← global currentGroup
        comm ← communicator grp
        final $ loopNAccum n bv (\bvect → do
                ldata ← parallel $ do a ← use bvect
                                      a ! block
                rdata ← parallel $ do a ← use bvect
                                      a ! 1
                ldata' ← shift NonPeriodic Inc 1 comm ldata
                rdata' ← shift NonPeriodic Dec 1 comm rdata
                parallel $ do
                        i ← rank
                        l ← if i >1 then use ldata'
                                    else return bl
                        c ← use bvect
                        r ← if i < b then use rdata'
                                     else return br
                        oneSeqIter op l r block c
                        )
```

Intermediate Stage

```
singleHarness :: IntermediateProg m ()
singleHarness = do
        sz ← size
        blocksize ← return (20 `div` sz)
        iterations ← return (20 :: Int)
        boundsLeft ← return (100 :: Float)
        boundsRight ← return (0 :: Float)
        bvect ← genArr blocksize (\_ → return (50 :: Float))
        let f l c r = return ((l/2 + c + r/2)/2)
        r ← singleWaveEq f iterations boundsLeft boundsRight blocksize bvect
        grp ← currentGroup
        comm ← communicatorRQ grp
```

```
        r' ← gatherRQ 1 comm r
        sOn 1 $ do putOut "result"
                  vect ← return r'
                  nulLoop $ for (1,(≤sz),(+1))
                          $ \i → do v ← vect ! i
                                    printOut v

singleWaveEq op n bl br block bv =  do
        b ← size
        grp ← currentGroup
        comm ← communicatorRQ grp
        final $ loopNAccum n bv (\bvect → do
                ldata ← do a ← return bvect
                           a ! block
                rdata ← do a ← return bvect
                           a ! 1
                ldata' ← shiftRQ NonPeriodic Inc 1 comm ldata
                rdata' ← shiftRQ NonPeriodic Dec 1 comm rdata
                do
                        i ← rank
                        l ← if i >1 then return ldata'
                                    else return bl
                        c ← return bvect
                        r ← if i < b then return rdata'
                                     else return br
                        oneSeqIter op l r block c
                        )
```

A.1.2 Simplified Version

```
cBLOCK = 4 :: Int

f l c r = return ((l/2 + c + r/2)/2)

oneSeqIter bl br vect = do
        result ← genArr cBLOCK (\i → do
                l ← if i >1 then vect ! (i−1)
                            else return bl
                c ← vect ! i
                r ← if i < cBLOCK then vect ! (i+1)
                                  else return br
                element ← f l c r
                return element
                )
        return result

singleWaveEq n bl br  bv =  do
        b ← size
        grp ← currentGroup
        comm ← communicatorRQ grp
        vect ← final $ loopNAccum n bv (\bvect → do
                ldata ← bvect ! cBLOCK
```

```
            rdata ← bvect ! 1
            ldata' ← shiftRQ NonPeriodic Inc 1 comm ldata
            rdata' ← shiftRQ NonPeriodic Dec 1 comm rdata
            i ← rank
            l ← if i >1 then return ldata'
                        else return bl
            r ← if i < b then return rdata'
                         else return br
            result ← oneSeqIter l r bvect
            return result
            )
      return vect

singleHarness :: SingleProg ()
singleHarness = do
      sz ← size
      iterations ← return (20 :: Int)
      boundsLeft ← return (100 :: Float)
      boundsRight ← return (0 :: Float)
      bvect ← genArr cBLOCK (\_ → return (50 :: Float))
      r ← singleWaveEq iterations boundsLeft boundsRight bvect
      grp ← currentGroup
      comm ← communicatorRQ grp
      r' ← gatherRQ 1 comm r
      sOn 1 $ do putOut "result"
                 nulLoop $ for (1,(≤sz),(+1))
                         $ \i → do v ← r' ! i
                                   printOut v
```

A.1.3 SAC Code

```
/* SAC+MPI WaveEquation program */
import StdIO : all;
import Hp: all;
import Array : all;

#include "hp.h"
#define BLOCK 4
#define RESULT 20

/* declare the communications used */
DEFINE_Gather_Vect(float,5,4,20)
DEFINE_Shift(float)


/* program proper begins*/
inline float f(float l, float c, float r){
    return ((l/2.0f + c + r/2.0f)/2.0f);
}

float[] oneSeqIter(float bl,float br, float[] vect){
  result = with ( . <= i <= . ) {
    if (i[[0]] > 0) {l = vect[i-1];} else {l = bl;}
```

```
    c = vect[i];
    if (i[[0]] < BLOCK - 1) {r = vect[i+1];} else {r = br;}
    element = f(l,c,r);
  } genarray([BLOCK],element);

  return (result);
}

float[] singleWaveEq(int n, float bl, float br, float[] bv){
  b = size();
  comm = currentComm();
  loopcount = n;
  bvect = bv;
  while (loopcount > 0){
    ldata = bvect[[BLOCK-1]];
    rdata = bvect[[0]]; /* bounds fiddling here */
    ldataA = Shift_float(false,1,1,comm,ldata);
    rdataA = Shift_float(false,-1,1,comm,rdata);
    i = rank();
    if (i > 1) {l = ldataA;} else {l = bl;}
    if (i < b) {r = rdataA;} else {r = br;}
    result = oneSeqIter(l,r,bvect);
    bvect = result;
    loopcount--;
  }
  vect = bvect;
  return (vect);
}

void seqHarness(){
  sz = size();
  iterations = 20;
  boundsLeft = 100.0f;
  boundsRight = 0.0f;
  bvect = with ( . <= ix <= . ){
  } genarray([BLOCK], 50.0f);
  r = singleWaveEq(iterations,boundsLeft,boundsRight,bvect);
  grp = currentGroup();
  comm = communicator(grp);

  rA = Gather_Vect_float_5_4(1,comm,r);

  if (rank() == 1) {
    print("result");
    print(rA);
  }
}

int main(){
  seqHarness();
  Mpi_Finalize();
  return(0);
}
```

A.1.4 Runlogs

```
WaveEquation> main
Specification version
result
{1 := 87.7614, 2 := 76.6355, 3 := 67.4444, 4 := 60.5512,
5 := 55.8637, 6 := 52.9792, 7 := 51.3764, 8 := 50.5741,
9 := 50.2106, 10 := 50.051, 11 := 49.949, 12 := 49.7894,
13 := 49.4259, 14 := 48.6236, 15 := 47.0208, 16 := 44.1362,
17 := 39.4488, 18 := 32.5556, 19 := 23.3645, 20 := 12.2386, }

RunLog of Execution
===================
Single Language : True
Language : Sequential  No. 1.0
Blocks in :

.
.
.

Blockwise version
result
{1 := 87.7614, 2 := 76.6355, 3 := 67.4444, 4 := 60.5512, }
{1 := 55.8637, 2 := 52.9792, 3 := 51.3764, 4 := 50.5741, }
{1 := 50.2106, 2 := 50.051, 3 := 49.949, 4 := 49.7894, }
{1 := 49.4259, 2 := 48.6236, 3 := 47.0208, 4 := 44.1362, }
{1 := 39.4488, 2 := 32.5556, 3 := 23.3645, 4 := 12.2386, }

RunLog of Execution
===================
Single Language : True
Language : Processor Limited Parallel Machine  No. 3.0
Blocks in :


% mpirun -np 5 WaveEquation

resultDimension:  2
Shape     : <  5,  4>
|87.761436 76.635468 67.444443 60.551178 |
|55.863747 52.979153 51.376411 50.574120 |
|50.210587 50.050983 49.949020 49.789413 |
|49.425880 48.623596 47.020847 44.136246 |
|39.448818 32.555553 23.364536 12.238565 |
```

A.2 Maximum Segment Sum

A.2.1 PEDL Codes

Vanilla Haskell Specification

```
mss :: [Int] → Int
mss x = let
        s = scanl1 (+) x
        m = scanr1 (max) s
        b = zipWith3 (\mi si xi → mi − si + xi) m s x
        mss = foldr1 max b
        in mss
```

PEDL Specification

```
specMss :: Vector Int → SpecificationProg m Int
specMss x = do
        s ← scanlArr1 sumOp x
        m ← scanrArr1 maxOp s
        b ← mod3Arr (\mi si xi → return (mi − si + xi))  m s x
        mss ← foldArr1 maxOp b
        return mss

specHarness :: SpecificationProg m ()
specHarness = do
        a ← genArr 6 (\i → case i of
                1 → return (2::Int)
                2 → return (− 4)
                3 → return 2
                4 → return (− 1)
                5 → return 6
                6 → return (− 3)
                ) -- PEDL needs syntax for  array declaration.
        printOut a
        r ← specMss a
        printOut r
```

Independent Stage

```
independentHarness :: IndependentProg m (Dval ())
independentHarness = do
        a ← parallel $ do
                i ← rank
                case i of
                        1 → return (2::Int)
                        2 → return (− 4)
                        3 → return 2
                        4 → return (− 1)
                        5 → return 6
                        6 → return (− 3)
        one $ nulLoop . for (1,(≤6),succ) $ \i → do
                v ← a 'get' i
                printOut v
```

```
        r ← independentMss a
        one $ do v ← r 'get' 1
                 printOut v

independentMss :: Dval Int → IndependentProg m (Dval Int)
independentMss x = do
        s ← scanlDval1 sumOp x
        m ← scanrDval1 maxOp s
        b ← parallel $ do
                 mi ← use m
                 si ← use s
                 xi ← use x
                 return (mi − si + xi)
        mss ← foldDval1 maxOp b
        return mss

scanlDval1,scanrDval1,foldDval1 :: (Storable a,NotRedistribution l) ⇒
        (a → a → Action (l (Computation (Processor Parallel)) s m) a) →
        Dval a → Action (l Coordination s m) (Dval a)

scanlDval1 op dv = do
        n ← global size
        rs ← parallel rank
        final . forAccum (0,(≤ ceiling (log (fromInt n))),succ) dv $
          \ j b → parallel $ do
                        i ← use rs
                        bi ← use b
                        if (i > 2^j) then do
                                xi ← b 'get' (i−2^j)
                                xi 'op' bi
                          else
                                return bi
scanrDval1 = ...
foldDval1 = ...
```

Redistribution Stage

```
redistHarness :: RedistributionProg m (Dval ())
redistHarness = do
        a ← parallel $ do
                 i ← rank
                 case i of
                          1 → return (2::Int)
                          2 → return (− 4)
                          3 → return 2
                          4 → return (− 1)
                          5 → return 6
                          6 → return (− 3)
        grp ← global currentGroup
        comm ← communicator grp
        a' ← gather 1 comm a
        on 1 $ do vect ← use a'
                  printOut vect
        r ← redistMss a
        on 1 $ do v ← use r
                  printOut v
```

```
redistMss :: Dval Int → RedistributionProg m (Dval Int)
redistMss x = do
        grp ← global currentGroup
        comm ← communicator grp
        s ← redistScanlDval1 sumOp comm x
        m ← redistScanrDval1 maxOp comm s
        b ← parallel $ do
                mi ← use m
                si ← use s
                xi ← use x
                return (mi − si + xi)
        mss ← redistFoldDval1 maxOp comm b
        return mss


redistScanlDval1, redistScanrDval1, redistFoldDval1
        :: (Storable a,Sendable a) ⇒
 (a → a → Action (Redistribution (Computation (Processor Parallel)) s m) a)
        → Communicator → Dval a
        → Action (Redistribution Coordination s m) (Dval a)

redistScanlDval1 op comm dv = do
        n ← global size
        rs ← parallel rank
        final . forAccum (0,(≤ ceiling (log (fromInt n))),succ) dv$
         \j b → do
              x ← shift NonPeriodic Inc (2^j) comm b
              parallel $ do
                    i ← use rs
                    bi ← use b
                    if (i > 2^j) then do
                          xi ← use x
                          xi `op` bi
                     else
                          return bi


redistScanrDval1  = ...
redistFoldDval1   = ...
```

Intermediate Stage

```
intermediateHarness :: IntermediateProg s (())
intermediateHarness = do
        a ← do i ← rank
               case i of
                     1 → return (2::Int)
                     2 → return (− 4)
                     3 → return 2
                     4 → return (− 1)
                     5 → return 6
                     6 → return (− 3)
        grr ← currentGroup
        comm ← communicatorRQ grp
        a' ← gatherRQ 1 comm a
        sOn 1 $ do printOut a'
        r ← intermediateMss a
        sOn 1 $ do printOut r
```

```
intermediateMss :: Int → IntermediateProg m Int
intermediateMss x = do
                grp ← currentGroup
                comm ← communicatorRQ comm
                s ← intermediateScanl1 sumOp comm x
                m ← intermediateScanr1 maxOp comm s
                b ← return (m − s + x)
                mss ← intermediateFold1 maxOp comm b
                return mss


intermediateScanl1, intermediateScanr1, intermediateFold1
        :: (Storable a,Sendable a) ⇒
 (a → a → Action (Intermediate (Computation (Processor ())) s m) a)
        → Communicator → a
        → Action (Intermediate (Computation (Processor ())) s m) a

intermediateScanl1 op comm dv = do
        n ← size
        i ← rank
        final . forAccum (0,(≤ ceiling (log (fromInt n))), succ) dv $
         \j bi → do
              xi ← shiftRQ NonPeriodic Inc (2^j) comm bi
              if (i > 2^j) then
                   xi `op` bi
              else
                   return bi


intermediateScanr1 = ...
intermediateFold1 = ...
```

A.2.2 Simplified Version

```
intermediateScanl1 comm dv = do
        n ← size
        i ← rank
        result ←final . forAccum (0,(≤ ceiling (log (fromInt n))), succ) dv $
         \j bi → do
              xi ← shiftRQ NonPeriodic Inc (2^j) comm bi
              if (i > 2^j) then
                   return (xi + bi)
              else
                   return bi
        return result


intermediateScanr1 comm dv = ...
intermediateFold1 comm dv = ...


intermediateMss :: Int → IntermediateProg m Int
intermediateMss x = do
        grp ← currentGroup
        comm ← communicatorRQ grp
```

```
        s ← intermediateScanl1    comm x
        m ← intermediateScanr1    comm s
        b ← return (m − s + x)
        mss ← intermediateFold1 comm b
        return mss

intermediateHarness :: IntermediateProg s (())
intermediateHarness = do
        i ← rank
        a ← if (i==1) then
                  return (2::Int)
                else if (i == 2) then
                  return (− 4)
                else if (i==3) then
                  return 2
                else if (i==4) then
                  return (− 1)
                else if (i==5) then
                  return 6
                else {-if (i==6) then-}
                  return (− 3)
        grp ← curentGroup
        comm ← communicatorRQ grp
        a' ← gatherRQ 1 comm a
        if i == 1 then printOut a'
                  else return undefined
        if i == 1 then putOut "----"
                  else return undefined
        r ← intermediateMss a
        if i == 1 then printOut r
                  else return undefined
        return ()

main = do runIntermediate 6 intermediateHarness
          return ()
```

A.2.3 SAC Code

```
/* SAC+MPI Mss Program */

/* standard headers */
import StdIO : all;
import Hp: all;
import Math : all;
#include "hp.h"

/* declare the comms we use */
DEFINE_Gather(int,6)
DEFINE_Shift(int)

/* integer versions of the math functions */
int pow(int i,int j){
  return (toi(pow(tof(i),tof(j))));
```

```
}

float log(int i){
  return (log(tof(i)));
}

int ceiling(float f){
  return(toi(ceil(f)));
}

/* The program proper */
int intermediateScanl1(Mpi_Comm &comm, int dv){
  n = size();
  i = rank();

  bi = dv;
  for (j=0;j < ceiling(log(n)); j++){
    xi = Shift_int(false,1,pow(2,j),comm,bi);
    if (i > pow(2,j))
      {bi = xi + bi;}
    else
      {bi = bi;}
  }
  result = bi;
  return(result);
}

int intermediateScanr1(Mpi_Comm &comm, int dv){
  n = size();
  i = rank();

  bi = dv;
  for (j=0; j < ceiling(log(n)); j++){
    xi = Shift_int(false,-1,pow(2,j),comm,bi);
    if (i < (n+1) - pow(2,j))
      {bi = max(xi,bi);}
    else
      {bi = bi;}
  }
  result = bi;
  return(result);
}

int intermediateFold1(Mpi_Comm &comm,int dv){
  n = size();
  i = rank();

  bi =dv;
  for (j=0; j < ceiling(log(n)); j++){
    xi = Shift_int(false,-1,pow(2,j),comm,bi);
    if (i < (n+1) - pow(2,j) && i % pow(2,j+1) == 1)
      {bi = max(xi,bi);}
    else
      {bi = bi;}
  }
  result = bi;
  return(result);
```

```
}

int intermediateMss(int x){
  comm = currentComm();
  s = intermediateScanl1(comm,x);
  m = intermediateScanr1(comm,s);
  b = m - s + x;
  mss = intermediateFold1(comm,b);
  return(mss);
}

void intermediateHarness(){
  i = rank();
  if (i==1) {a = 2;}
  else if (i == 2) {a = -4;}
  else if (i == 3) {a = 3;}
  else if (i == 4) {a = -1;}
  else if (i == 5) {a = 6;}
  else  {a = -3;}

  grp = currentGroup();
  comm = communicator(grp);
  aPrime = Gather_int(1,comm,a);
  if (i==1){print(aPrime);}
  if (i==1){print("---\n");}
  r = intermediateMss(a);
  if (i==1){print(r);}
}

int main(){
  intermediateHarness();
  Mpi_Finalize();
  return(0);
}
```

A.2.4 Runlogs

```
Mss> main

Specification Stage
===================
{1 := 2, 2 := -4, 3 := 2, 4 := -1, 5 := 6, 6 := -3, }
7

% mpirun -np 6 Mss

Dimension:  1
Shape     : <  6>
< 2 -4  3 -1  6 -3 >
---
7
```

Appendix B

Bibliography

Each citation in this bibliography is annotated by the page numbers in this thesis where the publication is cited.

Abelson, H. and Sussman, G. J. (1984). *Structure and Interpretation of Computer Programs*. MIT Press. (page 97)

Arvind and Nikhil, R. (1989). I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632. (page 22)

Ashcroft, E. A. and Wadge, W. W. (1985). *Lucid, the data-flow programming language*. Academic Press, New York. (page 22)

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., and Vanneschi, M. (1995). P^3l: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255. (page 20)

Bentley, J. (1984). Programming pearls: Algorithm design techniques. *Communications of the ACM*, 27(0):865–871. (page 133)

Bentley, J. (1986). Little languages. *Communications of the ACM*, 29(8):711–21. (page 102)

Bird, R. S. (1987). A calculus of functions for program derivation. Technical Monograph PRG-64, Oxford University Computing Laboratory. (page 17)

Bird, R. S. and de Moor, O. (1996). *Algebra of Programming*. Prentice Hall. (page 17, 26)

Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. (1999). Lava: Hardware design in Haskell. *SIGPLAN Notices*, 34(1):174–184. (page 103)

Blelloch, G. E. (1992). NESL: A nested data-parallel language. Technical Report CS-92-103, Carnegie Mellon University, School of Computer Science. (page 15)

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J., and Zagha, M. (1994). Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1). (page 16)

Brus, T., van Eekelen, M. C. J. D., van Leer, M. O., Plasmeijer, M. J., and Barendregt, H. P. (1987). Clean: A language for functional graph rewriting. In Kahn, G., editor, *Proceeding of the Third International Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384. Springer-Verlag. (page 61)

Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67. (page 26)

Carlsson, M. and Hallgren, T. (1993). FUDGETS: A graphical user interface in a lazy functional language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 321–330, New York, NY, USA. ACM. (page 98)

Church, A. (1941). *The Calculi of Lambda-Conversion.* Princeton University Press. (page 23)

Cole, M. (1989). *Algorithmic Skeletons: Structured management of parallel computation.* PhD thesis, Pitman / MIT. (page 20)

Danelutto, M., Di Meglio, R., Orlando, S., Pelagatti, S., and Vanneschi, M. (1992). A methodology for the development and the support of massively parallel programs. In *Future Generation Computer Systems*. North Holland. (page 20, 20)

Darlington, J., Field, A. J., Harrison, P. G., Harper, D., Jouret, G. K., Kelly, P. J., Sephton, K. M., and Sharp, D. W. (1991). Structured Parallel Functional Programming. In *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 31–52, Southampton, UK, June 5–7. Technical Report CSTR 91-07, University of Southampton. (page 21)

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q., and While, R. L. (1993). Parallel programming using skeleton functions. In *PARLE'93: Parallel Architectures & Languages Europe*, Lecture Notes in Computer Science, pages 146–160. Springer-Verlag. <http://theory.doc.ic.ac.uk/tfm/papers/KellyP/SkeletonsParle93.ps.Z>. (page 21)

Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands. ACM. (page 103)

Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. (1999). Calling Hell from Heaven and Heaven from Hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34 of *SIGPLAN Notices*, pages 114–125. (page 111)

Finne, S. and Peyton Jones, S. (1995). Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, Maastrict, Netherlands. <http://www.dcs.gla.ac.uk/~sof/haggis/composing-haggis.ps.gz>. (page 101)

Fissgus, U., Rauber, T., and Rünger, G. (1999). A framework for generating task parallel programs. In *Proceedings of Frontiers '99: The 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 72–80, Annapolis, Maryland. IEEE Computer Society. (page 32)

Fitzpatrick, S., Harmer, T. J., and Boyle, J. M. (1994). Deriving efficient parallel implementations of algorithms operating on general sparse matrices using automatic program transformation. In Buchberger, B. and Volkert, J., editors, *Parallel Processing: CONPAR 94-VAPP VI*, pages 148–159. Department of Computer Science, Queen's University of Belfast, Springer-Verlag. (page 27)

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee. <http://www.eece.ksu.edu/pvm3/ug.ps>. (page 13)

Gill, A. (1999). A Haskell HTML combinator library. <http://www.cse.ogi.edu/~andy/html/intro.htm>. (page 98)

Goodman, J. and O'Donnell, J. (1999). Nondeterminism in the APM methodology. In *Proceedings of First Scottish Functional Programming Workshop.* <http://www.dcs.gla.ac.uk/~joy/research/nondet.ps.gz>. (page 34)

Goodman, J., O'Donnell, J., and Rünger, G. (1998). Refinement transformation using Abstract Parallel Machines. In *Glasgow Workshop on Functional Programming 1998*, Glasgow University, Scotland. Revised version at <http://www.dcs.gla.ac.uk/~joy/research/maptri.ps>. (page 34, 104)

Gorlatch, S. (1996). Stages and transformations in parallel programming. In *Abstract Machine Models*, pages 147–161. IOS Press. <http://brahms.fmi.uni-passau.de/cl/papers/GorPel99a.ps.gz>. (page 27)

Gorlatch, S. (1998). Abstraction and performance in the design of parallel programs. Technical Report MIP-9802, University of Passau, Germany. (page 27)

Gorlatch, S. and Pelagatti, S. (1999). A transformational framework for skeletal programs: Overview and case study. In Rohlim, J. et al., editors, *Parallel and Distributed Processing. IPPS/SPDP'99 Workshops Proceedings*, Lecture Notes in Computer Science 1586, pages 123–137. <http://brahms.fmi.uni-passau.de/pub/local/parallel/papers/Gor96a.ps.Z>. (page 28)

Grelck, C. (1998). Shared memory multiprocessor support for SAC. In *Conference on Implementation of Functional Languages (IFL'98)*, Lecture Notes in Computer Science. University College, London, Springer-Verlag. <http://www.informatik.uni-kiel.de/~sacbase/papers/mt-support-london-98.ps.gz>. (page 154)

Grelck, C. and Scholz, S.-B. (1995). Classes and objects as basis for I/O in SAC. In *Conference on Implementation of Functional Languages (IFL'95).* Chalmers University of Technology. <http://www.informatik.uni-kiel.de/~sacbase/papers/sac-classes-objects-bastad-95.ps.gz>. (page 148)

Grimshaw, A. S. (1993). Easy-to-Use Object-Oriented Parallel Processing With Mentat. *IEEE Computer*, 26(5):39–51. <ftp://ftp.cs.virginia.edu/pub/techreports/CS-92-32.ps.Z>. (page 22)

Grimshaw, A. S., Loyot, E. C., and Weissman, J. B. (1991). Mentat programming language (MPL) reference manual. Technical Report CS-91-32, Department of Computer Science, University of Virginia. <ftp://ftp.cs.virginia.edu/pub/techreports/CS-91-32.ps.Z>. (page 22)

Hammond, K. (1994). Parallel functional programming: An introduction. In Hong, H., editor, *PASCO'94-International Symposium on Parallel Symbolic Computation*, volume 5 of *Lecture Notes in Computer Science*, pages 181–193, Hagenberg/Linz, Austria, 26–28 September. RISC-Linz, World Scientific. <http://www.dcs.st-and.ac.uk/~kh/papers/pasco94/pasco94.html>. (page 23)

Hammond, K., Loidl, H. W., and Partridge, A. (1994). Improving granularity in parallel functional programs: A graphical winnowing system for Haskell. In *Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 111–126, Ayr, Scotland, September 12–14. Springer-Verlag. <ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/authors/Andrew_Partridge/gran.ps.Z>. (page 24, 121)

Hammond, K. and Michaelson, G., editors (1999). *Research Directions in Parallel Functional Programming*. Springer-Verlag. (page 23)

Herath, J., Yuba, T., and Saito, N. (1987). Dataflow computing. In *Parallel Algorithms and Architectures*, pages 25–36. Springer-Verlag. Lecture Notes in Computer Science 269. (page 22)

HPF Forum (1993). *High Performance Fortran Language Specification*. High Performance Fortran Forum, Rice University, Houston, Texas, 1.1 edition. <http://www.erc.msstate.edu/hpff/hpf-report-ps/hpf-v11.ps>. (page 14)

Hudak, P. (1986). Parafunctional programming. *IEEE Computer*, 19:60–71. (page 23)

Hudak, P. (1998). Modular domain specific languages and tools. In Devanbu, P. and Poulin, J., editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press. (page 102)

Hudak, P. and Jones, M. P. (1994). Haskell vs. Ada vs. C++ vs Awk vs ... an experiment in software prototyping productivity. Technical report, Yale University. <ftp://nebula.cs.yale.edu/pub/yale-fp/papers/NSWC/jfp.ps>. (page 103)

Hudak, P., Peterson, J., and Fasel, J. (1999). A gentle introduction to Haskell 98. <http://www.haskell.org/tutorial/haskell-98-tutorial.pdf>. (page 99)

Hudak, P. and Sundaresh, R. S. (1989). On the expressiveness of purely functional I/O systems. Technical report, Yale University. (page 100)

Huet, G. P. and Lang, B. (1978). Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55. (page 165)

Hughes, J. (1995). The design of a pretty-printing library. In Jeuring, J. and Meijer, E., editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag. <http://www.cs.chalmers.se/~rjmh/Papers/pretty.ps>. (page 98)

Hutton, G. (1990). Parsing using combinators. In Davis, K. and Hughes, R. J. M., editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 353–370, London, UK. Springer-Verlag. British Computer Society Workshops in Computing Series. (page 98)

Hutton, G. (1992). Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343. (page 98)

Hutton, G. and Meijer, E. (1998). Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444. (page 101)

Jay, C. and Steckler, P. (1997). The Functional Imperative: Shape! Technical Report 06, University of Technology, Sydney. <http://www-staff.socs.uts.edu.au/~cbj/Publications/functional_imperative.ps.gz>. (page 16)

Jay, C. B. (1995). A semantics for shape. *Science of Computer Programming*, 25:251–283. <http://www-staff.socs.uts.edu.au/~cbj/Publications/shape_semantics.ps.gz>. (page 17)

Jay, C. B. (1998). The FISh language definition. <http://www-staff.socs.uts.edu.au/~cbj/Publications/fishdef.ps.gz>. (page 16)

Jay, C. B. (2000). Costing parallel programs as a function of shapes. *Science of Computer Programming*. in press. <http://www-staff.socs.uts.edu.au/~cbj/Publications/costing_parallel_scp.ps.gz>. (page 17)

Kelly, P. H. J. (1989). *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press. (page 23)

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag. (page 30)

King, D. J. and Wadler, P. (1993). Combining monads. Technical report, University of Glasgow. <ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/papers/combining-monads.ps.Z>. (page 102)

Knee, S. (1994). Program development and performance prediction on BSP machines using OPAL. Technical Report PRG-TR-18-94, Oxford University Computing Laboratory. (page 18)

Leijen, D. and Meijer, E. (2000). Domain-specific embedded compilers. *SIGPLAN Notices*, 35(1):109–122. (page 101, 108, 160)

Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. In *Conference Record of POPL 95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343. ACM SIGACT and SIGPLAN, ACM. (page 102)

Loidl, H.-W., Morgan, R., Trinder, P., and Poria, S. (1998). Parallelising a large functional program or: Keeping LOLITA busy. *Lecture Notes in Computer Science*, 1467. (page 24)

McColl, W. F. and Miller, Q. (1995). The GPL language: Reference manual. Technical Report ESPRIT GEPPCOM Project, Oxford University Computing Laboratory. (page 18)

McGraw, J., Skedzielewski, S., Allan, S., Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B., and Thomas, R. (1985). SISAL — Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, University of California. (page 22)

McGraw, J. R. (1993). Parallel functional programming in Sisal: Fictions, facts, and future. In *Advanced Workshop, Programming Tools for Parallel Machines*. (page 22)

Meijer, E. and Jeuring, J. (1995). Merging monads and folds for functional programming. In Jeuring, J. and Meijer, E., editors, *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer-Verlag, Berlin. `<http://www.cs.ruu.nl/~erik/mmfffp.ps>`. (page 102)

Miller, R. (1993). A library for Bulk Synchronous Parallel programming. In *Proceedings of the BSC Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, pages 100–108. (page 18)

Milner, R. (1983). A proposal for Standard ML. *Polymorphism: The ML/LCF/Hope Newsletter*, I(3). Also appeared in the Conference Record of the ACM Symposium on Lisp and Functional Programming, Austin, Texas, August 1984, pages 184-197 and as Tech. Report CSR-157-83, University of Edinburgh, Edinburgh, Scotland, 1983. (page 98)

Mitra, P., Payne, D., Shuler, L., van de Geijn, R., and Watts, J. (1995). Fast collective communication libraries, please. Technical Report CS-TR-95-22, University of Texas, Austin. (page 53)

MPI Forum (1995). *MPI: A Message Passing Interface Standard.* Message Passing Interface Forum, University of Tennessee. (page 5, 13)

O'Donnell, J. (1987). Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382, Amsterdam. North-Holland. (page 163)

O'Donnell, J. (2000). The collective and individual semantics in functional SPMD programming. In Mohnen, M. and Koopman, P., editors, *Proceedings of the*

12th International Workshop on Implementation of Functional Languages, number 2000–7 in Aachener Informatik-Berichte, pages 5–13, RWTH Aachen, Germany. (page 80, 145)

O'Donnell, J., Rauber, T., and Rünger, G. (2000). Cost hierarchies for Abstract Parallel Machines. In *13th International Workshop on Languages and Compilers for Parallel Computing*, Draft Proceedings. Springer-Verlag. (page 88)

O'Donnell, J. and Rünger, G. (1995). A case study in parallel program derivation: the heat equation algorithm. In *Glasgow Workshop on Functional Programming 1994*, pages 167–183. Springer-Verlag Workshops in Computation. <http://www.dcs.gla.ac.uk/~jtod/publications/heatEqu-FPG94.ps.gz>. (page 104)

O'Donnell, J. and Rünger, G. (1995). An explanatory formal derivation of a parallel binary addition circuit. Technical Report TR-1995-19, Department of Computing Science, University of Glassgow. (page 34)

O'Donnell, J. and Rünger, G. (1997a). A coordination level functional implementation of the hierarchical radiosity algorithm. In *Glasgow Workshop on Functional Programming 1997*, Glasgow University, Scotland. (page 104)

O'Donnell, J. and Rünger, G. (1997b). A methodology for deriving parallel programs with a family of parallel abstract machines. In *Third International Euro-Par Conference*, volume 1300 of *Lecture Notes in Computer Science*, pages 662–669. Springer-Verlag. <http://www.dcs.gla.ac.uk/~jtod/publications/apm-EuroPar97.ps.gz>. (page 7, 32, 58, 104)

O'Donnell, J. and Rünger, G. (2000). Abstract parallel machines. *Computers and Artificial Intelligence (continued as Computing and Informatics)*, 19(2):105–129. (page 32)

Pacheo, P. S. (1996). *Parallel Programming with MPI.* Morgan Kaufmann, San Francisco, California. (page 40)

Peterson, J., Hudak, P., and Elliott, C. (1999). Lambda in motion: Controlling robots with Haskell. *Lecture Notes in Computer Science*, 1551:91–105. (page 103)

Peyton Jones, S. (1996). Compiling Haskell by program transformation: A report from the trenches. In *European Symposium on Programming (ESOP'96)*, volume 1058 of *Lecture Notes in Computer Science*. Springer-Verlag. (page 165)

Peyton Jones, S. (1999). Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University. (page 98, 101)

Peyton Jones, S., Nordin, T., and Oliva, O. (1998). $C--$: A portable assembly language. *Lecture Notes in Computer Science*, 1467. (page 154)

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages.* Prentice-Hall. (page 23)

Pfister, G. (1998). *In Search of Clusters.* Prentice Hall, second edition. (page 18)

Pritchard, D. (1988). Mathmatical models of distributing computation. In Muntean, T., editor, *Parallel Programming of Transputer Based Machines*, pages 25–36, Amsterdam. IOS Press. (page 20)

Pritchard, D. J., Askew, C. R., Carpenter, D. B., Glendinning, I., Hey, A. J. G., and Nicole, D. A. (1987). Practical parallelism using transputer arrays. In *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume I: Parallel Architectures*, volume 258 of *Lecture Notes in Computer Science*, pages 278–294, Eindhoven, The Netherlands. Springer. (page 20)

Rauber, T. and Rünger, G. (1995). Parallel numerical algorithms with data distribution types. Technical Report 07-95, Computer Science Department, University of Saabrücken, Germany. (page 40, 162)

Rauber, T. and Rünger, G. (1996a). The compiler TwoL for the design of parallel implementations. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 292–301, Boston, Massachusetts. IEEE Computer Society Press. <http://www.informatik.uni-halle.de/~rauber/webpage/13_twoL.ps>. (page 4, 30)

Rauber, T. and Rünger, G. (1996b). Deriving structured parallel implementations for numerical methods. *Microprocessing and Microprogramming*, 9(3):181–202. (page 5, 19, 30)

Sage, M. (2000). FranTk – a declarative GUI language for Haskell. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming (ICFP'2000)*. to appear. (page 103)

Scholz, E. (1999). Imperative streams – A monadic combinator library for synchronous programming. *SIGPLAN Notices*, 34(1):261–272. (page 101)

Scholz, S.-B. (1994). Single Assignment C – functional programming using imperative style. In *Conference on Implementation of Functional Languages (IFL'94)*. University of East Anglia, Norwich, UK. <http://www.informatik.uni-kiel.de/~sacbase/papers/sac-overview-norwich-94.ps.gz>. (page 42, 55, 141)

Scholz, S.-B. (1997). On programming scientific applications in SAC – A functional language extended by a subsystem for high-level array operations. In *Conference on Implementation of Functional Languages (IFL'96)*, Lecture Notes in Computer Science, pages 85–104. Springer-Verlag. <http://www.informatik.uni-kiel.de/~sacbase/papers/scientific-applications-sac-bonn-96-ps.gz>. (page 55)

Scholz, S.-B. (1998). On defining application-specific high-level array operations by means of shape-invariant programming facilities. In *Proceedings of the APL98 International Conference*, pages 40–45. ACM-SIGAPL. <http://www.informatik.uni-kiel.de/~sacbase/papers/sac-defining-array-ops-rom-98.ps.gz>. (page 55)

Skillicorn, D. B. (1990). Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50. (page 17)

Skillicorn, D. B. (1991). Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–158. (page 17)

Skillicorn, D. B., Hill, J. M. D., and McColl, W. F. (1996). Questions and answers about BSP. Technical Report 15-96, Programming Research Group, Oxford University Computing Laboratory. <ftp://ftp.comlab.ox.ac.uk/pub/Packages/BSP/papers/SkillHillMcColl_QA.ps.gz>. (page 18)

Skillicorn, D. B. and Talia, D. (1998). Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169. <http://www.acm.org:80/pubs/citations/journals/surveys/1998-30-2/p123-skillicorn/>. (page 14)

Smetsers, S., Barendsen, E., van Eekelen, M., and Plasmeijer, R. (1994). Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. *Lecture Notes in Computer Science*, 776:358–379. (page 61)

Trinder, P. W., Hammond, K., Loidl, H.-W., and Peyton Jones, S. L. (1998). Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60. (page 24)

Trinder, P. W., Hammond, K., Mattson, J. S., Partridge, A. S., and Peyton Jones, S. (1996). GUM: a portable parallel implementation of Haskell. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, pages 79–88, Philadelphia, Pennsylvania. <ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/authors/Philip_Trinder/gumFinal.ps.Z>. (page 24)

Tullsen, M. and Hudak, P. (1998). An intermediate meta-language for program transformation. Technical Report YALEU/DCS/RR-1154, Yale University. (page 165)

Turner, D. A. (1985). Miranda: A non-strict functional language with polymorphic types. In Jouannaud, J.-P., editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag. (page 16)

Valiant, L. G. (1989). Bulk synchronous parallel computers. Technical Report TR-08-89, Computer Science, Harvard University. (page 18)

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111. <http://www.acm.org/pubs/toc/Abstracts/0001-0782/79181.html>. (page 162)

Wadler, P. (1990). Comprehending Monads. In *LISP'90, Nice, France*, pages 61–78. ACM. (page 99)

Wadler, P. (1992). Monads for functional programming. Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag. <ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/authors/Philip_Wadler/monads-for-fp.dvi>. (page 99)

Wallace, M. and Ranciman, C. (1999). Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34 of *SIGPLAN Notices*, pages 148–159, N.Y. ACM. (page 98)

Weber, M. (2000). hMPI, a Haskell binding for MPI. <http://www-i2.informatik.rwth-aachen.de/Software/Haskell/libs>. (page 121)

Wilson, G. V. (1995). *Practical Parallel Programming*. MIT Press. (page 2)

Winstanley, N. (1999a). Parallel programming by transformation. In *Fifth International Euro-Par Conference*, Lecture Notes in Computer Science. Springer-Verlag. (page 40)

Winstanley, N. (1999b). What the hell are monads? <http://www.dcs.gla.ac.uk/~nww/Monads.html>. (page 99)

Winstanley, N. (2000a). An embedded implementation of PEDL. <http://www.dcs.gla.ac.uk/~nww/PEDL/Implementation>. (page 104)

Winstanley, N. (2000b). A Haskell model of the semantics of PEDL. <http://www.dcs.gla.ac.uk/~nww/PEDL/Semantics>. (page 59)