UNIVERSITY
of
GLASGOW

Department of
Computing Science

Incremental Program Transformation Using Abstract Parallel Machines

Joy Ai-Leen Goodman

Submitted for the degree of Doctor of Philosophy in Computing Science at the University of Glasgow

June 2002

Abstract

Parallel processing is a key area of high-performance computing, providing the processing power to meet the needs of computation-intensive problems. Much research effort has therefore been invested in this area and it is growing. However, there are still several challenges that hinder its widespread use. In particular, parallel programs can be difficult to write because of the increase in the number of details to keep track of and design decisions to be made. Portability is also an important issue because the efficiency of a program depends heavily on the target machine. Another challenge arises when the issue of correctness is considered as the increased complexity and nondeterminism of parallel systems makes reasoning about them hard and renders traditional methods of testing unreliable.

This thesis presents a methodology for developing parallel programs that addresses these issues. In it, executable parallel programs are derived incrementally from high-level specifications. A specification is given initially in mathematical notation and changed into an abstract functional specification. This is then transformed through a series of stages, during which additional information is given about the program, the target architecture and the parallelism. Finally it is transformed into the target language to produce an executable parallel program. This thesis uses C+MPI as an example target language, but many languages are possible.

This methodology addresses several of the challenges of parallel programming. In particular, its incremental framework allows decisions about the program and its parallelism to be made one at a time, instead of all at once, easing the burden on the programmer and simplifying the decisions. Reasoning about the program is also made possible through the use of a pure functional language, such as Haskell, for intermediate versions of the program, as the program can then be transformed using equational reasoning, a correctness-preserving technique.

The methodology is based on previous work on Abstract Parallel Machines and program derivation, which this thesis develops. It presents the basic infrastructure needed in the methodology, and therefore investigates how parallel systems can be modelled and manipulated in Haskell, and how the resultant programs can be transformed. It augments the basic methodology with the ability to introduce and reason about some key parallel programming features, including data distributions and program optimisations. The work is supported and demonstrated through two case studies.

Acknowledgements

Thanks are due to many people for their help and support during the production of this thesis.

Firstly, I would like to thank my supervisor, John O'Donnell, for suggesting the research project that led to this thesis, and for his friendly and supportive help and advice.

Many thanks are also due to my examiners, Gudula Rünger and Muffy Calder, who provided many helpful suggestions and comments that improved this thesis.

I would also like to thank the Dependable Computing and Foundations research group at Herriot-Watt University for the use of their Beowulf machine. This was valuable for carrying out timings and profiles of case study programs. A special thanks is due to Hans-Wolfgang Loidl for sorting out the adminstrative details on this machine, helping with the GpH profiling tools, and providing helpful comments.

SCOFPIG (the Scottish Functional Parallel Interest Group) was also very valuable. It provided interesting discussions, a forum for presenting my research and some useful ideas and feedback. I would especially like to thank Phil Trinder and Kevin Hammond for their useful suggestions.

Thanks are also due to the programming group at Passau University, especially to Christian Lengauer for helpful discussions. I am also very grateful to Christoph Herrmann for helpful and friendly discussion and advice.

I gratefully acknowledge the financial support provided by EPSRC through research studentship 97305990.

Finally I would like to thank my friends and family for all their support and encouragement. This thesis is dedicated to my parents and my grandmother.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A key challenge in computer science today is the provision of sufficient processing power to meet the users' needs. Many areas of computing require high levels of performance to cope with large quantities of data, produce results within a restricted time scale and increase productivity.

One way this performance can be achieved is through the use of parallel processing, in which multiple processors cooperate on a single problem. When combined with the increases in speed in sequential machines, parallel processing allows levels of performance which would otherwise not be possible.

Much research effort has therefore been invested in this area and it is growing. However, there are still several challenges that hinder its widespread use. Firstly, parallel programs can be very difficult to write. As several tasks can be executed at the same time, often with communication between them, there are many extra details to keep track of. Not only does this increase a program's complexity, but it introduces many design decisions that need to be made. Among others, the placement and movement of tasks and data must be decided. These decisions must made correctly to obtain maximum efficiency, and each affects the others. The efficiency of a program also depends heavily on the target machine, and it may need to be rewritten to run efficiently on other machines, or, in many cases, even to run on them at all. Another challenge arises when the issue of correctness is considered. The increased complexity and nondeterminism of parallel systems makes reasoning about them hard and renders traditional methods of testing unreliable.

Various solutions to these challenges have been suggested and implemented, and several are discussed in detail in Chapter 8. However, in the main, these tackle one or more of the challenges without the others. This thesis therefore combines ideas from several different areas.

In particular, parallel combinators, or higher-order functions, increase the level of abstraction in a program, allowing the programmer to focus on the algorithmic aspects of the program, rather than on the parallel details. This makes it easier to write parallel programs, but prevents the manipulation of parallel details necessary to obtain high efficiency. Libraries of standard functions, such as skeletons, help with portability, through the availability of multiple versions for different machines. Incremental derivation approaches also ease the writing process by separating design decisions from each other, and considering them one at a time, and formal program transformation techniques help to maintain the correctness of the program. All of these areas are described in more detail later in the thesis.

This thesis presents a methodology based on these areas. It is based on an incremental derivation approach that allows formal program transformation techniques to convert a program from one version to the next. Within each version of the program, parallel combinators encapsulate the parallelism. These exist in many forms, at different levels of abstraction and for different target machines. The level of abstraction decreases during a derivation, allowing early stages of the derivation to be reused for other targets.

This chapter firstly presents some of the key ideas behind the methodology in more detail, setting the work in the thesis in context, before presenting the contributions of the thesis itself. It then makes several general comments about the research, before describing the structure of the remainder of the thesis.

1.1 Background

The methodology is based on ideas from several fields. Some of these key fields are introduced here, together with the basic ideas in the methodology itself. These and other relevant fields are discussed in greater depth in Chapter 8.

1.1.1 Incremental programming

It is common practice to write a program directly in its final form, as a program in, for example, C augmented with MPI, a library of message-passing functions. However this process can be difficult as all of the design decisions are mixed up together. This is a particular problem in parallel programming since there is a large number of decisions to be made, such as the placement of data and tasks and the introduction of optimisations.

This thesis uses a different approach, *incremental programming*, in which the decisions are made and introduced into the program one at a time, in a logical sequence. The programming process is therefore divided into a sequence of simpler steps, and the program goes through a series of versions before it arrives at its final form. Both these versions and the transformations between them are well-defined.

Incremental programming has many advantages, the main one of which is its separation of the decisions. They no longer need to be made all at once: each can be considered and made separately, thus simplifying the decision-making process. In addition, the well-defined transformations help the programmer to incorporate the results of decisions; they also help with the programming process itself. In this aspect, incremental programming bears similarities with structured programming and software engineering.

There are various existing incremental systems for parallel programming, some of which are described in Section 8.2.

1.1.2 Program transformation

Incremental programming requires some method to transform the program between its different versions, keeping its "meaning", the results it produces, the same, while changing its parallel behaviour and performance.

There is a variety of such program transformation methods available. Such methods are usually based on sets of rules, which, when applied to a program, transform it in the required ways. These can be of varying degrees of formality, may be applied by hand or automatically, and may or may not be proved correct.

This thesis uses a standard program transformation technique in functional programming called *equational reasoning*. It preserves the meaning of a program, and so allows the transformation rules to be proved correct. However, it does not insist on proving every detail. Some properties, such as arithmetic identities, can be assumed to be true. This allows greater flexibility in a derivation.

Equational reasoning works by replacing expressions with other expressions of equal value. For example, function calls can be replaced by the instantiated function's body and vice-versa, and arithmetic properties can be used. Previously proved lemmas can also be used to transform the program. In the proofs of such lemmas, additional techniques such as structural induction can be combined with equational reasoning. More about equational reasoning can be found in introductory books such as [Bir98].

1.1.3 Previous work on the methodology

The methodology described in this thesis was first proposed in 1997 by John O'Donnell and Gudula Rünger in [OR97], and its basic structure is also described in [OR00].

It is a derivation methodology for parallel programs, based on the incremental programming approach described above. Each step in a derivation is expressed using an appropriate model of parallelism and can be proved equivalent to the previous step using equational reasoning. The models of parallelism are defined using ***abstract parallel machines*** (APMs). These specify some global properties of the model, such as the number of sites and the state type for each site, along with a set of parallel operations on the model. Due to the key role of these APMs, the methodology is often known as the *APM methodology.*

A derivation starts off with an abstract version of the program which is gradually made more and more concrete by the addition of parallel details. The intermediate versions could be written in a range of languages, but Haskell has always been used because it has several features particularly suited to such a methodology (see Section 1.3.3). Finally, a version is reached which contains all of the needed parallel details. This is then transformed into the target language.

However, there are often several ways to transform each version of the program, and several options when a decision is to be made. Different choices may lead to different final programs. The methodology therefore has a branching structure of possible derivations as shown in Figure 1.1.

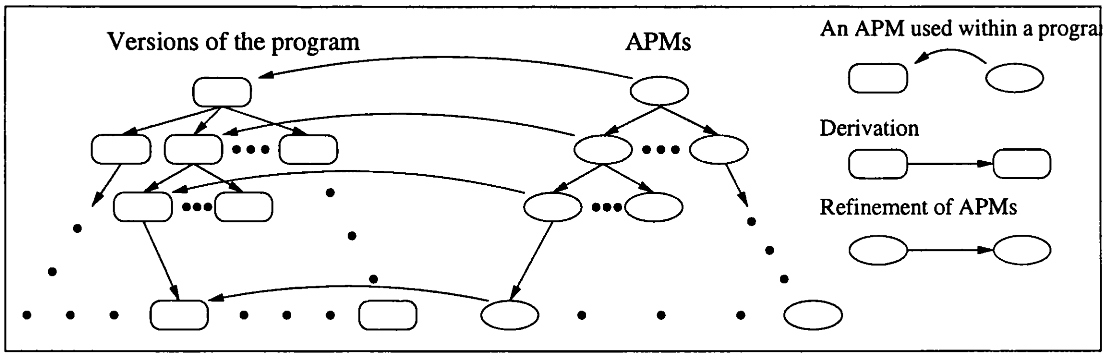


Figure 1.1: The basic structure of the methodology

Connected to each stage in the methodology is an APM, which encapsulates the parallelism used in the program at that stage. These APMs therefore separate the parallel from the algorithmic aspects of a program. Figure 1.1 shows how these APMs form a tree in which APMs lower down in the tree are more concrete about parallel details than those further up it.

Programs are transformed from one stage to the next by equational reasoning. The transformations can be divided into two main categories. In ***horizontal transformations***, both stages use the same APM, and the transformation changes the algorithm by manipulating the functions at the current level of abstraction. Such transformations can be used to improve the program's efficiency or to prepare the way for other transformations. In ***vertical transformations***, on the other hand, the stages use different APMs, but the algorithm essentially stays the same. These transformations bring the program closer to the target, reducing its level of abstraction.

ParOps

The initial APM paper [OR97] introduces the ParOp form. This is a canonical form for the operations in an APM. It is capable of expressing many types of computation, and specifies the parallelism in a program explicitly, while remaining more abstract than many other models. It views the system as a set of sites, each of which may be a processor, a thread in a processor, a location in the memory of a sequential processor, or an abstract location which could be any of these. The nature of a site depends on the current level of abstraction in the derivation. The level of abstraction may be implicit or may be indicated in the APM.

The ParOp definition makes the initial and final states of the sites and the communication between them explicit. It is a generalised relation which can represent a multitude of parallel operations and is parametrised by functions f_i and g_i which describe what the processors do to their states, what they output to the other processors and which other processors they accept inputs from.

This is illustrated in Figure 1.2. P_1 to P_n are the sites, which can communicate with each other using an interconnection network. At the start of the computation, some input values, $x_1, \ldots, x_r$, may be fed into the system. Each processor then chooses some of these values and some of the values calculated by other processors using its function, g_i. The processor uses them and any values already in its state with its other function f_i to calculate a new state value and a value A_i which is returned to the network so that other processors can access it. Some of these values may be chosen to be outputs, $y_1, \ldots, y_t$, from the system.

The system can deadlock if there is a cyclic dependency between the values needed and calculated by the processors, but this is acceptable because such systems exist in real life.

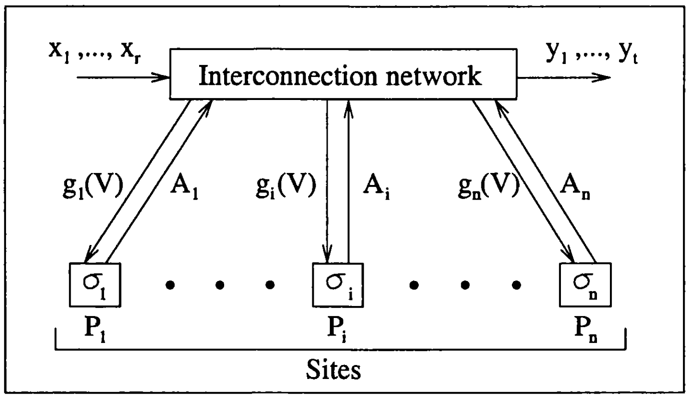


Figure 1.2: Operation of ParOp

The ParOp definition can be specified as follows, where σ_i and σ'_i are the initial and final states in site i, and g_{out} selects the output values from the system.

$$\begin{aligned}
\text{ParOp } (\sigma_1, \ldots, \sigma_n)\ (x_1, \ldots, x_r) &= ((\sigma'_1, \ldots, \sigma'_n),\ (y_1, \ldots, y_t)) \\
\text{where} \quad (\sigma'_i, A_i) &= f_i(\sigma_i,\ g_i\ (V)) \\
(y_1, \ldots, y_t) &= g_{\text{out}}\ V \\
V &= ((x_1, \ldots, x_r), A_1, \ldots, A_n)
\end{aligned}$$

These details are given for the sake of interest and comparison, but the ParOp form is not used explicitly in this thesis. Instead APM functions are given as Haskell definitions. However this Haskell method is closely related to the ParOp method, and, in fact, the Haskell definitions can be viewed as ParOps because the ParOp form can be written in Haskell, as shown, by example, in [ELG99]. By then inlining the f and g functions and further manipulating the definition with equational reasoning, other Haskell definitions can be obtained. See Section 2.3.5 for an example of this and for more information about the role of ParOps in this thesis.

Other work

The APM model resulted from the analysis of two case studies in deriving parallel programs [OR94, OR95]. However, because the APM methodology is a relatively recent idea, this thesis presents the first case studies done using the methodology itself.

For the same reason, other work on the methodology is preliminary, but there are several research projects in the area.

The PolyAPM project, based at Passau University, stands for "Abstract Parallel Machines for the Polytope Model", and uses APMs as a tool to investigate the order of code generation steps in the polytope model. This model is a mathematical execution model on which static parallelisation methods for scientific computing can be based. PolyAPM uses APMs as targets for the code generation at each step of such a method. So far, only fairly high level APMs have been used, but, in future, the group hopes to design more APMs at lower levels of abstraction, down to C, and perform the transformations automatically in a parallelising compiler. Existing work is described in [EGL98, ELG99].

Noel Winstanley has also worked on APMs. His prototype system, PEDL (Parallel Embedded Derivation Languages) [Win01, Win99], is based on the APM approach in a restricted context, the derivation of SPMD array based numerical programs. This work bears many similarities to this thesis—the derivations are structured in a similar way, and the intermediate stages are written in Haskell. However, these stages are written in specialised sub-languages of Haskell, rather than using APMs as such. The restricted context also allows a higher degree of automation, but much less flexibility. The work tends towards a Haskell-oriented derivation approach, and is less clearly related to the APM methodology.

O'Donnell, Rauber and Rünger have been working on an extension of the APM methodology with a hierarchy of cost models [ORR01]. I have also carried out work with John O'Donnell on incorporating the ability to express nondeterminism into the methodology [GO99].

Some of the work in this thesis has been previously published. The structure of the methodology was introduced in [Goo98]. Part of the first case study in Chapter 5 was published in [GOR98], and the introduction of load balancing to this study in [GO01]. [Goo01b] shows how some of the pipelining optimisations in Chapter 6 are introduced to the Gaussian Elimination case study.

1.2 Contributions of the thesis

This thesis describes in detail the Abstract Parallel Machine (APM) methodology for deriving parallel programs, focusing especially on its use for general SPMD programs. This is a style of programming common on distributed memory MIMD machines, and in wide-spread use today. It is described in Section 2.10.1. In particular the thesis:

- clarifies the essence of the methodology, with explanations as to why things are done the way they are, and discussion of important issues,
- gives a detailed structure for the methodology and identifies its key stages with details of the APMs and their implementations for these stages and sample other stages,

- shows how parallel systems can be modelled in Haskell at different levels of abstraction, providing different amounts of information about the parallelism,
- demonstrates how this can be used to support transformations, including the introduction of optimisations, to a parallel program,
- gives the transformation rules and lemmas needed to perform a range of simple derivations from start to finish, including rules for transforming a mathematical specification into Haskell, performing various optimisations, dealing with monads, and introducing input and output details,
- divides these transformations into logical and manageably-sized steps, which deal with a single feature or decision at a time,
- extends the basic methodology with ability to introduce and reason about data distributions and program optimisations, such as load balancing and communication-specific optimisations,
- describes how decisions can be made in the context of the methodology, illustrating this with examples including the choice of data distributions and static load balancing, and
- demonstrates the methodology in practice through two case studies. The second, in particular, shows how detailed communication-specific optimisations can be introduced into a program.

In summary, this thesis:

- investigates how parallel systems can be modelled and manipulated in Haskell, and how the resultant programs can be transformed.
- brings the methodology significantly closer to a usable state, providing the support needed for the derivation of basic C+MPI programs, and augmenting this to cope with some additional features.
- critically assesses APMs expressed in Haskell after carrying out detailed work on them.

1.3 General comments about the research

It is worth-while making some general comments about the research and this thesis before the presentation of more specific material. These comments concern both the scope of the thesis and the nature of the APM methodology.

1.3.1 Completeness

Although the methodology can, in principle, be used for a wide range of parallel programs and languages, this thesis is limited in length and time, and therefore restricts its scope to general SPMD programs, which are described in Section 2.10.1. It targets C+MPI as an example to allow the entire derivation process to be demonstrated. C+MPI is used because it is common, widely-available and implemented on multiple machines including networks of workstations. This thesis also focuses on programs involving finite sequences, such as arrays, rather than other data structures, because these are common in parallel programming, relatively simple and illustrate the key ideas.

Not all possible APMs and transformation rules are given, either for general programs or for C+MPI programs. This is because there is an enormous number of possible techniques and optimisations which can be employed, more than can be discussed in this thesis, and new techniques and insights can still be produced.

Therefore, the methodology is designed to be expandable. The stages and transformations in the thesis are included as much to give examples of how new stages and transformations can be written and included as to provide implementations which can be used to derive programs.

1.3.2 Flexibility

The methodology was designed with flexibility in mind. This is manifest in several ways:

- As mentioned above, the APM methodology is extensible. A programmer can add new APMs to model different situations and targets, along with corresponding transformation rules for these APMs. The APMs and transformation rules in this thesis can serve as examples to demonstrate this process.
- Individual APMs are also extensible, particularly the high level APMs near the start of a derivation. These allow any parallel operation to be encapsulated in a new APM function. Lower level APMs are less extensible because they model the specific operations provided by a particular language or machine. However they may still be extended by the programmer to be more realistic and extra functions can be added that are implemented in terms of the standard function set.
- The order of steps in the methodology is not fixed. Although the APMs are arranged in order of decreasing abstraction, and movement between them is only possible in this direction, there is a large degree of flexibility in the transformations. They don't have to be carried out on the whole program at once—it is possible to have parts of the program using different APMs. The same transformation can also be carried out multiple times. Several transformations, such as load balancing, don't change the APM, and can be done at various points in the derivation.
- Some stages may be skipped altogether. In the methodology the formal steps are kept small to simplify the transformations, but such small steps can be tedious to do by hand. It is possible to skip some of these, or to combine several of them into a larger transformation.
- While it is possible to do the transformations formally, it is also possible to do some or all of them informally, if so desired. Some transformations are sufficiently simple or commonplace that the programmer may feel capable of carrying them out correctly without the need for formalism. This speeds up and simplifies the transformation process, but produces a proof with a coarse degree of precision, or, in some cases, no proof at all. If a more detailed proof is later required, these steps can be redone formally.

These points are discussed further in the body of the thesis.

1.3.3 Language

The methodology uses a pure non-strict functional programming language throughout. Other languages are possible, but there are various advantages of such a language, some of which are given in [OR94]:

- Functional languages are useful for specification because they allow programs to be written in an abstract way which is close to the mathematics, encapsulating common operations in higher-order functions. They also do not introduce any extraneous data dependencies. This is examined further in Section 2.4.2.

- While functional programs can be abstract and high-level, they can also be low-level, and express low-level technical details. For example, Section 2.10 describes a low-level Haskell model of a parallel system with MPI functions. This ability is important because it allows the same language to be used for all the levels in the methodology.
- Pure non-strict functional programs can be transformed using equational reasoning, which retains their meaning, as described in Section 1.1.2. It is therefore possible to transform a high-level functional program into a low-level one which fulfils the same specification, as shown in Chapter 3.
- Functional programs are actually programs and hence can be executed. This allows the programs at intermediate stages in a derivation to be tested, catching mistakes early on.

In principal, any such language would do, but in practice Haskell [Bir98, PJea99] is used because it is well-supported and the most widely used of these languages today.

1.3.4 Execution targets

This thesis focuses on the derivation of SPMD programs, especially C+MPI programs, as discussed above. However, in principle, if the methodology were equipped with the appropriate APMs and corresponding transformation rules and lemmas, it could be used for a much wider range of targets. In particular, the SIMD, MIMD and SPMD programming models all fit with the concept of APMs. Different kinds of programming language style, such as implicit and explicit parallelism, and message-passing and shared-memory style, are also expressible. Outlines of APMs for a variety of models are given in the literature—O'Donnell et al give the outline of a PRAM APM in [ORR01], and [OR00] includes APMs for a MIMD machine, a specialised scan machine, and a digital circuit. These machines are very different from each other, and illustrate the power of the APM methodology and ParOp framework.

Nevertheless there are similarities between the targets, and it is therefore possible to use the abstract versions of a program for more than one target, so that the derivation paths for these targets share several initial stages.

1.3.5 Practicality

If all of the steps in the methodology are carried out by hand, the derivation process can be complicated, difficult and time-consuming. It may seem that writing parallel programs has been made more difficult instead of easier. However, this is not the only way to use the methodology. It is possible to omit several of the transformations and stages and skip fiddly details. Alternatively many of the stages could be automated or given a high level of tool support, as discussed in Section 3.2.3.

1.4 Thesis structure and organisation

1.4.1 Lemmas and transformation rules - important note

Many lemmas and transformation rules for the methodology are given in this thesis. They are collected together in Appendix A where they are categorised by topic and numbered consecutively. When they are given or referred to in the body of the thesis, these numbers are used. However it is often necessary to introduce the lemmas and rules in the main part of the thesis in a different order to that in the appendix. Therefore they often appear with non-consecutive numbers in the thesis body.

1.4.2 Thesis structure

The remainder of this thesis examines various issues in greater depth, presents the details of the APM methodology, and illustrates these with case studies.

The next three chapters give the details of the APM methodology. In particular, **Chapter 2** describes the structure of the methodology, focusing on the key stages in the derivation of a C+MPI program. The chapter shows how parallelism is represented and modelled at each stage, and describes the parallel functions in each APM and how they can be implemented and applied. In addition to the basic stages necessary for a C+MPI program, stages supporting data distributions are presented. **Chapter 3** then focuses on the transformations within and between these stages. Lemmas and transformation rules are given and discussed, as well as their derivation, proofs and application. **Chapter 4** describes how design decisions can be made and incorporated in the context of the methodology. It illustrates the general guidelines with three examples of decisions.

This is followed by three chapters that present case studies in the methodology. These illustrate and expand on the information in the previous chapters. **Chapter 5** presents a simple case study which is carried out in detail from its mathematical specification to its implementation in C+MPI. **Chapter 6** looks at one particular APM function, and takes it through its stages from specification to implementation. This involves the choice and incorporation of data distributions and detailed communication optimisations. This function is then used in **Chapter 7** which presents the derivation of a Gaussian Elimination program. This is an example of a larger derivation, and shows how the necessary extra details and problems can be dealt with.

Chapter 8 then examines various issues relevant to the work in this thesis, and discusses how previous work tackles these issues and is related to this thesis.

Finally **Chapter 9** concludes and describes some possibilities for future work.

Chapter 2

The Stages of the Methodology

2.1 Introduction

An important part of the methodology presented in this thesis is the sequence of stages through which a derivation passes, together with the APMs (Abstract Parallel Machines) or models of parallelism used at each stage.

Therefore this chapter discusses each of the basic stages and APMs of the methodology, explaining what they do, how they work and how they can be used. Not all of these stages are relevant to all programs or to all target languages. In particular, this thesis and therefore this chapter focus on the target language C+MPI, and some of the later stages are specific to a message-passing style of programming or to C+MPI itself. Most of the stages have been chosen to be applicable to a wide range of programs of this type.

Another important part of the methodology is the transformations within and between the stages presented here. These are given and discussed in the following chapter (Chapter 3).

2.1.1 Layout of this chapter

This chapter starts by examining some of the issues relevant to all of the stages and to the derivation as a whole. Then, in Sections 2.4 to 2.11, it examines some of the particular stages in the derivation of a C+MPI program, in order of decreasing abstraction. Sections 2.7 and 2.12 then examine the changes that using explicit data distributions make to some of these stages. In particular, Section 2.7 deals with the introduction of data distributions and their incorporation into the more abstract stages, while Section 2.12 presents the changes to the monadic stages. Finally, Section 2.13 summarises.

2.2 Layout of the stages

2.2.1 General layout

The stages in the methodology can be divided into three main types, specification, intermediate and target stages, as shown in Figure 2.1. The rectangles represent types of stages in the methodology and the arrows between them show their arrangement in a derivation. Specification stages come first, followed by intermediate stages, where details of the problem and target are introduced, and finally the target stages where extra language-specific details are added, and the program is finally transformed into the target language itself.

As a derivation progresses, the stages become more and more concrete. This is especially true of the intermediate stages, in which parallel details are introduced, and the program is transformed

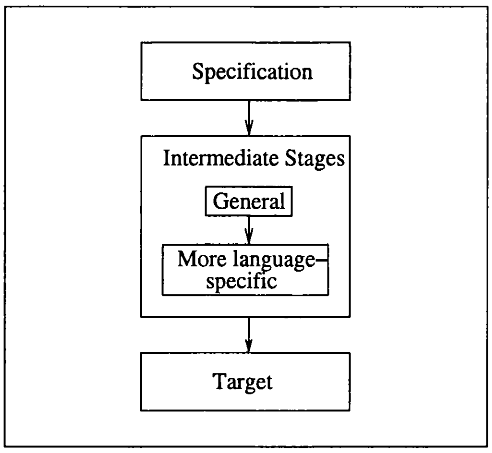


Figure 2.1: Key types of stages

from a general form appropriate for a wide range of targets to one geared towards a specific target or small range of target languages.

2.2.2 Layout of the individual stages

Figure 2.1 shows the types of the stages in the methodology. Individual stages fall into these three categories, and are related in a similar way. However their layout is not so linear since there are many possible derivations, not all of which use the same set of stages. The derivation is structured like a tree, branching at each stage, depending on the next step.

These stages are related to the APMs (Abstract Parallel Machines), described in Chapter 1. Each stage has one or more APMs connected to it which provide the model of the parallel system and the parallel operations used in that stage's programs. Basically each stage corresponds to a single APM, but in order to allow different types of parallelism to be expressed in a single program, more than one APM is allowed.

Figure 2.2 shows the structure of one possible set of APMs, presented in this chapter, along with the type of stage at which each APM is used. The arrows show the relationships between APMs, and indicate whether functions from one may be transformed into functions from another. Neither all possible APMs nor all possible connections between APMs are given.

The specification stages, which are discussed further in Section 2.4, are split into several parts. The specification is given initially in mathematics, first abstractly, then with an algorithm or method. It is then transformed into Haskell.

The intermediate stages are described in Sections 2.5 to 2.8 and Section 2.12.1. Due to time and space constraints, only a selection of the possible intermediate stages are presented, and in practice there can be many more, modelling different aspects of a program and targeting different models and languages. Monads are used in the later intermediate stages in this chapter, because they model imperative features of C, such as state and IO. For other targets, they may not be necessary.

This thesis uses C+MPI as a target language, and so the target stages, described in Sections 2.9 to 2.12, are geared towards this language. Again, in general, many other languages are possible. There is more than one target stage, modelling C+MPI with differing levels of detail. For example, the basic MPI APM views the whole system at once, on the collective level, while C+MPI uses an individual level viewpoint. Therefore there is an extra individual level stage at the end of the derivation. Such a stage is only necessary for languages with an individual level viewpoint. More on this can be found

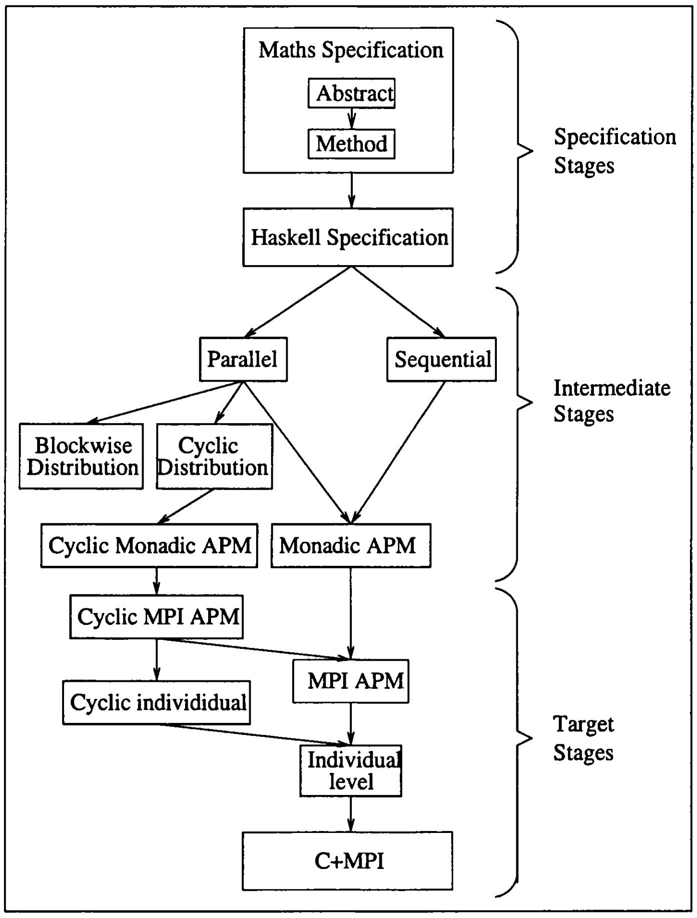


Figure 2.2: The stages in this chapter

in Section 2.11.

These stages are presented and discussed in depth later in this chapter, and the transformations within and between them are presented in the next chapter (Chapter 3). An example of a complete transformation, together with the actual sequence of steps performed is given in Figure 5.6 in Chapter 5, and provides an interesting comparison.

2.2.3 Other comments

There is a fair degree of flexibility in the methodology and its layout. For example, programs can use functions from more than one APM. This allows different parts of a program to be transformed at different times. This is useful, for example, when the same optimisation can be applied to different parts of a program but there is, as yet, only enough information to carry it out on one part, and not on another. Nested data may lead to a similar situation when the parallelism of outer layers of data is known, but that of inner layers is not yet decided.

It is also not necessary to carry out the operations in a fixed order. There are a variety of intermediate stages given in this chapter, and, although they are presented in a fixed order, they don't have to be carried out in this way. For example, in some cases it may be advantageous to specify the data distributions before introducing monads, whereas, in other cases, it is better to do it the other way round. The basic structure of the methodology is general enough to allow this flexibility.

Multiple target languages and models are also possible (see Section 1.3.4). These vary widely, and it is useful to model their characteristics in the final few Haskell stages. An example of this is the individual level stage in the C+MPI derivation. Such extra stages can simplify the final conversion to the target language. It is usually best to keep this conversion as simple as possible because it's the only step in the derivation which cannot be proven. Therefore as much detail as possible is put into the previous (Haskell) stage.

2.3 APMs in general

As indicated above, different stages in the methodology correspond to different APMs. These provide the model of the parallel system and the parallel operations used within the main program at that stage [OR97].

2.3.1 Advantages of APMs

The use of APMs brings several advantages, including the following:

- The APMs separate parallel and algorithmic aspects by restricting the parallelism to APM functions. This simplifies the programs and gives them a higher level of abstraction. The parallel and algorithmic aspects can also now be dealt with separately.

- APMs structure the programs and the derivations by associating a particular model of parallelism with each part of a derivation.

- They make it easier to add in and change parallel details because of the clear relationships between different APMs and their functions. A function or set of functions from one APM can transform into a fixed combination of functions from another.

- They also allow us to include extra information about functions formally and concisely. For example, it is important to know whether a function is to be executed in parallel or not. This information can be given by using an APM whose functions are to be executed in parallel, and another whose functions are all sequential. These APMs are given in Section 2.6.

2.3.2 The use of multiple APMs within a single stage

The original concept of APMs, presented in [OR97] and [OR00], linked a single APM with each stage in the methodology. Each version of the program used parallel functions from its associated APM. In contrast, this thesis allows each version of a program to use functions from multiple APMs. This was done for two main reasons.

Firstly, it simplifies vertical transformations. When only one APM is linked with each version of a program, a vertical transformation from one APM to another must convert the whole program in a single step. No functions from the old APM can remain in the new version of the program. The resulting transformations can be complicated and difficult. It is simpler and easier if they can be broken down into several smaller steps. To do this, the program must pass through intermediate versions involving functions from both APMs. There are two main ways of doing this. The first is to create a new intermediate APM between each pair of APMs. This new APM would provide all the operations of the pair. Another way is to allow operations from different APMs to be used within a single program. The latter option was chosen in this thesis because it allows the originating APM of each function to be easily identified. It also gives greater flexibility as it can be easily extended to allow functions from more than two APMs at once. This allows more gradual transformations.

The second reason concerns the nature of an APM. An APM corresponds to a model of the parallel system, but there are different levels of complexity possible in such a model. It is common for the model to refer to processors and communications, but it is often useful for it to provide more abstract information, for example, about the distribution of data used between these processors. This gives the programmer more information about parallelism without having to give details of data placement and communication within the program. This thesis therefore takes this view of APMs. It provides, for example, different APMs for different data distributions, even though they all use the same model of the processors and communications. However a single program may involve different data distributions or different values of such abstract information. Therefore taking this view of APMs necessitated that multiple APMs be used within a single version of the program.

2.3.3 Implementation of APMs

Since an *APM* is a collection of functions that operate on a model of a parallel system, it can be implemented as a module which exports the operations provided by the APM. The operations themselves are simply Haskell functions.

Each APM has one model of the parallel system on which its functions work. This model must be sufficiently detailed to allow the characteristics of the APM to be represented, and therefore is very closely connected to the nature of the APM itself.

Such a model can be implemented in Haskell using data structures. It must contain some items that represent sites, which are processors, threads, or locations of potential parallelism depending on the level of abstraction of the APM. They are implemented in this thesis as single data values or tuples of values, and the whole system as a sequence (or list) of these items. However, this is only one of many possible ways of modelling the parallel systems in Haskell.

The APM functions can then operate on this model. Some operate monadically, updating the system representation, while others take one picture of the system and return a new and separate one.

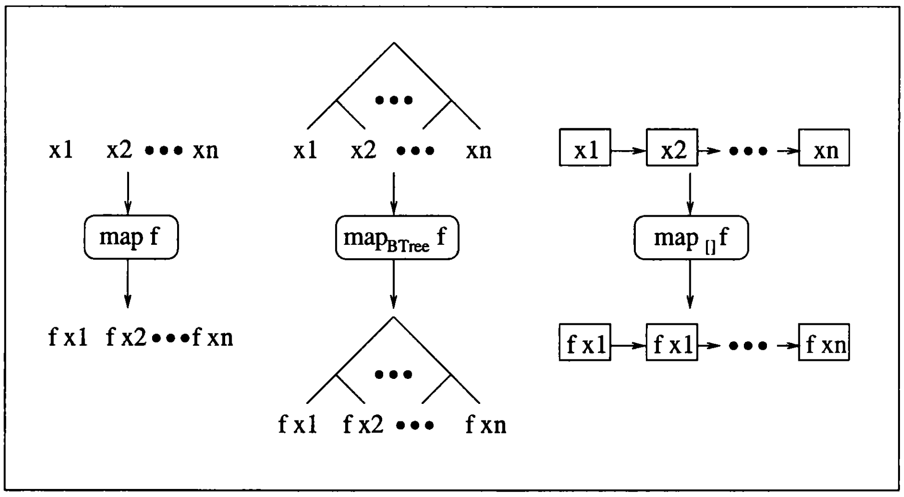


Figure 2.3: Different versions of map

2.3.4 Different versions of the same function

Different APMs often provide functions with the same name, and more than one APM may be used in a single program during a derivation. However the functions from these APMs may have differing operational behaviours. Some method is needed to tell them apart.

This thesis does this by adding a subscript to each function, indicating the function's APM. It may be the name of the APM, an abbreviation of this, or some other easily identifiable tag. In this thesis, functions from all APMs except the most abstract are annotated in this way. Functions from the most abstract APM are left unannotated, but since this is the only APM with this property, the functions can still be easily identified. In addition, in order to simplify the presentation in this thesis, annotations may dropped if it is clear which version of the function is being used.

For example, the function *map* occurs in many APMs. It has, among others, a version which operates on binary trees of type *BTreeα*, and one which operates on lists, [α]. The former is written map_{BTree} and the latter $map_{[]}$. Both of these have the same semantics as the standard *map* but on their own datatype (see Figure 2.3). The structure of the data operated on and returned varies with the particular version of *map* which is used, as does the time complexity of the operation. $map_{[]}$ on a linked list is sequential, while map_{BTree} may be done in parallel. *map* with no annotation is the most abstract function that applies its first argument to each element of its second. It is shown on the left in Figure 2.3. No operational details are specified and a type is not given. A type *is* used for the implementation of the function, but this is only for the implementation and does not represent anything. Both the structure of the data and the time complexity of the operation are left unspecified.

2.3.5 ParOps

As explained in Section 1.1.3, the ParOp form is a set form which can express the parallelism in a program. It is especially useful for specifying the parallel details of APM functions. However, the APM functions are given in this thesis as Haskell functions, and ParOps don't seem to appear.

Actually, ParOps are still being used in the thesis, even though they are hidden. A Haskell definition of an APM function is equivalent to its ParOp definition, if the syntactic sugar is removed and the function tidied up. As mentioned in [OR00], a ParOp function can be implemented in Haskell if some restrictions on its form are accepted, such as restricting the type and number of values that can be stored in the sites. The APM's functional model of the parallel system can be used as the implementation's basis. The f_i and g_i functions that parametrise the ParOp form can then be inlined, and equational reasoning applied to the result to give an ordinary Haskell definition of the function.

Example For example, take the ParOp of map_P (the parallel version of map). This function applies a function, h, to the values in each processor. Its f_i and g_i functions are as follows (the details are not important):

$$\begin{aligned} & f_i\,(\sigma_i,\ g_i(V)) = (h\,\sigma_i,\ ()) \\ & g_i\ V = () \end{aligned}$$

These are particularly simple because there is no communication, and the only thing processor i does is apply h to its value.

This can be written in pseudo-Haskell as follows. This is made simpler because there is no input to or output from the system as a whole.

$$\begin{aligned} & \mathsf{ParOp}\ \mathit{old\_state} = \mathit{new\_state} \\ & \textbf{where}\ (\mathit{new\_state}!!(i-1), A_i) = f_i(\mathit{old\_state}!!(i-1),\ g_i(V)) \\ & \qquad V = (A_1, \ldots, A_n) \end{aligned}$$

Inlining the f_i and g_i functions gives:

$$\begin{aligned} & \mathsf{ParOp}\ \mathit{old\_state} = \mathit{new\_state} \\ & \textbf{where}\ (\mathit{new\_state}!!(i-1), A_i) = (h\ \mathit{old\_state}!!(i-1),\ ()) \\ & \qquad V = (A_1, \ldots, A_n) \end{aligned}$$

which is equivalent to

$$\mathit{new\_state}!!(i-1) = h\ \mathit{old\_state}!!(i-1)$$

If this is written in actual Haskell, using, for example, a list comprehension, it can be proved equivalent to the ordinary definition of map_P.

Use of ParOps

Using ParOps, we can therefore produce Haskell definitions of APM functions, using a standard translation of the ParOp format into Haskell.

However they are designed mostly as a specification format, since they are capable of expressing the operational semantics of a function. Although the Haskell definition is so closely linked to the ParOp definition, it is sometimes not so clear about operational details. It may incorporate assumptions and implicit information about such things as the representation of parallel sites by data structures, and may obscure the function's communication. These assumptions have to be made explicit before the reader can understand the operational semantics.

The ParOps can therefore be used to make things clearer, e.g., to make the implicit assumptions explicit, although it may be more useful for the reader if such assumptions are also stated in words.

2.3.6 The use of parallel Haskell

As described above, the stages are all implemented in sequential Haskell—the parallelism is merely modelled rather than actually executed. This has advantages for simplicity and prototyping. Programs can be developed and tested quickly and without access to a parallel machine. However it also has disadvantages. Although intermediate programs can be run, and their correctness can be checked, their parallel operational behaviour cannot really be observed, although it can be reasoned about and simulated to a limited extent. This is a drawback because of the importance of this behaviour in a parallel program. Many of the transformations, such as load balancing, deal mainly or solely with such behaviour. It would therefore be useful if these intermediate stages could actually be executed in parallel.

This can be done using GpH [THLPJ98], a parallel version of Haskell, for the implementation of the APMs. GpH is described in more detail on the following page. It is possible to run GpH both on a real machine using GUM ([THMJP96]) or to use the GranSim simulator ([Loi96]) to simulate its behaviour accurately on a variety of machines with different speeds and characteristics. It also comes with a set of *profiling tools* which allow one to visualise different aspects of the processors' behaviour and load [Loi96], and thus observe the operational behaviour of a program. These facilities are employed in this thesis in Section 5.4 to visualise the effect of a decision and to see if it has the expected effect.

GpH is most suitable for use at fairly abstract levels when basic parallelism (what is and isn't executed in parallel) has been specified but not more concrete parallel details. In the more abstract levels when parallelism hasn't yet been specified, GpH can still be useful. It can help to investigate initial parallelism and data dependencies to help to determine which things it's best to make parallel and which sequential. In these cases, its simulator, GranSim, can model an idealised machine. However at these levels, GpH often complicates the code. At more concrete levels, it isn't always suitable either because its model of parallelism doesn't completely match with that provided by some of the target languages. In particular, it is not location-aware—it does not specify which processor has which data or does which task. C+MPI, on the other hand, *is* location-aware, and several parallel techniques, such as data distribution and explicit static load balancing, depend on location awareness.

Therefore it is useful to be able to use both GpH and sequential Haskell. This can be done by providing two versions of an APM, one implemented in each language. As both languages are versions of Haskell, functions from both APMs can be used in the main programs without difficulties. The sequential Haskell version can be used the majority of the time, and the GpH version when the programmer wants to check operational behaviour. An example of this is given in Section 5.5.

It can also be used later on to check operational behaviour, or to help the programmer to make difficult choices. Sometimes it is difficult to make such choices using algebra, and it can be helpful to run some of the options to see what happens. This method is not used in this thesis, but GpH *is* used to check the result of a decision made using algebra and cost models (Section 5.5). This is not necessary but provides reassurance about the choice.

However, due to the characteristics of GpH, such as its location independence, the parallel behaviour in such situations may not be a completely accurate representation. Nevertheless it can still provide valuable information. For example, static load balancing is location aware—tasks are explicitly moved from particular processors to other specified ones. The location of the tasks cannot be observed in a GpH implementation, but the total load on the system can be. This situation is examined in a particular case in Section 5.5.

Naturally, GpH is of greater use when targeting a language which is similar to it than when targeting languages such as C+MPI. It is also possible to use GpH as the target language itself. The effects of parallel transformations on the programs could then be seen immediately. An example of such a transformation can be found in [LTB01].

Details of GpH

GpH(Glasgow Parallel Haskell) [THLPJ98] is an extension of GHC [GHC] (an implementation of Haskell [Bir98, PJea99]) in which the parallel annotations, 'par' and 'seq', advise the compiler as to what can and can't be done in parallel. Expressions annotated with 'seq' have to be evaluated sequentially, one after the other. This is often used to force evaluation, ensuring the right amount of strictness for parallelism. In contrast, those annotated with 'par' don't *have* to be evaluated in parallel. The annotation just indicates that this is a possibility.

For example, in the expression, a *'par'* b, a spark for a is generated, and then the current processor continues by evaluating b. The spark indicates that a could be evaluated in parallel, and when there is a free processor, it can pick up the spark for a, turn it into a thread and start to evaluate it. If b requires the value of a, it waits until a is finished evaluating and then gets its value. If, however, b needs the value of a *before* a is turned into a thread, then the current processor just starts to evaluate a and a's spark is abandoned.

Programming in GpH can be a bit fiddly, as the parallel details are mixed up with the basic algorithm. However a method called strategies can be used to increase abstraction. A *strategy* is a function which controls the dynamic behaviour of an expression (e.g., its evaluation degree or the extent of its parallelism). It is also written in GpH, and is attached to the corresponding expression as an annotation, with the Haskell function 'using'. The main expression then need not contain the parallel details.

Example For example, the quicksort function,

```
quicksort [] = []
quicksort (x : xs) = lowsort ++ [x] ++ highsort
  where
    lowsort  = quicksort (filter (< x) xs)
    highsort = quicksort (filter (≥ x) xs)
```

can be parallelised by introducing 'par's and 'seq's into the code directly or by using a strategy in the last pattern:

$$quicksort(x : xs) = lowsort \mathbin{+\!\!+} [x] \mathbin{+\!\!+} highsort \text{ `using' } strategy$$

This has the advantages that the main code need not be changed and that the parallel behaviour can be changed simply by replacing *strategy* with a different strategy.

An example strategy which might be used with this example is

$$parstrategy\ result = lowsort \text{ `par' } highsort \text{ `seq' } result \text{ `seq' } ().$$

This says that the two main parts of the calculation can be done in parallel.

In GpH parallelism must be identified by the programmer. Therefore the programmer can control features such as the size of the tasks which are sparked off in parallel (granularity) and which pieces of data are operated on in the same processor (data clustering). However the programmer has no say in schedule or allocation—there is no way to specify which processor should execute which piece of code. The decisions of data and task placement, load balancing, etc. are up to the run-time system.

GpH uses a dynamic load balancing strategy: when a processor is idle, and has no work in its local spark pool, it tries to get a spark or a thread from other processors. Thread migration and other such communication strategies can be turned on or off by the programmer as a run-time system (RTS) option.

2.4 Specification stages

At the beginning of a derivation the problem to be solved must be specified. This specification takes up the initial few stages of the methodology. In principle, any programming problem can be specified, but this thesis focuses on mathematical and scientific problems.

It is important that the initial specification is correct and clear. An incorrect specification will lead to an incorrect program which will not give the right results. An unclear specification may lead to a program which does different things from those originally intended, thus not giving any useful results.

For example, the specification might say "the program should sort a sequence of integers", but not say what ordering should be used. Both the specifier and the programmer may think they know which ordering is meant, but they might have different ideas about it. For example, the original intention may be decreasing order, but the programmer assumes increasing order. This leads to a program which produces a different list from that intended (see Figure 2.4). This is, of course, rather an extreme case in which a large part of the problem is left unspecified. Usually it is smaller, but still significant, details which are ambiguous. These still lead to incorrect or useless programs.

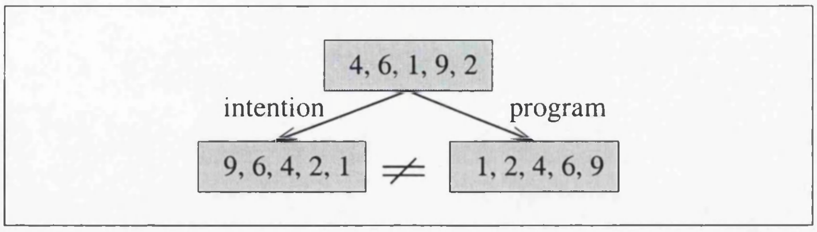


Figure 2.4: An unclear specification of a sort

Therefore it is important to specify the problem clearly and without ambiguity. This doesn't mean that there must only be one possible solution or one way of obtaining a solution, but that, given a set of inputs, it is clear what possible value or values the output can have. For example, $x^2 + 2x + 8 = 0$ specifies the possible values of x clearly, but there is more than one value of x which solves it and more than one way of finding these values.

A natural language specification does not usually fulfil this role because a natural language such as English is full of ambiguities and multiple meanings. For example, consider the sentence earlier in this section, "the program should sort a set of integers". The problem above arose because of the ambiguity of the word "sort". This and other problems are common in natural language [Inc88], and hence a more precise method of writing the specifications is needed.

There are several possible methods which could be used. Mathematics, pseudo-code, an imperative or declarative programming language, and the ParOps mentioned earlier in this chapter are just some of the possibilities. These have varying levels of abstraction and explicitness about parallel and other details. In the APM methodology, mathematics is used initially because of its high level of abstraction and its wide-spread use.

2.4.1 Mathematical specification

Mathematics is often used as a specification method (see, for example, [Inc88]). It is clear, precise and unambiguous. It can also be abstract, unlike many programming languages, avoiding details, such how a solution is obtained, although these can be given if necessary. It can be used to specify only *what* must be done, without saying *how*.

For example, the above specification, "the program should sort a sequence of integers" can be written using mathematics, without any information as to how to do it, as:

Given a sequence, $< s_1, \ldots, s_n >$, of integers,
find a permutation $< t_1, \ldots, t_n >$ of $< 1, \ldots, n >$ such that
$s_{t_i} \leq s_{t_j}$ for $1 \leq i \leq j \leq n$.

It can also avoid operational details, such as locations of values and communication, which are given in other forms such as ParOps. It can be useful to be able to specify such things, but that doesn't belong in the most abstract stage of the methodology.

The things which it must specify are the essentials, such as the result value or values in terms of the input values and the range of input values for which this must hold.

Algorithm specification

However, even though these details don't have to be included, it can be helpful to include some of them. A totally abstract specification is a good starting point, as it allows multiple implementations, but it often cannot be easily transformed into a programming language such as Haskell. The specification can therefore be refined, still within the mathematical framework, to give some indication of how to solve it, such as an algorithm.

For example, the abstract specification above says clearly what the resultant set should look like, but it doesn't give the programmer much help with actually calculating this set. There are many sorting algorithms which can actually be used (see, for example, [CLR90]) and one of these can be chosen and given as a refinement of the specification. Later transformations in the derivation can also modify and improve this algorithm.

For example, here is one algorithmic specification for the problem, using the quicksort algorithm, which is given in many textbooks including [BW88].

Given a sequence, $< s_1, \ldots, s_n >$, of integers, partition the sequence of indices, $< 1, \ldots, n >$ into three sequences:
$T_1 =< i | s_i < s_1 >$
$T_2 =< i | s_i = s_1 >$
$T_3 =< i | s_i > s_1 >$
If T_1 or T_3 have more than one element, then sort them using this same algorithm to give T_1' and T_3'. The result $< t_1, \ldots, t_n > = T_1' +\!\!+ T_2 +\!\!+ T_3'$.

In this case there are two separate specifications, an abstract one and an algorithmic one. Both of these can be given within the methodology, but the programmer has the responsibility of proving that the latter specification fulfils the first.

Limits imposed by the mathematical notation

Despite the power and advantages of maths notation, it has its limits. For example, it cannot express subjective qualities. One can express in mathematics the structural properties required of a bridge, but not that it should be aesthetically pleasing.

It is important to be aware of such limitations so that undue time is not spent trying to formulate specifications in mathematics that cannot be written in this form. The APM methodology does not deal with such specifications. If desired, however, such specifications can sometimes be reconsidered and changed so that they *can* be written in mathematical notation.

These limitations are acceptable because of the connection between mathematics and programming languages. A specification which cannot be written in maths cannot be written in a programming language either, at least at present. Mathematics is at least as expressive as programming languages.

Limits imposed by the target language

In fact, mathematics is more expressive than most programming languages. By using the full power of mathematics, problems are allowed that can be programmed in only a few or not even any programming languages. This is done to allow the derivation of programs for a wide range of languages and of the full range of programs in those languages. However, since Haskell is used in the majority of derivation stages, some restrictions are introduced when Haskell is introduced (see Section 2.4.2). But even Haskell contains many features not present in some of the target languages.

Therefore, there are problem specifications that cannot be solved in the target language. An example of this occurs with a target language such as C which does not have polymorphism—functions have to operate on data of specified types, not on general data. But a program specification can say that it should work for multiple types, e.g., any numerical type.

In such a case, we could say that the specification is not solvable in the target language. This is strictly true, but not very helpful to the programmer. He or she may prefer to have a program which only works on *Integer*s rather than no program at all.

The methodology deals with this as follows: As well as introducing details, the derivation tree can narrow down the specification, for example, by only insisting that it works for certain types of values, or for values with a certain property. The resulting program will then fulfil this special case of the specification rather than the specification itself. It is important to keep note of these modifications to the specification so that the user doesn't think that the program fulfils the initial specification, and try to use the program for data for which it does not work.

The map-triangle case study in Chapter 5 contains two examples of this. In it, the specification is narrowed twice, first in Section 5.3 and then in Section 5.7.

2.4.2 Haskell specification

While mathematics is very useful for the specification, it is not best suited for the intermediate stages in the methodology. Haskell is chosen instead for the reasons given in Section 1.3.3. In order to use Haskell for these stages, the initial specification must first be written in Haskell.

Difficulties

However, although functional languages such as Haskell are closely related to mathematics, the transformation from one to the other is not trivial. This is because there are some substantive differences between mathematics and Haskell in terms of their expressiveness. Some of these are discussed below.

Relations Haskell is a functional language and is therefore based on functions, which map values in the domain to single values in the codomain. However, mathematical specifications are often relational, mapping values in the domain to possibly many values in the codomain. A relational language [CBS97], such as Libra [Dwy95] could therefore be used here with greater expressiveness. Occasionally this power is very useful, but usually it is not necessary because most programming languages are based around functions.

Relational specifications are therefore usually only used in the methodology to specify sets of acceptable results from the program, i.e., to express nondeterministic results. The Haskell specification can, however, express this using methods such as nondeterministic sets [HO89, GO99]. Alternatively the specification may be refined at this point to specify which of the alternative values will be produced. The resultant programs will still satisfy the specification, but this will rule out several possible solution programs.

Algorithmic details If a maths specification is given only in its abstract form, it can be very hard or even impossible to transform it into Haskell. As explained in Section 2.4.1 above, this can be overcome by giving a more concrete maths specification involving algorithmic details.

Another solution is to allow algorithmic details in the implementation of functions in Haskell, but to hide them from the main program using the module system. This way the functions seem abstract to the programmer and any transformations of them depend only on their specified properties not on their implementation.

Types Although Haskell has a powerful type system, including, among other features, polymorphism and type classes, the type system is limited and does not provide totally general types and type manipulation mechanisms. For example, existential types are missing. There are extensions available to Haskell and also research going on all the time, but current standard versions of Haskell [PJea99] are limited. Therefore, in order to transform a mathematical specification into Haskell, it may sometimes be necessary to either restrict the specification or simulate the type features using other features available.

Despite these reasons, Haskell is used in the methodology. The problems can be overcome as discussed above, and I feel that the advantages of Haskell, as listed in Section 1.3.3 outweigh the disadvantages.

2.4.3 An APM for the specification level

The Haskell specification is a stage in the derivation and therefore needs an APM (see Section 2.2). However at this stage it is undesirable to specify any operational details, such as whether an operation is parallel or sequential. Giving such details at this stage will restrict the choices available further on and thus may lead to sub-optimal solutions. Therefore the APM should be as abstract as possible.

In general, APM functions at this level should say what the final results are given the initial values, but not anything about the distribution of the data or what is parallel and what sequential. Their implementations may assume some of these things, such as a particular distribution of the data, but these are hidden from the programmer.

Finite sequences

In common with other stages in this thesis, the Haskell specification uses finite sequences as the basic datatypes for representing ordered data such as matrices and arrays. Other datatypes can be used for other types of data, such as unordered data (e.g., sets) and structured data (e.g., graphs). The user can also use other datatypes or create his own. However, for the purposes of this thesis, it suffices to focus on ordered data represented by finite sequences.

An abstract *finite sequence* (abbreviated to *FinSeq*) is a finite ordered sequence of values of the same type, such as finite lists and arrays. It is an abstract data type and says nothing about the storage or location of the data. Such information belongs later in the methodology.

The implementation of finite sequences is hidden from the user, and properties of the implementation should not be confused with properties of the sequences themselves. They can be implemented in various ways, for example using arrays or specialised structures, but standard Haskell lists have been used in this thesis for ease of implementation. Lists are well-supported in Haskell and many of the desired higher-order functions are already provided on them. List manipulation functions can also be used to construct and manipulate sequences.

Function	Operation
map	Applies a function to every element of a sequence.
fold	Sums the elements of a sequence using a given function.
fold variants (e.g., *foldr*, *foldl1*)	Similar to *fold* but may have specified evaluation orders and starting values.
scan and variants	Return the partial sums of a sequence.
accum_scan and variants	Versions of *scan* in which one result depends on all previously generated results.
filter	Returns those elements of a sequence satisfying a given property.
take	Returns the specified number of elements of a sequence, starting from the front of the sequence.
reverse	Reverses a sequence.
special_sum	An example of a user-defined function. Returns a sequence in which each location contains the sum of its neighbours' values in the original sequence.

Table 2.1: Outline of the specification APM

The APM for the specification level

The APM is based on finite sequences, and contains a number of functions which can be added to. These functions specify the final result given the initial values, but their implementations are hidden from the user.

For example, *map* can be specified as follows, using list notation for the sequences, and ... to denote intermediate elements in the sequence:

$$\begin{aligned}&map :: (\alpha \rightarrow \beta) \rightarrow FinSeq\,\alpha \rightarrow FinSeq\,\beta\\ &map\,f\,[x_1,\,\ldots,\,x_n] = [f\,x_1,\,\ldots,\,f\,x_n]\end{aligned}$$

This indicates that *map* applies f to each element in the sequence but not how the data is stored or the order in which f is applied to the elements.

The APM is outlined below and in Table 2.1 with some examples of the type of functions which are in it. It is outlined and not given fully because we expect and encourage the programmer to expand on it in order to tailor it for her particular applications and problems. It is not fixed, and is meant to be flexible. The examples should aid the programmer in writing her own functions.

APM function examples

This section presents some further examples of functions from the specification APM with their specifications and English descriptions. The first two are standard Haskell higher-order functions, and their implementations can be found in the standard Prelude. The third function, *accum_scanr*1, was specially written. It is discussed in detail in Chapter 6 and its implementation can be found in Section 6.2.2. Finally, *special_sum* is an example of a user-defined function and its implementation is given here.

foldl:

$$\begin{aligned}&foldl :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow FinSeq\,\alpha \rightarrow \alpha\\ &foldl\,(\oplus)\,a\,[x_1,\,x_2,\,\ldots,\,x_n] = (((a \oplus x_1) \oplus x_2) \oplus \ldots) \oplus x_n\end{aligned}$$

foldl sums the elements of a sequence from its left-hand end, starting the calculation with a and using $\oplus$. *foldl* and *foldr* (which sums the elements from the right) are included in the APM, not just *fold*, because *fold* does not specify an evaluation order, leading to nondeterministic values when $\oplus$ is not associative.

foldl and *foldr* don't actually have to be evaluated in the specified order, as long as the specified result is obtained. In particular, if $\oplus$ is associative, then they can be evaluated in a parallel fashion.

take:

$$\begin{aligned}&\mathit{take} :: \mathit{Int} \to \mathit{FinSeq}\,\alpha \to \mathit{FinSeq}\,\alpha\\&\mathit{take}\,m\,[x_1,\ \ldots,\ x_n] = [x_1,\ \ldots,\ x_{\min\{m,n\}}]\end{aligned}$$

take returns the first m elements of a sequence, or all the elements if the sequence is shorter than m. This may be used as a mask so that only certain parallel elements are operated on, or done sequentially to remove some values. However this version of *take* does not specify such details.

*accum_scanr*1:

$$\begin{aligned}&\mathit{accum\_scanr}1 :: (\alpha \to \mathit{FinSeq}\,\beta \to \beta) \to \mathit{FinSeq}\,\alpha \to \mathit{FinSeq}\,\beta\\&\mathit{accum\_scanr}1\,f\,[x_1,\ \ldots,\ x_n]\\&\ = [f\,x_1\,[\mathit{res}_2,\ \ldots,\ \mathit{res}_n],\ f\,x_2\,[\mathit{res}_3,\ \ldots,\ \mathit{res}_n],\ \ldots,\ f\,x_{n-1}\,[\mathit{res}_n],\ f\,x_n\,[]]\\&\ = [\mathit{res}_1,\ \ldots,\ \mathit{res}_n]\end{aligned}$$

*accum_scanr*1 is one of a family of accumulating scan functions in which the ith value of the result may depend on *all* the previously calculated values, not just the immediately previously calculated one. This particular version of the function starts its calculations at the right-hand end of the sequence, and has no initial value.

This family of functions is useful for implementing certain kinds of **for** loops as explained in Section 7.2.3. The functions can be written in terms of ordinary scans, but are defined separately so that they can be used and manipulated more easily. More information about them may be found in Chapter 6 in which *accum_scanr*1 is implemented in a variety of APMs, right down to an implementation in C+MPI.

A similar accumulating *scan* function is given in [Pep93], although its relation to *scan* and to **for** loops is not identified, the author focusing instead on a recursive definition involving *map* and on its systolic dataflow implementation.

special_sum:

$$\begin{aligned}&\mathit{special\_sum} :: (\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \mathit{FinSeq}\,\alpha \to \mathit{FinSeq}\,\alpha\\&\mathit{special\_sum}\,(\oplus)\,l\,r\,[x_1,\ldots,x_n] = [l \oplus x_2,\ x_1 \oplus x_3,\ \ldots,\ x_{n-1} \oplus r]\end{aligned}$$

special_sum returns a sequence in which the value of each location depends on its neighbours' values in the original sequence. l and r are boundary values. This expresses a pattern of computation common in numerical analysis.

This is an example of a function which the user can define and add to the APM. It can be written in Haskell using list comprehensions as follows:

$$\mathit{special\_sum}\,f\,l\,r\,\mathit{xs} = [f\,x\,y \mid x \leftarrow l : (\mathit{init}\,\mathit{xs}),\ y \leftarrow (\mathit{tail}\,\mathit{xs}) +\!\!+ [r]]$$

2.5 Intermediate stages

After the specification stages come the intermediate stages, as shown in Figures 2.1 and 2.2. They all introduce details to the program, but, within this, vary in purpose and type. Some of these stages only occur in particular derivations, while some of them occur in all. For example, it is important to specify what is parallel and what sequential in any program, while static load balancing is only of use to a limited subset of programs.

There are two main kinds of intermediate stages. At some the program is brought closer to its target language or machine. Examples include the specification of parallelism and data distributions. The transformations at these stages are known as *vertical transformations.* At other stages, such as load balancing and restrictions of the specification, the algorithm itself is modified. The transformations at these stages are called *horizontal transformations.* Both vertical and horizontal transformations are discussed in more detail in the following chapter in Sections 3.4 and 3.5.

The two different kinds of stages are characterised in the methodology by their use of APMs. When the program is brought closer to the target, the program usually uses a different APM after the transformation than before. Algorithm modifications, on the other hand, usually use the same APM. This is because algorithm modifications aren't more specific about the parallelism, they just manipulate it in different ways. However, when a program is brought closer to the target, new parallel and system details often need to be modelled. These two different types of stages correspond to two types of transformations, horizontal and vertical ones as described in Section 3.2.

Most of the intermediate stages involve making decisions. These may be about the characteristics of the target language or machine, the particular implementation to be used, or details of optimisations. The first kind are determined by the target which is often known in advance. The others usually involve efficiency considerations. How such decisions can be made in the context of the methodology is considered in Chapter 4.

The following sections (2.6 to 2.8) look in detail at a few intermediate stages which use separate APMs. An example of an algorithmic modification, load balancing, is given separately in Section 4.4.

2.6 Expressing parallelism

The specification so far is fairly abstract; it does not say anything about where the data is stored, not even which calculations are done sequentially and which in parallel. We may have an intuitive understanding of what is going on, but so far we haven't limited the program to execute in any particular fashion. Before intermediate stages such as load balancing and the specification of data distributions can be carried out, it is necessary to decide on this basic parallel behaviour. More complicated parallel behaviours are specified later in the derivation.

It is also possible to leave the parallelism of some computations unspecified, and to use the transformation to this stage again later in the derivation to specify them. This may be done multiple times. This is allowed because some transformations, such as certain types of load balancing, only require the parallelism of the outermost nesting levels to be specified. That of more deeply nested levels doesn't need to be. It may be advantageous to leave the specification as late as possible because some functions benefit from being executed in parallel only under certain restrictions. It is best to introduce these restrictions as late as possible in order to keep the top levels in the methodology general, so that the first part of the derivation can be shared among the maximum number of programs.

2.6.1 Representation of parallel behaviours

As indicated in Section 2.3.4, properties such as parallel behaviours can be indicated by annotating functions with subscripts. These act as indicators to further levels in the derivation of the role of the data and the operation of the functions, and help in future derivation (e.g., when calculating data distributions or introducing load balancing). These functions operate over data structures that are variants of the *FinSeq* data type introduced earlier.

Parallel functions have subscript P and operate on data of type *ParFinSeq*. This is a version of *FinSeq* carrying the assumption that its elements will be stored in several processors. Sequential functions have subscript S and use data of type *SeqFinSeq*. The *SeqFinSeq* type carries the assumption that its elements will all be stored in a single processor. The implementations of these functions are also affected—S functions must be implemented sequentially, while P functions may or may not be executed in parallel. This uncertainty arises because data may have to be dealt with sequentially even if it's stored in different processors due to data dependencies. This usually causes extra communication.

These data types are implemented by lists in a similar way to *FinSeq*, but again they should be viewed simply as finite sequences of data without thinking about their implementation. They are implemented sequentially using a particular data structure, but their denotational meaning carries information about their parallelism, and abstracts away from the sequentiality of lists. This denotational meaning is not given formally but is attached informally to the data type as part of the APM. The programmer should think in terms of it, although the implementations may be useful for proofs.

2.6.2 The models of the system

A parallel function uses *ParFinSeq* α as the model of the parallel system where α is the type of values in a processor. If there are multiple values of type β in a processor then these form a sequential sequence and α = *SeqFinSeq* β. Instead of individual processors, the values can be distributed over clusters of processors, each with a value in it. In this case the type becomes *ParFinSeq* (*ParFinSeq* β).

Example For example, a sequence of sequences, $[[1,2],[3,4,5,6],[7,8]]$, may be stored in parallel with one sub-sequence in each processor. $[1,2]$ could be stored in the first processor, $[3,4,5,6]$ in the second, and so on. Such a sequence would have type *ParFinSeq*(*SeqFinSeq Int*). Alternatively the sub-sequences can be distributed over the processors in a cluster, the value 1 in one processor and 2 in another. This would have type *ParFinSeq*(*ParFinSeq Int*).

2.6.3 The APMs

Together the annotated functions and the representations of the parallel system form two APMs: Parallel and Sequential APMs. These APMs are closely related to each other and to the abstract APM from Section 2.4. They provide essentially the same functions, just with different annotations. Their implementations are also very similar, and in some cases are identical.

For example, in the most abstract APM there is a function, *map*. Corresponding to this, there are functions map_P and map_S in the APMs here. A few of the functions from these APMs are given in Table 2.2 with an explanation of their meanings.

Function	Meaning
map_P	A function, f, can be applied to all the elements in the given sequence at the same time.
map_S	The function is applied to each element in turn.
$filter_P$	A sequence is filtered by examining each element to see whether it satisfies the given predicate. Each element of the sequence is examined at the same time.
$filter_S$	Each element is examined in turn.

Table 2.2: Brief outline of the parallel and sequential APMs (See Table 2.1 for general meanings of these functions.)

Functions are also provided to convert between these data formats. These correspond to data redistribution operations on the target, and hence are usually expensive functions to implement. They are:

seqtopar :: *SeqFinSeq* $\alpha \dashrightarrow$ *ParFinSeq* α and
partoseq :: *ParFinSeq* $\alpha \dashrightarrow$ *SeqFinSeq* α.

As with the abstract APM, the provided functions form the foundations of a set which the user can build up rather than a definitive set. In addition, this is not the only way of representing the state at this level, and different APMs can be produced by the programmer as necessary.

These two APMs can be used together. A program at an intermediate stage will usually contain functions from both of these APMs, since all of the functions in a program are unlikely to have the same parallel behaviour. This is a slight modification of the original idea presented in [OR97] but allows greater flexibility.

2.7 Data distributions

The specification of data distributions in a program is an intermediate stage in the methodology. The use of data distributions affects the model of the parallel system and so new APMs are used. One APM is provided for each distribution to make it easier to manipulate and reason about the distributions.

Data distributions describe the way in which the data is stored across the processors; they say which data is in which processor. It is important to know this in order to create an efficient and correct program.

Data distributions, other than the simplest naive ones, are usually used when there is more data than the number of processors. More than one piece of data must then be allocated to each processor. Particular distributions can also be used to facilitate the use of particular programming techniques (e.g., tree-like distributions are useful for divide-and-conquer methods), or to give a program a good load balance.

Data distributions can be assigned statically before run-time, or dynamically at run-time. The latter carries (sometimes substantial) runtime overheads, but can result in distributions better suited to the actual data. This thesis focuses on the former, because run-time optimisations provide less scope for the investigation of APMs at this stage, being more concerned with detailed algorithms than with the representation of parallelism.

2.7.1 Examples of data distributions

Some standard distributions

There are various standard data distributions, and it is worth looking at some of these in a little detail just now to illustrate what follows and embed it in reality. Other such distributions exist and are similar.

Figure 2.5 illustrates these distributions. The large rectangles represent the memory of the processors, and the notation, xi, represents the ith element in a sequence.

The sequential distribution, corresponding to the sequential APM introduced previously, is not really a distribution in the strictest sense of the word, as all of the data is in a single processor, rather then being distributed. However it does represent one way of arranging the data.

The naive distribution is the simplest way of distributing the data across multiple processors. It assigns one value to each processor. This assumes at least as many processors as there are pieces of data, thus limiting the number of data elements the program can deal with. However most parallel programs operate on large data sets, and therefore this distribution is often unrealistic in practice. The parallel APM can be used for this distribution.

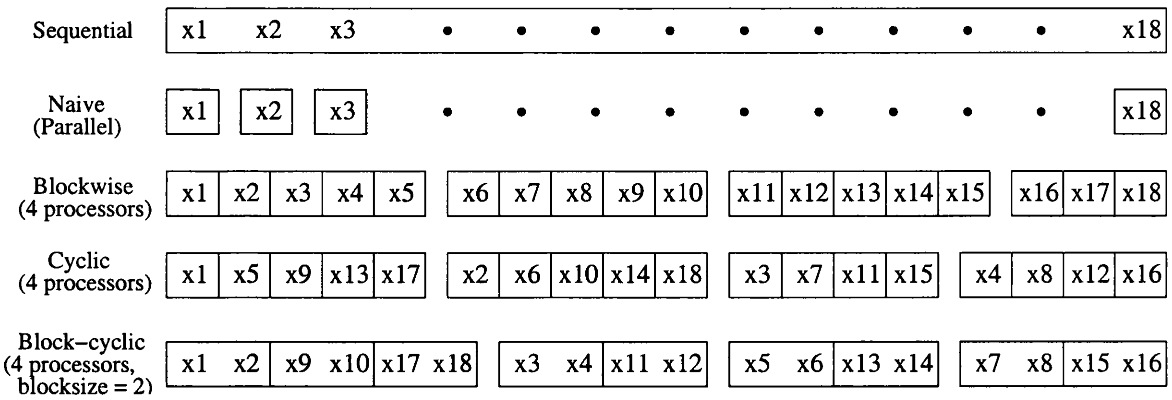

Sequential
x1 x2 x3 • • • • • • • • • x18
Naive (Parallel)
x1 x2 x3 • • • • • • • • • x18
Blockwise (4 processors)
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18
Cyclic (4 processors)
x1 x5 x9 x13 x17 x2 x6 x10 x14 x18 x3 x7 x11 x15 x4 x8 x12 x16
Block–cyclic (4 processors, blocksize = 2)
x1 x2 x9 x10 x17 x18 x3 x4 x11 x12 x5 x6 x13 x14 x7 x8 x15 x16

Figure 2.5: Some standard data distributions

The blockwise, cyclic and block-cyclic distributions, on the other hand, can handle large amounts of input on a fixed size machine, by assigning multiple values to each processor.

The blockwise data distribution assigns the data in blocks of size b, one block to each processor, until it arrives at the last processor which gets all the remaining data items. The block size, b, is usually calculated from the total size of the input data in such a way that the last processor gets less than b items. If there are n pieces of data and p processors, b is often $\lceil n/p \rceil$. In some situations consecutive data is used together, and in these, this distribution reduces communication costs by placing such data together in the same processor.

The basic cyclic distribution assigns one piece of data to each processor and when it runs out of processors it goes back to the first processor and continues assigning data as before. The values can also be indexed in this cyclic fashion, so that element $x1$ in Figure 2.5 has index 1, $x2$ 2, and so on. This cyclic indexing is used later in this chapter.

In general, cyclic distributions don't have to assign individual items. They can use blocks of a set size as in the blockwise distribution. Such as distribution is known as block-cyclic. An example with block-size 2 is shown in Figure 2.5.

As indicated in [To95] (p151), cyclic distributions often give a better load balance than blockwise ones, but may have higher communication costs. This tends to be true when the size of individual pieces of data and tasks is variable and consecutive data is not used together.

Specialised distributions

As well as the standard distributions, specialised ones can be created for particular situations. This may be to balance the load on the processors, or to keep data which is accessed together in the same processor. APMs for these distributions can be created in similar ways to those for standard ones, but the standard ones are sufficient to demonstrate the ideas.

Higher dimensional data

The distributions described above deal only with 1-dimensional data, but they can be easily extended to work with higher dimensions. A common example of such higher-dimensional data is a 2-dimensional matrix.

In such cases, there are various possibilities. Each dimension of the data can be distributed separately using a 1-dimensional distribution or some of the dimensions can be treated as single value, reducing the number of dimensions under consideration, as shown in the examples in Figure 2.6, which uses similar notation to Figure 2.5. The first example shows the original matrix, and the subsequent ones show its distribution over four processors. The rows are distributed first in a blockwise fashion, and then the columns are distributed in three different ways. Alternatively, a specialised multi-dimensional distribution can be used.

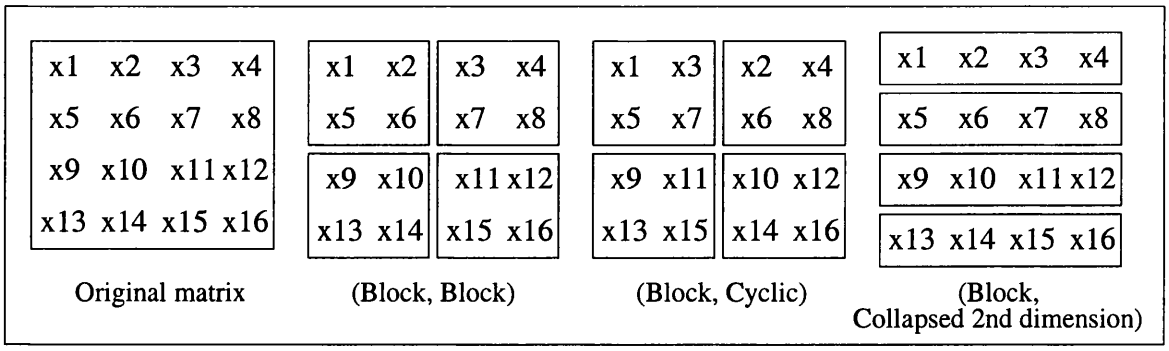


Figure 2.6: Some distributions of two-dimensional data on four processors

The target architecture also needs to be considered. Architectures such as n-dimensional hypercubes can deal directly with multidimensional data, but they are rare and expensive. 2D (and sometimes 3D) matrices of processors are more common. Higher-dimensional data distributions *can* be mapped onto such architectures, but there will no longer be direct connections between processors which are adjacent in the distribution, increasing communication time.

2.7.2 Data distributions in the methodology

It is useful to express data distributions explicitly and provide operations that manipulate explicitly distributed data. These operations can then be used without worrying about how they are implemented, and what communications they require. This is especially useful during stages which are abstract about such details.

However, many target languages, including C+MPI, don't provide data distributions—the necessary communications and other details must all be given in the program. Therefore, if such a language is targeted, we cannot leave the explicit data distribution functions in the program. Later in the methodology, as the program is brought closer to the target language, they must be replaced with combinations of ordinary functions which produce the same results.

When introducing data distributions to a program, it is not necessary to introduce them everywhere all at once. They can be given for some parts of the program and not for others. The parts for which specific distributions are not given continue to use the *ParFinSeq* APM, and so are assumed to use the generic abstract or naive distribution with one data element per processor. These parts can be refined later to use a more specific distribution.

It is also possible to use different data distributions within one program. If different distributions are given for the same piece of data at different points, then a data redistribution operation must be included between those points. Different pieces of data within the same program can also use different data distributions. Therefore there may be more than one data distribution APM associated with a program.

2.7.3 Data distribution APMs

One APM can be produced for each possible data distribution, leading to a set of possible data distribution APMs. The functions from different APMs are clearly annotated to indicate which APM they come from and which data distribution they are associated with, and the datatypes for the APMs are similarly named. Therefore it is made clear which APM applies to which part of the code.

We have already seen two of these APMs: the Parallel APM for the naive distribution and the sequential APM for the sequential distribution (Section 2.6). The other distributions' APMs provide similar operations to these, but with added detail about the parallel distribution of the data. The implementation must ensure that any properties required of the distribution used hold. For example, the cyclic APM provides operations such as map_{Cyclic} and $scan_{Cyclic}$ and indicates that the data on which these operate is distributed in a cyclic manner. The implementation must ensure that the data returned from such operations is still in a cyclic format.

The model of the parallel system also changes. Because a data distribution APM assigns more than one piece of data to each processor, modelling the parallel system requires a more complicated data type than it did previously.

The data distribution APMs are related to the previous APMs in other ways as well. The data in a distribution is usually spread over more than one processor. Therefore only functions and data previously marked as parallel (P) can be converted into distributed data.

Later in the derivation, in the cases of many languages, including C+MPI, the explicit data distribution functions and types must be removed because the target language does not support them. This is not true of all languages, e.g., HPF [MC97] provides explicit data distributions. However, in many cases, functions from the data distribution APMs must be transformed into combinations of lower-level functions that do not explicitly deal with distributions. For example, Cyclic APM functions can be transformed into Cyclic MPI APM functions and from there into combinations of ordinary MPI APM functions, as described in Section 3.10.

2.7.4 Example: Cyclic APM

The *cyclic APM* represents the basic cyclic distribution as described in Figure 2.5. It is based on the parallel APM in Section 2.6, and provides essentially the same operations as that APM, but annotated with the subscript *Cyclic*. It also uses a different model of the parallel system in order to cope with the extra complexity of multiple data elements in each processor.

This model is encapsulated in a type, $Cyclic\ \alpha$, which models the parallel system as a set of processors each of which contains multiple elements of type α, i.e., a sequence of type $SeqFinSeq\ \alpha$. Therefore the model can be implemented as follows:

type $Cyclic\ \alpha\ =\ ParFinSeq\ (SeqFinSeq\ \alpha)$

The APM functions operate as before, but on data of this type. For example, a function such as $reverse_P :: ParFinSeq\ \alpha \rightarrow ParFinSeq\ \alpha$ now becomes

$reverse_{Cyclic} :: Cyclic\ \alpha \rightarrow Cyclic\ \alpha$

with essentially the same semantics, but being careful to maintain the cyclic distribution of data. In the case of *reverse*, this means that all values must be moved, including those within each processor, and not just blocks of values.

As well as versions of functions from the *ParFinSeq* APM, the cyclic APM provides functions for converting data between the cyclic representation and other representations. These are important for expressing redistribution operations and also for constructing and manipulating cyclic data manually. Some of these are summarised in Table 2.3.

2.8 Modelling imperative features

If the target of the derivation is an imperative language, as it is in this case, there are various imperative features of that language that aren't usually present in Haskell. Introducing these to the

Function	Operation
makecyclic p xs	Converts a list, *xs*, into a Cyclic data structure over *p* processors. The elements are distributed cyclicly.
toCyclic :: *ParFinSeq*(*SeqFinSeq* α) $\rightarrow$ *Cyclic* α	Converts the representation of a Cyclic data structure into the structure itself.
fromCyclic :: *Cyclic* α $\rightarrow$ *ParFinSeq*(*SeqFinSeq* α)	Opposite of *toCyclic*.
$cons_{Cyclic}$	Adds a value onto the front of the first site in the system. Variants may redistribute the values to maintain the cyclic distribution.
cyc2block	Redistributes cyclic data in a blockwise manner over the same system.

Table 2.3: Some Cyclic APM construction functions

Haskell program prepares the way for the language-specific stages later in the derivation (see Section 2.9). They bring the Haskell program closer to the target, and allow the language to be modelled more accurately.

The most common imperative features involve side-effects such as state and IO (input/output). These are usually manipulated explicitly in an imperative program. Variables are stored explicitly in the state, although the retrieval of their values is often hidden by syntax. For example, in C an occurrence of a variable, x, represents its value retrieved from that variable. Similarly IO is usually done through specialised input and output functions, such as `printf` and `scanf`.

These can be modelled in Haskell using monads as described below. This section only deals with data using the naive distribution with one element per processor. Other distributions are considered in Section 2.12.

2.8.1 Monads

A monad is a concept from mathematics which has been adapted to express and encapsulate side-effects, such as state and IO, in pure functional programming languages, noticeably Haskell [Wad92]. It has the big advantage of being sound under equational reasoning, and thus monads can be used within the APM methodology and still allow correctness-preserving program transformations.

This soundness is achieved because monadic programs, in themselves, are simply Haskell programs. They only model side-effects - these side-effects are not actually carried out. However, Haskell compilers are free to implement the programs in any way that is "safe". Safe implementations are possible because the monadic way of modelling side-effects restricts the ways in which the state can be manipulated. It only allows the current version of the state to be accessed at any time.

Monadic code can be reasoned about in a similar way to non-monadic code, using equational reasoning. In addition, an enhanced proof notation, giving the environment and state explicitly, is often useful. An example of such notation is given in Appendix A.6 where it is used to prove a sample lemma. There are also some standard properties of monads, given in Appendix A.4, but they are rarely used directly as they are very basic and simple. All of these can be combined to prove more complicated and useful lemmas about monadic programs. These are often specific to particular monads rather than of generic use. Some examples are given in Appendix A.4.

Monads do have some disadvantages. In particular, when dealing with state, every store and retrieve must be given explicitly, whereas in many imperative languages retrieves are implicit. However, the safety of monads under equational reasoning and their wide-spread use in Haskell, outweigh their disadvantages for use in the APM methodology.

This is not the place to explain how monads work in detail. All that is needed here is a basic understanding of how they are used. Interested parties can find a fuller introduction to them in [Wad92] and [PJ01].

How monads are used in Haskell

A Haskell expression that uses monads is called a *monadic expression.* This may be a function or a value. Most monads provide a selection of basic monadic functions. For example, *IO*, the standard Haskell monad for input/output provides functions that write characters and strings to the output and receive them from the input. In addition, all monads can use the standard function, *return*, which converts an ordinary non-monadic value into a monadic one. For example, *return* 5 is a monadic expression with the value 5.

Monadic expressions can be combined using Haskell **do** notation as follows:

$$\begin{array}{l}\textbf{do}\ E_1\\ \quad\ \ E_2\end{array}$$

This program carries out the computation expressed by the monadic expression, E_1, finishes it, and then carries out the computation expressed in E_2.

If the intermediate expressions return values, for example, a number that was entered on the input, these values can be given names in much the same way as in a **let** expression:

$$\begin{array}{l}\textbf{do}\ x \leftarrow E_1\\ \quad\ \ E_2\end{array}$$

In this piece of code, E_1 returns a value which is then bound to x. This value can then be used in E_2 and further pieces of code by referring to x. Although this looks similar to an assignment statement in an imperative language such as C, x is not a variable in the imperative sense and is not stored in the state. It is just a label or name. Stores in the state are carried out by explicitly calling store functions.

A monad has a name and type M. For example, *IO* is a standard Haskell monad for expressing input/output. Monadic expressions and values have type $M\ \alpha$. The first part, M, says what monad it uses, and the second part, α says what type of value it returns. For example, the function that takes an integer from the input has type *IO Int*. It is part of the *IO* monad and returns an integer. If no value is returned the type is $M()$. In general, monadic functions also have parameters, so their types look more like: $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow M\ T$. For example, a simple monadic function that sends its integer parameter to the output has type, $Int \rightarrow IO()$.

There are a variety of different monads, which encapsulate different side-effects. In particular, two standard Haskell monads, *ST* and *IO*, allow state to be manipulated and input/output to be carried out, respectively [LPJ95, PJW93]. There are also other standard monads which encapsulate other common features.

However the standard state monad, *ST*, uses a fairly simple model of the state. To model a parallel system a more complicated model is needed. A user can write his own monads in several ways. He can write them from scratch or by modifying an existing implementation, and can combine monads to allow multiple characteristics to be expressed [PJL94]. This section uses all of these methods to create a monad, *IOPST* (Input/Output Parallel State Transformer), to allow the manipulation of a model of the parallel state as well as providing input/output. The next few pages consider these aspects separately, and the implementations of the monad and its functions can be found in [Goo01a].

2.8.2 State and variables

Part of the job of the IOPST monad is manipulating the model of the system state.

This state, called *GlobalState*, contains the variables used in the program. It is represented by this collection of variables, each containing a sequence of values, one for each processor. These values come from a restricted set of types, which includes arrays, but excludes user-defined data-types, as is the case in C+MPI. The set as present does not contain all possible target types, but it can be extended.

The data is represented within the state in a special form, as an enumerated data type, in order to allow data of different types to be stored together, but the user need not worry about this representation. The IOPST operations perform automatic type conversions, so that the user only sees standard Haskell types.

The following types are used:

GlobalState The type of the whole system state.

VarFn α The type of variables of type α.

Dyn α The type class of types which can be stored in the state.

ItemType An enumerated type which describes the types which can be stored in the state.

This is not the only possible structure for the state. Other representations were considered, including a sequence of processor states, each of which contains a set of variables. The current method was chosen because it eases the transformation to the individual level on which MPI is based, by referring primarily to variables instead of processors. It also corresponds to the structure used within the ST monad in which the state is viewed as a set of pairs of variables and values.

IOPST state functions

The monad *IOPST* manipulates this representation of the state in a similar way to the standard monad *ST*, on which it was modelled. It is also augmented with some functions for managing multiple processors, some of which are described briefly in Table 2.4. In this table proc is used as an abbreviation for processor.

A few of these functions are described in a bit more detail below as examples to help the reader to understand the case study and to write his or her own programs.

$\textit{create\_var} :: (\textit{Dyn}\ \alpha) \Rightarrow \textit{ParFinSeq}\ \alpha \rightarrow \textit{IOPST}\ (\textit{VarFn}\ \alpha)$
create_var values

This creates a variable with one element of ***values*** in each processor and returns a name for it. It is designed to be used with a sequence of identical values as a parameter, such as that obtained using the $\textit{repeat}_P$ function. This ensures that the variable is initialised with the same value in each processor which is what happens in C+MPI. These initial values often need to be given a type annotation in order to remove type ambiguity over numerical values.

$\textit{store} :: (\textit{Dyn}\ \alpha) \Rightarrow \textit{VarFn}\ \alpha \rightarrow \textit{ParFinSeq}\ \alpha \rightarrow \textit{IOPST}()$
store var values

This updates ***var*** with ***values***, a sequence with one element for each processor. ***var*** must exist before this function is called.

$\textit{retrieve} :: (\textit{Dyn}\ \alpha) \Rightarrow \textit{VarFn}\ \alpha \rightarrow \textit{IOPST}(\textit{ParFinSeq}\ \alpha)$
retrieve var

Function	Type	Operation
start	$Int \rightarrow IOPST()$	Produces a new global state with the given number of procs and a dummy variable in it.
get_size	$IOPST\ Int$	Returns the number of procs in the state.
get_pid	$IOPST(ParFinSeq\ Int)$	Returns a sequence containing the id numbers of the procs.
create_var	$(Dyn\ \alpha) \Rightarrow ParFinSeq\ a \rightarrow IOPST\ (VarFn\ a)$	Creates a variable initialised with the given values. The variable can be given a name by assigning the result of the function to an identifier. See Section 2.8.4 for more details.
store	$(Dyn\ \alpha) \Rightarrow VarFn\ \alpha \rightarrow ParFinSeq\ \alpha \rightarrow IOPST()$	Updates the given variable with the given values.
retrieve	$(Dyn\ \alpha) \Rightarrow VarFn\ \alpha \rightarrow IOPST\ (ParFinSeq\ \alpha)$	Returns the values that are in the given variable.
$store_{indiv}$	$(Dyn\ \alpha) \Rightarrow Int \rightarrow VarFn\ \alpha \rightarrow \alpha \rightarrow IOPST()$	Stores a single value in the variable in the given proc, leaving the other procs alone.
runIOPST	$IOPST\ \alpha \rightarrow IO\ (\alpha,\ GlobalState)$	Runs an IOPST program.

Table 2.4: IOPST state functions

This returns the values stored in *var* in a sequence form. The *ParFinSeq* type emphasises that they represent values stored in different processors. Each element of the sequence corresponds to the value stored in one processor.

Dealing with individual processors

It is also sometimes helpful to be able to access values in a single processor rather than all the values in all the processors. $store_{indiv}$ and $retrieve_{indiv}$ do this. They both take the processor id of the appropriate processor as well as their more ordinary parameters. The store or retrieve now only affects the variable in the given processor.

2.8.3 Input/Output

As well as being modelled on the *ST* monad, *IOSPT* is built on top of the *IO* monad. Its type, therefore, follows the same pattern as the ST type, but involves IO:

$$\textbf{newtype}\ IOPST\ \alpha\ =\ IOPST\ (GlobalState\ \rightarrow\ IO\ (\alpha,\ GlobalState))$$

Its IO functions are defined simply by converting the standard IO functions to the IOPST type, using a lifting function,

$$liftIOPST\ ::\ IO\ \alpha\ \rightarrow\ IOPST\ \alpha.$$

The *IOPST* version of *putStr* which prints out a string, is then:

$$pst\_putStr\ xs\ =\ liftIOPST\ (putStr\ xs).$$

A brief description of the key IO functions is given in Table 2.5.

Function	Type	Operation
pst_putStr	*String* $\rightarrow$ *IOPST* ()	Displays a given string on the output.
pst_getChar	*IOPST Char*	Gets a character from the input.
pst_getLine	*IOPST String*	Gets a line from the input.

Table 2.5: IOPST IO functions

2.8.4 The form of an IOPST program

IOPST is the basis for most of the remaining stages in the methodology, and so many of the programs will have a similar form. It is important to understand how to interpret this form in order to understand many of the programs in this thesis.

Throughout the examples in this thesis, the following convention is used: a variable's name is usually suffixed with "*_v*", for example, *x_v*. The value extracted from such a variable usually takes the same name, but without this "*_v*". For example x is the name of the value extracted from *x_v*. This difference and this connection between a variable and its value are key in understanding much of the code in this thesis.

This difference leads to an abundance of *retrieves* in a program because a value *must* be retrieved from its variable before it can used. In other languages, such as C, this is not always necessary. C deals with a variable and its value as a single entity. This is a disadvantage of using monads to model the state in Haskell, but their ability to deal with side-effects outweighs this. The reader may find it helpful, therefore, to ignore all instances of *retrieve* in a program, and simply view x as the value obtained from *x_v* in the remainder of the code.

The only exceptions to this rule occur with standard global variables such as *n_v* and *p_v*, and in the function $retrieve_{indiv}$. In the former, n can be assumed to be the size of the input matrix and p the number of processors, without worrying about the code used to set these values. The latter case will be dealt with at the end of this section.

Another piece of code which can safely be ignored is a *create_var* expression. Although these are needed in the program for the sake of correctness, and can provide information to the reader about initialisation and the types of variables, they rarely add to the substance of the algorithm.

Example The following simple example illustrates how the IOPST state functions can be used together. Bigger examples can be found in Chapters 5 to 7.

```
main :: IOPST()
main = do a_v ← create_var (repeat (0 :: Int))
          store a_v xss
          a ← retrieve a_v
          store a_v (f a)
```

Ignoring the *create_var* and *retrieve*, this reads:

```
main = do store a_v xss
          store a_v (f a)
```

So *xss* is stored in *a_v*, then f is applied to the value in *a_v*. The extra code in the first version is necessary to make this work, but not necessary when reading it.

Dealing with individual processors

The introduction of $store_{indiv}$ and $retrieve_{indiv}$ (and later functions such as $retrieve_{Cyclicindiv}$—see Section 2.12) into a program makes things slightly more complicated. These functions deal with single values from one processor in the system and must be looked at a bit more carefully.

However, they often occur within code which deals specifically with one particular processor i. In this case it is fairly safe to assume that values are being extracted from that processor.

This situation often occurs in this thesis with the Haskell function, *for*, which simulates a C `for` loop. It is used in the following form: *for* xs $(\lambda\, i \rightarrow E\ i)$. In this, i takes each value from xs in turn, and E, a monadic expression, is executed with that value. xs is often $[a..b]$, so that i takes each value from a to b. Sometimes it is $[b, b-1, a]$, which is the reverse of the first case.

In Chapter 6, this function is used to step through the processors in a pipeline. Each time through the loop, the value i refers to the current processor.

Example In the following (rather contrived) example, each processor from 0 to $p-1$ is dealt with in turn. Even though $retrieve_{indiv}$ and $store_{indiv}$ are used, they both refer to the current processor i, and so do not add much complexity to the program.

```
for [0..p − 1]
   (λ i → do     — do calculation for proc i
              x ← retrieve_indiv i x_v
              new_vals ← f x
              store_indiv i new_vals_v new_vals)
```

2.9 Target stages

The target stages lie at the end of a derivation and are specific to the target language. They allow transformations and optimisations of the code that are specific to the target. For example, different languages allow different communication operations and therefore detailed communication optimisations are often language-specific. These stages also simplify the transformation into the target language at the end of the derivation.

The stages model the language using an APM or APMs in Haskell. A language-oriented APM models the target by providing Haskell functions and data types which correspond to and model the main parallel constructs in the language or typical ones in the language model. The APM depends a lot on the characteristics of the target language, and for more abstract languages, such as GpH, may be similar to the non-language specific stages above.

There may be more than one APM at this stage, to allow the language to be modelled at varying degrees of accuracy. Each APM is more detailed and closer to the target than the last. For example, when targeting C+MPI, the initial APM functions view the system on a collective level (see the next section 2.10). However C+MPI takes an individual processor's point of view. This is modelled at a later stage in the derivation, as shown in Section 2.11.

Finally the program is transformed from Haskell into the target language. It may not be possible to prove this transformation correct as the target language often does not have any formal semantics. Therefore the final Haskell version should use a model as close to the target as possible in order to make this final transformation as simple as possible.

In principle, many target languages can be modelled. However the size of this thesis allows only one to be examined. The following sections therefore focus on C+MPI [MPI97], a commonly-used extension of C with a library of message-passing functions. The APMs not only allow C+MPI to be used as the target in derivations, but also provide insight into modelling a target language in Haskell in general.

2.10 Modelling MPI

MPI [MPI97] is a library of message-passing functions that can be used with C or Fortran to write parallel programs. It is described in greater detail below. This section then goes on to describe an initial APM for modelling MPI in Haskell as an example and illustration of a language-oriented APM.

This initial APM continues to use the *collective view* of the parallel system used in the intermediate stages. In this view, the program operates over the whole system and changes the state of multiple processors. This is in contrast to the *individual view* that MPI uses, in which a program describes the action of a single processor and modifies the state of only that processor. It is instantiated several times, once for each processor. O'Donnell further discusses the difference between these individual and collective views in [O'D01]. The collective view is used here in order to simplify the transformation at this point and because a collective level view is more natural in Haskell. Implementing an individual level view involves some extra complexities as described in Section 2.11. Therefore this extra level of detail is introduced later in the derivation.

The MPI APM in its current form does not represent a final polished version, but rather a prototype in which the key features are implemented while some of the less central points are omitted. In particular, MPI's communicator system is not implemented, and the type of data that can be stored is simplified, as is the structure of the parallel system. This is sufficient for the derivation of many parallel programs and for the illustration of the key points.

This APM can be seen in use in the case studies in various sections including Sections 5.8 and 7.5.

2.10.1 MPI

MPI [DOSW96, MPI97] provides a set of message-passing functions, that deal primarily with communication and process topologies. An MPI programmer must specify the communication which takes place, and the placement of data. Both point-to-point and collective communication functions are available to help him with this.

Point-to-point communications involve passing data between a single pair of processors. They are given by sending and receiving functions, `MPI_Send` and `MPI_Recv`, and their variants which specify the type of communication, e.g., asynchronous, synchronous, buffered or blocking. An operation must give various information including the id of the other processor in the pair, although a receiving processor can receive from any other processor. MPI is explicit about the location of data.

Collective communications involve a pattern of communication between a group of processors. All the processors in the group must call the collective function before it executes. For example,

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

is the collective function for a broadcast. When all the processors in the group called *comm* execute this operation, then *count* values from *buffer* in the *root* processor are communicated to all of these processors.

MPI uses the SPMD style of programming. *SPMD* stands for Single Program Multiple Data, and is commonly used to mean two different but related models of computing. The more specific definition of SPMD is described in Section 8.2.3, but here its more general sense is used. A C+MPI or Fortran+MPI program is written from the viewpoint of a single processor, and specifies that processor's actions. The program is copied to all the processors in the system, which run it simultaneously. This does not mean that all the processors do the same thing at the same time as in the SIMD or data-parallel styles of programming. The processors may operate at different speeds, and the code may branch on processor id, so that different processors execute different parts of the code. For more about SPMD and SIMD programming see, for example, [WA99].

Function	Operation
mpi_spt2pt	This simulates a pair of functions—a synchronised send and receive in MPI. One processor sends a value to another processor.
mpi_jointpt2pt	This simulates a set of send/receive pairs in MPI. Each processor sends a value to one other processor.
mpi_bcast	A root processor sends a value to all the processors.
mpi_scatter	Values from a root processor are scattered to all the processors, a different value or values going to each processor.
mpi_scatterv	Same as *mpi_scatter* except that it allows a different amount of data to be sent to different processors.
mpi_gather	Each processor sends values to a root processor.
mpi_reduce	Each processor sends values, these are reduced to a single value using some operation and this is stored in a root processor. This is similar to *fold*.

Table 2.6: Outline of the MPI APM

2.10.2 The basics of the MPI APM

The *MPI APM* is based on the IOPST monad introduced in Section 2.8. It operates on *GlobalState*, the same model of the parallel system as *IOPST*, via *IOPST* functions. These functions may occur in programs at this stage, and they are also used to implement the APM operations themselves. The APM operations are therefore also monadic and return monadic values of type *IOPST* α for some type α.

2.10.3 APM functions

The MPI APM contains simplified Haskell versions of the functions provided by MPI. These functions specify and simulate the behaviour of message-passing functions in a parallel system. Here we focus on two of the MPI APM functions as representative examples. A fuller set is summarised in Table 2.6. Their implementations are discussed in Section 2.10.4.

mpi_jointpt2pt is a function that doesn't correspond directly to a single MPI function, but rather to a pattern of individual send and receive function calls in various processors. Such send and receive functions in MPI are described in Section 2.10.1.

$$\begin{aligned}&\mathit{mpi\_jointpt2pt} :: (\mathit{Dyn}\,\alpha) \Rightarrow \mathit{VarFn}\,\alpha \rightarrow \mathit{VarFn}\,\alpha \rightarrow \mathit{ItemType} \rightarrow\\ &\qquad\qquad\qquad (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow (\mathit{Int} \rightarrow \mathit{Int}) \rightarrow \mathit{IOPST}\,()\\ &\mathit{mpi\_jointpt2pt}\ \mathit{send\_v}\ \mathit{recv\_v}\ \mathit{datatype}\ \mathit{sendfn}\ \mathit{recvfn} = \ldots\end{aligned}$$

The two function parameters specify the pattern of communication. If a processor has rank or id *pid*, then it sends data to the processor with rank (*sendfn pid*). At this stage these processor ids are hidden inside the function implementation. Later they are accessed using the standard function, *get_pid*, described in Table 2.4. Receives correspond to the sends, and therefore *recvfn* must be the inverse of *sendfn*. Both are specified in order to allow this function to be transformed into C+MPI in which both functions need to be known.

The parameters *send_v* and *recv_v* are the variables from which the data is sent and into which it is received respectively. The parameter *datatype* indicates the type of this data, and is included because it is used as a parameter in the MPI functions.

This function allows the user to simulate an MPI program in which every processor engages in a send and receive, without writing out all of the individual sends and receives for every processor in the communicator. An example of its use is given in the function *requestrev* below in Section 2.10.5.

mpi_bcast is a more typical example. It corresponds directly to the MPI function of the same name, and is a collective communication function, i.e., it specifies communication between multiple processors. Such functions, and `MPI_Bcast` in particular, are described in Section 2.10.1.

```
mpi_bcast :: (Dyn α) ⇒ VarFn α → ItemType → Int → IOPST ()
mpi_bcast var_v datatype root = ...
```

As the MPI function does, this broadcasts or sends a value from *var_v* in the processor *root* to *var_v* in all the processors. The parameter *datatype* again gives the type of this data.

2.10.4 Function implementation

These functions manipulate the state of the system, and so can be implemented using the *IOPST* monadic functions.

For example, *mpi_bcast* can be implemented as follows. The other implementations are similar.

```
mpi_bcast :: (Dyn α) ⇒ VarFn α → ItemType → Int → IOPST ()
mpi_bcast var_v datatype root =
      do n ← get_size
         bcast_vals ← retrieve buffer
         bcast_vals' ← return (distfstoList bcast_vals)
         store buffer (toDistFinSeq (replicate n (bcast_vals'!!root)))
         return ()
```

The state manipulation function *retrieve* is used to obtain the value to be broadcast. However this gives the values, *bcast_vals*, in all the processors in the system. Sequence and list manipulation functions are used to pick the value from the root processor out of this. This is then duplicated so that there is one element per processor, before *store* is used to store that value in each of the processors.

2.10.5 Extra communication functions

Sometimes it is useful to have extra functions which perform communication but aren't provided by the MPI library. Often these functions are used frequently enough that it is sensible to define them in advance and include them in the MPI APM. Alternatively they can be given as local functions in the program. In the case-study in Chapter 5 such a function is required. That function performs a *reverse*—data from variable *send_v* in processor *i* is swapped with data from the reverse processor $n - i - 1$. The received data is stored in *recv_v*. This can be implemented using a set of send/receive pairings, and hence using *mpi_jointpt2pt* from the MPI APM, as follows:

```
requestrev :: (Dyn α) ⇒ VarFn α → VarFn α → ItemType
                → IOPST()
requestrev send_v recv_v datatype =
      do n ← get_size
         mpi_jointpt2pt sendbuf recvbuf datatype
             (λ i → (n − 1 − i)) (λ i → (n − 1 − i))
```

This is discussed further in Section 3.8.1.

2.10.6 ParOps

As with any APM, the behaviour of these functions can be expressed using ParOps (see Section 1.1.3). For example, here are the ParOp parametrisation functions for *mpi_jointpt2pt*:

$$\begin{aligned} \textsf{ParOp}: \; f_i(\mathit{sendvar}, \mathit{recvar}) &= (\mathit{recvar}, \mathit{sendvar}) \\ g_i(A_0, \ldots, A_{n-1}) &= A_{\mathit{recvfn}(i)} \end{aligned}$$

The g functions specify that each processor i takes in the value output by processor $\mathit{recvfn}(i)$. f_i then says that this is the value kept by the processor, and that the processor outputs its old value.

In addition, the MPI APM functions themselves form ParOps, as discussed in Section 2.3.5. However this is harder to see in the MPI APM case than with previous cases, because of the increased complexity of the model of the parallel system and because of the introduction of monads. It can be seen if the monads are viewed as simply syntactic sugar for functions that take one copy of the state and return another.

The difficulty is also increased because *GlobalState* is a set of variables, not a set of processors, which is what the ParOp form uses. Despite this difference in viewpoint it is still possible to identify the elements in an entity of type *GlobalState* which come from a particular processor, and hence possible to show the equivalence between a function of this type and the equivalent **ParOp** definition.

2.11 Individual level

As mentioned above and described in Section 2.10.1, MPI uses an individual level view of the parallel system: it describes the actions of a single processor. Previous stages in the methodology have used instead a collective view, describing the actions of the whole system. This section shows how the individual view can be represented in Haskell.

Not all languages need to have a stage such as this in the derivation of their programs. Many languages, such as HPF and GpH, use collective views of the system which match those of the previous stages.

2.11.1 Implementation

An implementation of the individual level is not actually necessary, and this thesis, because of time constraints, simply gives the individual level functions without an implementation. The individual level stage can be viewed as simply part of the final transition to the target language. Programs at this stage can be written, and the derivation can proceed without an implementation. However, if the program is to be run at this level (e.g., for prototyping purposes), an implementation must be provided.

It is not trivial to do this because the standard Haskell semantics assume a collective viewpoint. A Haskell function needs to be provided with all the data on which it operates. Therefore, in order to affect the entire system, a function must have access to data representing the whole system. This is in contrast with the individual viewpoint in which a function only has access to the data in a single processor and that received by communication. However, there are ways to get around this.

O'Donnell in [O'D01] has proposed a collective semantics that gives the "real" (i.e., collective) meaning of an individual level expression. This allows programs at the collective and individual levels to be proved equivalent.

It is also possible to write a program, or *wrapper function*, based on the collective semantics, which takes an individual level program and simulates its execution on multiple processors. Such a wrapper function would also have to handle the communication between these processors. It would play the role of the run-time system, and be quite separate from the program itself. It is only the

Function	Operation
store var value	The processor stores *value* in *var*.
retrieve var	Returns the value(s) stored in *var* in the current processor.
create_var ($V :: T$)	Creates a variable of type T initialised to V.
mpi_send var T dest	The processor sends *var*'s value (of type T) to processor *dest*.
mpi_recv var T source	The processor receives a value from processor *source* and stores it in *var*.
mpi_bcast var offset count datatype root	When called by all the processors, this initiates a broadcast of *count* elements from *var* in processor *root*.

Table 2.7: Some standard individual level functions

main individual level program which would correspond to the program in the target language, and therefore only this program which the programmer need be concerned with.

2.11.2 APM functions

The individual level APM models the same system as the MPI APM in the previous section, and so provides mostly the same functions. However these functions now operate on a single processor instead of on the whole system.

For example, storage manipulation functions, *store*, *retrieve* and *create_var* are provided as before, except that now they just deal with the state in a single processor. The type of *retrieve* changes from

$$(\mathit{Dyn}\ \alpha) \Rightarrow \mathit{VarFn}\ \alpha \rightarrow \mathit{IOPST}\ (\mathit{ParFinSeq}\ \alpha) \text{ to}$$
$$(\mathit{Dyn}\ \alpha) \Rightarrow \mathit{VarFn}\ \alpha \rightarrow \mathit{IOPST}\ \alpha.$$

It returns the single value stored in the current processor instead of the sequence of values stored in the whole system.

Collective communication functions operate in a similar way to their MPI counterparts. Each processor must call the function with the appropriate parameters. That processor then waits until all the other processors also call that function with appropriate parameters, and then the whole system carries out the operation.

For example, if *mpi_bcast* is called with the same type and root by all the processors, then the broadcast is carried out. If only one processor calls this function, then that processor waits until the other processors also call *mpi_bcast* before it does anything. If this doesn't happen then the processor hangs.

The necessary parameters for the call of one of these functions by an individual processor are the same as those in the collective call in the ordinary MPI APM. Therefore a call of these functions is the same as in the MPI APM.

Unlike these functions, individual communication functions, *mpi_spt2pt* and *mpi_jointpt2pt*, don't have counterparts with the same names in the individual level world. They can be replaced by combinations of the functions *mpi_send* and *mpi_recv* which model individual sends and receives in MPI.

Some of the standard individual level functions are summarised in Table 2.7, and the transformations from the ordinary MPI APM to this level are described in Section 3.9.

Function	Type	Operation
$create\_var_{Cyclic}$	$(Dyn\ \alpha) \Rightarrow (Cyclic\ \alpha) \rightarrow \rightarrow IOPST\ (VarFn_{Cyclic}\ \alpha)$	Creates and initialises a cyclic variable.
$store_{Cyclic}$	$(Dyn\ \alpha) \Rightarrow VarFn_{Cyclic}\ \alpha \rightarrow Cyclic\ \alpha \rightarrow IOPST()$	Updates the given cyclic variable with the given values.
$retrieve_{Cyclic}$	$(Dyn\ \alpha) \Rightarrow VarFn_{Cyclic}\ \alpha \rightarrow IOPST\ (Cyclic\ \alpha)$	Returns the values from a cyclic variable.
$store_{Cyclicindiv}$	$(Dyn\ \alpha) \Rightarrow Int \rightarrow VarFn_{Cyclic}\ \alpha \rightarrow \alpha \rightarrow IOPST()$	Stores a single value in position i of a cyclic variable (counting from 0).
$retrieve_{Cyclicindiv}$	$(Dyn\ \alpha) \Rightarrow Int \rightarrow VarFn_{Cyclic}\ \alpha \rightarrow IOPST\ \alpha$	Returns the ith value (using cyclic indexing) from a cyclic variable.

Table 2.8: Some Cyclic state functions

2.12 Data distributions in monadic stages

Data distributions can be introduced into a program in the methodology as described in Section 2.7. However, they aren't included in only that one stage in a derivation, but in several other stages as well. This section examines their effect on the monadic stages in a derivation targeting C+MPI, focusing on the cyclic distribution as an example.

2.12.1 Basic monadic stage

Monads can be introduced into a program to model imperative features as explained in Section 2.8. That section introduces a monadic type *IOPST* which operates on a model of a parallel system, *GlobalState*. The operations given in that section don't explicitly deal with data distributions, but they can be extended to do so.

First of all, a type is used to represent a variable with a particular data distribution. In such a variable, multiple values are stored in each processor, so, for example, a Cyclic variable can be implemented using the type:

$$\textbf{type}\ VarFn_{Cyclic}\ \alpha = VarFn\ (SeqFinSeq\ \alpha)$$

Such variables can be manipulated in a similar way to ordinary variables, except that they deal with *Cyclic* data. Some storage manipulation functions on cyclic variables are given in Table 2.8.

Note the similarity of the first three functions in the table to their non-cyclic counterparts in Table 2.4. These functions use the whole of the cyclic data structure, but, just as it was sometimes useful to deal with data from a particular processor in Section 2.8.2, it is sometimes useful to access particular pieces of data in a cyclic structure. The last two functions in the table, annotated with *Cyclicindiv*, are provided for this purpose. They use cyclic indexing as mentioned in Section 2.7 to identify the element being dealt with.

2.12.2 MPI APM

Data distributions can be represented in the MPI APM as variables containing *SeqFinSeq* data. These can then be manipulated using ordinary MPI APM functions, making sure that you remember the extra values in each processor. However data distributions can also be dealt with explicitly, using variants of MPI APM functions dealing with the distribution, even though the target, MPI, doesn't provide operations to do so. This simplifies the introduction of MPI APM operations as the data distribution details don't have to be dealt with by the programmer yet. Later on the explicit data distribution functions will be removed and replaced by the combinations of ordinary MPI APM functions before the final transformation to C+MPI (see Section 3.10.4).

A data distributed MPI APM is based on the modifications to the *GlobalState* above, and the functions in this APM are implemented using combinations of ordinary MPI APM functions and the state manipulation functions given above.

Example For example, here is a cyclic version of *mpi_spt2pt*, which sends and receives a single value in a cyclic structure. To simplify its use, *source* and *dest* indicate positions in the cyclic structure (using cyclic indexing) rather than processor ids.

$$\begin{array}{l}
mpi\_spt2pt_{Cyclicindiv} :: (Dyn\ \alpha) \Rightarrow VarFn_{Cyclic}\ \alpha \rightarrow ItemType \rightarrow Int \\
\quad \rightarrow VarFn_{Cyclic}\ \alpha \rightarrow ItemType \rightarrow Int \rightarrow IOPST() \\
mpi\_spt2pt_{Cyclicindiv}\ send\_v\ sendtype\ source\ recv\_v\ recvtype\ dest\ = \\
\quad \mathbf{do}\ message \leftarrow retrieve_{Cyclicindiv}\ source\ send\_v \\
\qquad store_{Cyclicindiv}\ dest\ recv\_v\ message
\end{array}$$

Collective functions don't access individual values in the same way. They manipulate the state of the whole system, being careful to maintain the data distribution of the values.

Example Here, for example, is a version of *mpi_scatter* for cyclic data. It scatters the values in processor *root* in *send_v*, which is an ordinary sequence variable, and stores them in all the processors in *recv_v*. However, instead of sending one value to each processor, it distributes them in a cyclic manner. Therefore it's useful for distributing the values initially. However it's still called *mpi_scatter* to emphasise its link with the ordinary form of *mpi_scatter*.

$$\begin{array}{l}
mpi\_scatter_{Cyclic} :: (Dyn\ \alpha) \Rightarrow VarFn\ (SeqFinSeq\ \alpha) \rightarrow ItemType \\
\quad \rightarrow VarFn_{Cyclic}\ \alpha \rightarrow ItemType \rightarrow Int \rightarrow IOPST() \\
mpi\_scatter_{Cyclic}\ send\_v\ sendtype\ recv\_v\ recvtype\ root\ = \ \ldots
\end{array}$$

This function can be implemented using the *makecyclic* function from the Cyclic APM (see Table 2.3) to convert a list into a Cyclic data structure. It can also be implemented using ordinary MPI APM functions but this is more complicated. The values must be reordered using *makecyclic* within the *root* processor, and then appropriate size and displacement variables set up before an ordinary *scatter* function is used.

2.12.3 Individual level

Data distributions can also be used in the individual level, although the functions here are closer to the ordinary APM functions since the program only sees the data in a single processor. Section 3.10.3 shows how collective Cyclic MPI APM functions can be transformed into these individual level functions.

Table 2.9 gives some individual level functions which can be used to manipulate cyclic data more easily. They operate on the sequence of values belonging to the given variable in the current processor. The length of this sequence, *length*, may need to be known. If so, it can be easily calculated by:

$$\begin{array}{l}
length\ =\ \mathbf{if}\ (pid\ \geq\ n`mod`p)\ \mathbf{then}\ (n`div`p)\ \mathbf{else}\ (n`div`p) + 1 \\
\quad \mathbf{where} \\
\qquad pid\ =\ processor\ id \\
\qquad p\ \ \ \, =\ total\ number\ of\ processors \\
\qquad n\ \ \ \, =\ total\ number\ of\ elements\ in\ the\ variable
\end{array}$$

For communication, two basic functions, *mpi_send'* and *mpi_recv'*, are provided. These are variants of the standard send and receive MPI functions, but access particular elements of sequences.

Function	Operation
$create\_var_{Cyclic}$	Creates a Cyclic variable. Takes *length* values to initialise it.
$retrieve_{Cyclic}$	Operates on a $VarFn_{Cyclic}$ and returns the sequence of values stored in the processor.
$retrieve_{indiv}$	Returns the ith value which is stored in the variable in the current processor.
$store_{indiv}$	Stores a value in the ith position of the sequence associated with the variable in the current processor.

Table 2.9: Some individual level Cyclic state functions

This can be done in the target, C+MPI, using offsets from the base address of the variable. They can be implemented using *store*, *retrieve* and the standard functions.

mpi_send' For example, *mpi_send'* sends the value in location *offset* of *var_v* to processor *dest* and can be implemented as follows. It stores the value to be sent in a temporary variable, *tmp_v*, before it sends it.

```
mpi_send' var_v T dest offset =
    do tmp_v ← create_var (V :: T)
       var ← retrieve var_v
       store tmp_v (var !!S offset)
       mpi_send tmp_v T (dest`mod`p)
```

Other communication functions can be created, based on these.

2.13 Summary

This chapter has presented and discussed key stages in the APM methodology for a derivation targeting C+MPI, as well as discussing stages in the methodology in general. The stages use APMs (Abstract Parallel Machines) to encapsulate the parallelism, and these, their functions and their implementations have also been described.

However the stages only allow the program to be given at each stage. An important question is how one can move between stages. Given a program at an abstract, specification stage, how is an implementation derived? This is the topic of the next chapter.

Chapter 3

The Transformations in the Methodology

3.1 Introduction

Although the stages through which a derivation passes are very important, it is equally as important to know how the derivation progresses from one stage to the next. This is the topic of this chapter. It discusses various key transformation steps in the methodology, relating them to the stages and APMs in the previous chapter and giving transformation lemmas and rules for them. The transformations given are those necessary for the production of a basic C+MPI program that involves data distributions. The transformations and their techniques are also discussed in general.

This chapter focuses on how the transformations are carried out, although it does also mention how to choose which transformations should be done. However this latter is considered in greater detail in the next chapter (Chapter 4).

3.1.1 Layout of this chapter

After examining some of the general issues surrounding the transformations in Section 3.2, this chapter examines several different transformations in turn, starting with the generation of specifications (Section 3.3) and finishing with the transformation into the target language, in this case, C+MPI (Section 3.11). While these are, in general, ordered as they would be in a derivation, some of the vertical and horizontal transformations are considered together because of the similarity between them.

3.2 General discussion

The remainder of this chapter looks at individual transformations in detail, but first of all it is useful to consider more general aspects of the transformation process. This section examines the layout of that process, and some features and issues which are common to all or most of the transformations.

3.2.1 Layout of the transformations

The specific transformations only really make sense in context, when one can see how they are arranged and related to each other, and what types of program they are applied to.

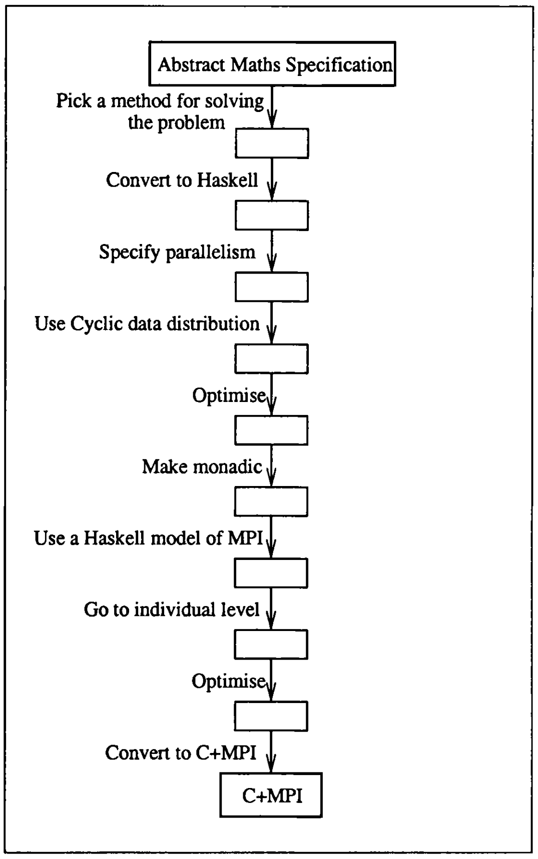

Abstract Maths Specification
Pick a method for solving the problem
Convert to Haskell
Specify parallelism
Use Cyclic data distribution
Optimise
Make monadic
Use a Haskell model of MPI
Go to individual level
Optimise
Convert to C+MPI
C+MPI

Figure 3.1: The layout of the transformations in a possible derivation

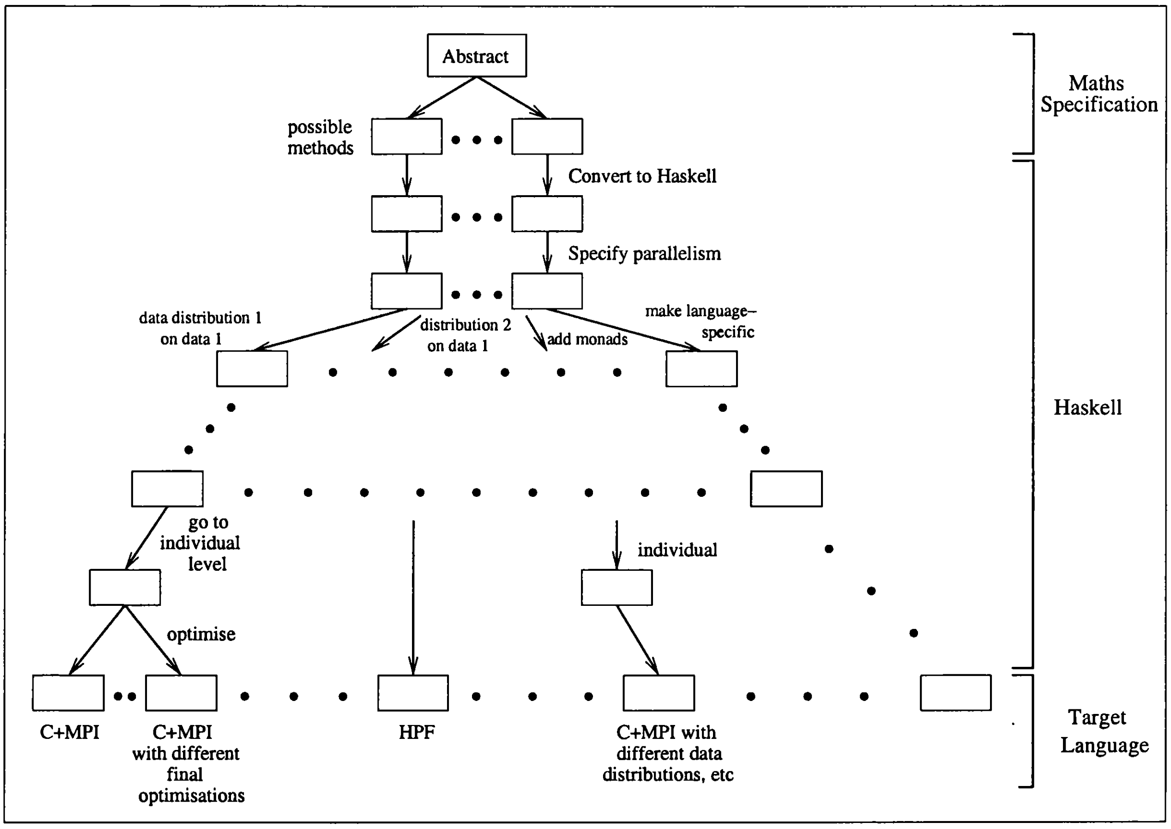


Figure 3.2: The structure of possible program derivations

Therefore Figure 3.1 sketches out the layout of one possible derivation, using the transformations presented in this chapter. Although this is just one possible layout, it illustrates several key points. The program starts off as an abstract maths specification, and then gradually details are introduced to derive a concrete C+MPI program. Each transformation operates on the program to increase the concreteness in a specific identifiable way. Some transformations bring the program closer to the target, while others introduce optimisations.

This layout is related to the layout of the stages, illustrated in Figure 2.2 in the previous chapter. Many of the transformations convert the program from using one APM or combination of APMs to another, keeping within the structured APM layout, so that more abstract APMs are replaced by more concrete ones. Such transformations are called *vertical* transformations because they move vertically in the APM structure. Others, known as *horizontal* transformations, keep the same APM, and manipulate the functions and data within the program.

However, this is only one of many possible derivations, as shown in Figure 3.2, which illustrates several derivations starting with the same specification. The key points above apply to all of the derivations, but they are different from each other. A particular derivation forms one possible path through this tree of possible derivations.

At many of the stages, the user has a choice of transformations to apply. Different choices produce different derivation paths and may lead to different resultant programs. Some transformations are necessary for certain programs or target languages, but not for others. For example, the change to an individual level viewpoint, near the end of a derivation, is only needed for target languages which view the parallel system from this viewpoint (mostly SPMD and SIMD programs). Other transformations, such as load balancing, are completely optional, or can be applied in different ways depending on the program.

Transformations also don't have to be applied in a fixed order. As Section 2.2 emphasises, the derivation paths are flexible. The order of steps can be changed and a transformation can be carried out more than once. This may lead to different target programs but the same target program can also be produced in more than one way. This typically occurs when the order of certain steps is immaterial. In such a case, applying transformation 1 before transformation 2 may have same result as applying transformation 2 first. This is not shown in the diagram as it would obscure its structure.

3.2.2 Transformation rules and lemmas

As described in Section 1.1.2, equational reasoning, a standard technique from functional programming, is used in the methodology to carry out the transformations. It can be used to prove lemmas and rules that encapsulate key transformation and reasoning steps. These can then be applied to a program to carry out those steps.

The rules and lemmas in this thesis use a specific notation. A *lemma* has the general form $E_1 == E_2$. This notation indicates that expressions are equivalent and avoids ambiguity with the symbol $=$ as the latter is often used inside programs themselves. A lemma can be applied in either direction, replacing instances of E_1 with E_2 and vice-versa. A transformation rule has the general form $E_1 \Rightarrow E_2$. It is uni-directional and indicates that the left hand side of the rule can be transformed into the right hand side, rather than asserting that the two sides are equivalent. Both rules and lemmas may have associated conditions and they describe the type and use of the variables within them.

Important Note: The lemmas and rules are gathered together in Appendix A for ease of reference. There they are organised into topics and numbered consecutively. These numbers are also used to refer to them in this and other chapters. However, it is often necessary to introduce the lemmas and rules in a different order in the thesis body to that in the appendix, and so they often appear with non-consecutive numbers.

Libraries of lemmas and rules

It is often useful to gather the lemmas and rules into libraries to structure and organise the collection, making it easier to locate a particular rule. These libraries also prevent the search space from becoming crowded because we can allow access to only the relevant libraries and hence lemmas at each step in the transformation.

This is useful because not all of the lemmas are applicable to all the stages in the methodology. Some of them, the rules for transforming between APMs in particular, are only applicable to one step in the transformation. This is because these rules explicitly mention functions from particular APMs, and thus can only be applied when those functions occur in the program. Other lemmas apply to a larger subset of the transformations. For example, the monadic lemmas in Appendix A.4 can be applied to monadic programs. They are not applicable to the whole derivation since monads are only introduced part-way through it (see Figure 3.1). However, after they are introduced, there are several stages involving them, for which such a set of lemmas is useful. There are also lemmas that apply to the whole of the derivation, such as the general lemmas for tidying up the program in Section 3.5.1 and Appendices A.1 and A.2.

This thesis organises the lemmas in Appendix A into several sections, each of which can be considered to be a library of lemmas. In addition, this chapter presents several transformation rules. Those in the same section can be considered to form a library.

However, these libraries are far from complete. In some cases, only example lemmas and rules are given. Other lemmas in the same libraries can be written along the same lines. In addition, this

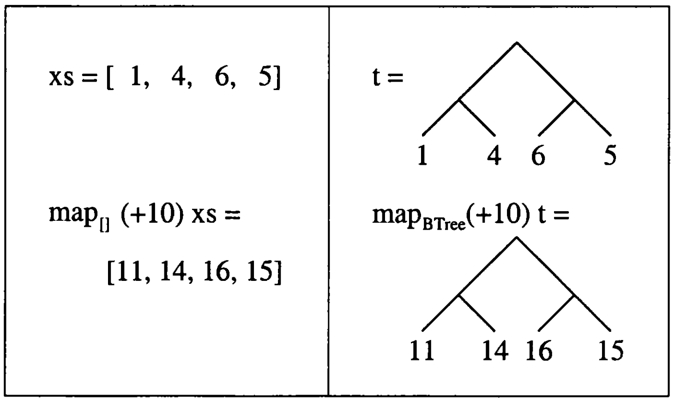


Figure 3.3: Example of a vertical transformation

thesis focuses on the derivation of C+MPI programs, and therefore libraries for other target languages are not considered, although they could be produced using similar methods. In a way, the libraries will never be truly complete, because a user may add his own APMs and corresponding lemmas for transformation within and between them.

Proofs

Since the transformation lemmas state equivalence between different Haskell expressions they can usually be proved using equational reasoning, perhaps with techniques such as structural induction and co-induction [Gor94]. However, vertical transformations present extra challenges. As mentioned previously, such transformations convert functions from one APM or APMs to another. If both APMs use the same model of the parallel system, this is fine, but if not, there can be problems.

For example, Figure 3.3 shows two data structures, or models of a system, which contain the same data. The corresponding APMs provide functions, $map_{[]}$ and map_{BTree}, which both map a function over the given structure, as shown. In a vertical transformation between these APMs, the code on the left is converted into that on the right. However, these pieces of code are obviously not equal because their results, despite containing the same values, have different data structures. Therefore, rather than proving the desired rule, $map_{[]}\ f\ xs \Rightarrow map_{BTree}\ f\ t$ when $xs \Rightarrow t$, directly, an associated lemma is proved.

An ***observation*** function converts values between the two types. In this case, the function $flatten :: BTree\ \alpha \rightarrow [\alpha]$ can be used. The rule becomes: $map_{[]}\ f\ (flatten\ t) = flatten\ (map_{BTree}\ f\ t)$ which is an ordinary lemma and can be proved using equational reasoning.

In general, proofs of vertical transformation rules are done in this way, using observation functions and associated lemmas. However, for the sake of simplicity the rules themselves are given without the observation functions. This allows the production of programs without observation functions all over the place.

3.2.3 Simplification of the derivation

There are many transformation steps, some of which are very detailed, and it is unlikely that a programmer would want to apply them all by hand.

Therefore the derivation process can be shortened by skipping some steps in the derivation, if the programmer feels that they are sufficiently simple or commonplace. Alternatively some versions of the program can be written by hand without formally applying the transformation, although this is often likely to prove harder than using the transformation rules. If a formal proof of the program is later required, the missing steps can be filled in.

An alternative is to use machine support and automation. This may be particularly useful in the later stages when the transformations are very detailed. Many of these transformations involve little intelligence, and could possibly be automated, by a compiler.

This is, however, not always desirable as many of the transformations involve difficult choices. These can be made using cost models, but the automation of this process is difficult and not entirely satisfactory. We also want to leave room in the methodology for intuition and insights on the part of the programmer. Nevertheless tool support is still possible for such stages, helping the programmer with the fiddly details, but without restricting the choices available. This could be done using a reasoning assistant or theorem prover for Haskell, such as Era (Equational Reasoning Assistant) [Win]. It could help prove the lemmas, and apply them to transform the program.

However, this remains for future work as it lies outside the scope of this thesis.

3.3 Specifications

As mentioned in Chapter 2, a derivation starts with a specification of the problem to be solved. There are several specification stages in the methodology: the programmer starts with an abstract mathematical specification, for which she may produce an algorithmic maths specification. This is more concrete and a Haskell specification can be produced from it. The production of such specifications and their modifications are dealt with in this section.

3.3.1 Mathematical specifications

The use of mathematics to specify the problem is discussed in Section 2.4.1 where two different types of maths specification are introduced. The original abstract specification simply gives the essentials of the problem, while the algorithmic specification gives a method for solving it. There are various ways to produce both of these specifications, but this is not the topic of this section because it is the transformations in Haskell which are really of interest, and because much previous work has been done on this in other fields of research. More information on producing the initial specification can be found in books such as [Inc88], and books such as [CLR90] and [AHU74] provide an introduction to and examples of algorithms.

3.3.2 Producing a Haskell specification

As mentioned in Section 2.4.2, the specification is then written in Haskell, so that further transformations can be done within this language. The process of translating a mathematical specification into Haskell can be complex because of the wide range of programs which can be specified using maths. This thesis does not attempt to comprehensively cover all possibilities, but focuses on some common basic expressions as examples. Section 7.2 discusses how transformations of other maths expressions can be produced.

Table 3.1 gives some transformation guidelines, focusing on finite sequences as these form the key data structure used within this thesis. The table is broken down into several sections, with several rules in each. These are not meant to be exhaustive or to show the only possible way of transforming an expression, but rather demonstrate how certain types of mathematical constructs can be transformed. Further examples, involving incremental loops, can be found in Section 7.2.

The guidelines make heavy use of Haskell higher-order functions, such as *map*, *foldl*1 and *take* to encapsulate common mathematical operations and structures such as loops. This helps to clarify the

No.	Mathematics	Haskell
Specific mathematical loops These hold for all $\oplus :: \alpha \to \alpha \to \alpha$ for any type α		
1	$\bigoplus$	$\mathit{foldl1}\ \oplus$
2	$\sum$	$\mathit{sum}\ (= \mathit{foldl}\ (+)\ 0)$
3	$\prod$	$\mathit{product}\ (= \mathit{foldl}\ (*)\ 1)$
Some for loops with independent iterations (see Table 7.1 for incremental loops) The first holds for any $f :: \alpha \to \beta$, $S :: [\alpha]$ The second holds for any $f :: \alpha \to \beta \to \gamma$, $X :: [\alpha]$, $Y :: [\alpha]$		
4	for each $x \in S.fx$	$\mathit{map}\ f\ S$
5	for $(x, y) \in \mathit{zip}\ X\ Y . f\ x\ y$	$\mathit{map2}\ f\ X\ Y$ $(= \mathit{zipWith}\ f\ X\ Y)$
Loops that only affect certain elements These hold for any $f :: \alpha \to \beta$, $p :: \alpha \to \mathit{Bool}$, $S :: [\alpha]$		
6	for each $x \in S$ s.t. px holds. calculate fx	$(\mathit{map}\ f).(\mathit{filter}\ p)$
7	for each $x \in S$. if px then fx else gx (i.e., same as 6 but using g if p doesn't hold)	$\mathit{map}\ f'$ where $f'\ x\ =\ \mathit{if}\ p\ x\ \mathit{then}\ f\ x\ \mathit{else}\ g\ x$
8	for elements $i = j + 1..n.f$	$(\mathit{map}\ f).(\mathit{drop}\ j)$
9	Same as 8 but returning the whole set,S	$(\mathit{take}\ j\ S)$ ++ $(\mathit{map}\ f).(\mathit{drop}\ j)\ S$
Some examples of other functions These hold for any $f :: \alpha \to \alpha$, $x_0 :: \alpha$, $p :: \alpha \to \mathit{Bool}$, $a, b :: \mathit{Float}$		
10	iterate f on x_0 until p holds	$\mathit{until}\ p\ f\ x_0$
11	$\int_a^b f(x)dx$	$\mathit{integrate}\ f\ a\ b$ for an appropriate definition of *integrate*

Table 3.1: Some guidelines for producing Haskell specifications. In these $\oplus, f, g$ are functions, p is a predicate, S, X, Y are sequences, i, j are floating point numbers, and x_0 is a value of f's domain.

structure of the program, and the nesting levels of the data. It is also useful in future transformations and because it lets the Haskell specification stay fairly close to the mathematics.

Most of these functions are APM functions, and are discussed in more detail in Section 2.4.2 where the APM for the Haskell specification is described. The programmer may also define new functions to be included in that APM and for use in the Haskell specification. The function *integrate* in guideline 11 is an example.

However, sometimes such functions cannot be implemented accurately. This is the case with the function *integrate*. Approximation algorithms can then be used for the implementation to approximate results rather than evaluate them accurately. These should, however, be hidden from the main program, and only the function name and an appropriate specification of the function given. However, these situations are awkward as the subsequent program may be difficult to transform. It is preferable to use an algorithmic mathematical specification which can be transformed more easily into Haskell.

Care should also be taken to ensure that the new Haskell specification actually specifies the same thing as the old maths one. If necessary, this can be done formally. As all the sequences are of finite length, induction can often be used for these proofs, as in the example below.

Example proof of a transformation rule Transformation rule 1 from Table 7.1 in Chapter 7 is used as an example, as it is less trivial than most of the examples here, and yet still simple enough to demonstrate a proof.

For $f :: \alpha \to \alpha, a :: \alpha, xs :: [\alpha], n :: Int$.

$$\begin{array}{lcl} res = a & \equiv & res = foldl\ f\ a\ xs \\ for\ i = 1, \ldots, n.res = f\ res\ x_i & & \mathbf{where}\ xs = [x_1, \ldots, x_n] \end{array}$$

This can be proved using induction:

<u>**Base case: n = 0 ⇒ xs = []**</u>

$$\begin{array}{lcl} res = a & \equiv & res = a = foldl\ f\ a\ [] \\ for\ i = 1,\ \ldots,\ 0.\ res = f\ res\ x_i & & \end{array}$$

<u>**Inductive case: n = n + 1**</u>

$$\begin{array}{ll}
 & res = a \\
 & for\ i = 1,\ \ldots,\ n+1.\ res = f\ res\ x_i \\
\equiv & \\
 & res = a \\
 & res = f\ res\ x_1 \\
 & for\ i = 2,\ \ldots,\ n+1.\ res = f\ res\ x_i \\
\equiv & \\
 & res = f\ a\ x_1 \\
 & for\ i = 2,\ \ldots,\ n+1.\ res = f\ res\ x_i \\
\equiv & \{\text{by induction hypothesis}\} \\
 & res = foldl\ f\ (f\ a\ x_1)\ [x_2, \ldots, x_{n+1}] \\
 & \quad = foldl\ f\ a\ (x_1 : [x_2,\ \ldots,\ x_{n+1}]) \quad \{\text{by definition of } foldl\} \\
 & \quad = foldl\ f\ a\ xs
\end{array}$$

Therefore the result holds by induction.

Example of a rule in action The following simple example shows how these guidelines can be applied to a mathematical specification. In this, x_{ij} are elements of a matrix x. $x_{ij}, s_i :: \alpha$, $\oplus :: \alpha \to \alpha \to \alpha$, and $n :: Int^+$.

$$s_i = \bigoplus_{j=0}^{n-1} x_{i,j}, \quad \text{for } 0 \le i < n$$

becomes (using guideline 1):

$$s_i = foldl1 \oplus x_i, \quad \text{for } 0 \leq i < n$$

and therefore (by 4):

$$s = map(foldl1\oplus)x$$

Further examples can be found in Section 7.2.

3.3.3 Restricting the specification

It can be useful to keep the original mathematical and hence Haskell specifications general. They can then be used to derive a wide range of programs for a wide range of targets and situations. However too much generality has problems because, as mentioned above, abstract specifications are often hard to manipulate. The extra algorithmic maths specification was introduced to help with this.

Early in the derivation, it is better not to give too much detail or the possible targets will be too restricted. Therefore, it is often useful to restrict the specification or make it more concrete later in the derivation. This can be used to allow particular techniques which only apply in special cases, or to optimise the program for one particular form of input. It can also make the program easier to manipulate or reason about.

Restricting the specification may involve adding assumptions, for example, about the input data. It may therefore restrict the situations in which the program can be used correctly. It is important to keep track of these changes to the specification so that the final program is not used in situations for which it was not intended and in which it may not produce the right results.

Examples from map-triangle

The map-triangle case study in Chapter 5 gives some examples of this, especially in the introduction of load balancing in Sections 5.3 and 5.4. Load balancing can only be applied if some restrictions are fulfilled - the method used to split the tasks up only works if the function, f, is associative and has unit a. Otherwise, it does not produce the correct results. This can be seen in the conditions on the lemma involved (Lemma 15).

Load balancing also illustrates another point. Sometimes the program is especially inefficient for particular inputs or sets of inputs, and so it is worth-while considering them separately from the general case. In the case study, this occurs when the input matrix is triangular, because this produces a particularly bad load balance.

Instantiating types

Another example is the instantiation of data types. The original specification usually does not say which particular type each input or output should have. It may work for many different types of data. This can be expressed in Haskell using polymorphism, but many target languages require specific types. For example, the MPI library requires the types of values sent and received to be given explicitly. In addition, some of the later Haskell stages also use specific types (see, for example, Section 2.8.2). The use of specific types may also allow particular techniques to be used as described above.

Therefore it is often useful to instantiate the general, polymorphic types at some point in the derivation. This is another restriction of the specification. Once the types are instantiated, the program no longer has to work for a range of types, but only for the given one.

This can take place at various points during the derivation, but it must occur before APMs which don't support polymorphism are used. In the examples in this thesis, this must happen before variables are stored in the state because of the implementation of the state used in the *IOPST* monad.

This transformation does not produce an equivalent program, but it does produce one which gives the same results on data of the given types. The code remains the same as before, and only the types are changed. The transformation can therefore be summed up by the following lemma.

Rule 2 (Instantiate types)
For all $f :: T(\alpha_1, \ldots, \alpha_n)$, $g :: T(T_1, \ldots, T_n)$, $x_i :: T_i$ such that $f\ x_1 \ldots x_n = g\ x_1 \ldots x_n$ for $x_i :: T_i$.

$$f\ x_1 \ldots x_n \quad \Rightarrow \quad g\ x_1 \ldots x_n$$

Where α_i represents a polymorphic type, and T_i any specified type, which makes α_i more concrete.

3.3.4 Modifying the data format

It may also be useful to modify the specification in other ways throughout a derivation. In particular, the data format used may be changed. This format is not a characteristic of the original maths specification, which assumes no particular format for the data. However the Haskell specification must use some format for the data, and the one chosen is not always the best one. It may be useful to change it, either at the start of the derivation or later on when the best format becomes more apparent, due to additional information about the parallelism and the target.

This can be illustrated with an example. The key data structures used in this thesis are matrices and arrays, and matrices can have multiple data formats. Although in the maths specification they are simply grids of elements, in this thesis they are usually represented as sequences of rows of elements. This is abstract in several ways, but affects the ease with which different parallel distributions can be expressed. Because matrices are given row-wise, as sequences of rows, it is easy to express the distribution of rows across the parallel system, but not the distribution of columns. Therefore it can be useful to change the data format to a column-wise implementation, in which the matrix is represented as a sequence of columns rather than a sequence of rows.

This can be done using the standard function, *transpose*, which transposes the rows and columns of a nested list of type $[[\alpha]]$. A standard implementation on which the *FinSeq* implementations and proofs can be based is:

```
transpose                :: [[α]] → [[α]]
transpose []             =[]
transpose ([] : xss)     =[]
transpose ((x : xs) : xss) =(x : [h | (h : t) ← xss]) :
                              transpose (xs : [ t | (h : t) ← xss])
                            or(x : map head xss) :
                              transpose (xs : map tail xss)
```

However the data cannot be transposed without changing the functions that act on it to take this into account. The necessary changes can be encapsulated in a set of lemmas dealing with *transpose*. These are based around the basic property that *transpose* is its own inverse:

Lemma 27 (Inverse of transpose)
For all $xss :: [[\alpha]]$ such that $\#(xss!!i) = \#(xss!!j)\ \forall\, i, j$.

$$transpose(transpose\ xss) == xss$$

The condition simply states that xss is a matrix. The lemma can be proved by structural induction, as shown in Appendix A.6.

Other lemmas describe the effect that introducing *transpose* has on other functions. For example, the effect on *map* is very simple as shown in the following lemma.

Lemma 31 (Transpose and map)
For all $f :: \alpha \to \beta$.

$$map(map\ f)\ ==\ transpose \cdot (map(map\ f)) \cdot transpose$$

Other lemmas are more complicated, but are also based on the basic properties of *transpose*. They can be proved using these properties and structural induction.

Applying these lemmas produces a program with instances of *transpose* in it. However it is not always necessary to include these explicitly. If the new data format is to be used for the duration of the program, then the data can be transposed before the program is called and transposed back again afterwards. If the program is to be self-contained, then the assumptions about the format in which the data is input can be changed. In this example, the data can be entered column by column instead of row by row. The *transposes* are removed from the program after the above lemmas are applied, but one must keep track of the assumptions about the data format.

3.4 Basic vertical transformations

As explained in Section 1.1.3, a *vertical transformation* transforms the program from using one APM or set of APMs to another. Due to the implementations of APMs, this usually involves a change in the Haskell types on which the APM functions operate. *Observation or conversion functions* convert between the old and new types in the proofs of the transformation rules. However the rules themselves don't mention the observation functions to keep them from cluttering up the code. The observation functions can also be used to change the types of data in a program.

This section examines some of the vertical transformations in the first half of a derivation. In these the parallelism is made concrete and data distributions are specified. Some of the later transformations are also vertical, but they are considered separately in Sections 3.6 and 3.8 because they involve many stage-specific details.

3.4.1 Making parallelism explicit

One of the key vertical transformations makes the parallelism in the program explicit. This is important for many, though not all, target languages because many of them, including C+MPI, the target language in this thesis, are explicit about what is to be executed in parallel and what not, even if about nothing else. Knowledge about this basic parallel property is also important when introducing optimisations and data distributions later in the derivation.

Because of its wide applicability and relatively high level of abstraction, this stage usually occurs near the start of a derivation, as shown in the case studies. However, it *can* occur later in the derivation. It is also not necessary to carry it out on all parts of a program at the same time. The parallelism of one piece of data or nesting level can be given early in the derivation, but that of another may be left until later.

The parallelism of a program is made explicit using the annotations and data types introduced in Section 2.6. The data type *SeqFinSeq* α is a finite sequence type, but also indicates that all of its data is stored in a single processor; its corresponding functions are annotated with S. Similarly, *ParFinSeq*α and P indicate data stored in multiple processors.

This transformation therefore has two distinct parts—data can be made either sequential or parallel. The next two sections look at these two cases separately, but first of all, this section considers several of the aspects that they have in common.

Lemmas

The sets of lemmas and transformation rules for these transformations replace functions from the abstract specification APM with ones from the sequential and parallel APMs. These APMs are very similar, and so a function from the abstract APM is usually replaced by a single function of the same name from the new APM. Therefore the lemmas have the general form, $f = f_T$ where T is S in the sequential case and P in the parallel.

However, these transformations cannot be applied to a function in isolation. When a function changes, the data on which it operates changes and therefore other functions on that same data must also change. In addition, making one piece of data sequential or parallel has impact on other related data.

Therefore, the transformations are combined using larger lemmas which transform the whole program, not just a single function. These lemmas are different for the sequential and parallel cases and are therefore considered separately in the following sections.

Proofs

The lemmas in this section are vertical lemmas, and so are proved using observation functions, as described in Section 3.2.2. For these basic vertical transformations, the initial APM is the abstract APM. This uses the type *FinSeq* α to model the parallel system. The sequential and parallel APMs use *SeqFinSeq* α and *ParFinSeq* α respectively. Therefore the observation functions have types *FinSeq* $\alpha \rightarrow$ *SeqFinSeq* α and *FinSeq* $\alpha \rightarrow$ *ParFinSeq* α.

The details of the observation functions depend on the particular implementations of *FinSeq*, *SeqFinSeq* and *ParFinSeq*. In this thesis, they are all implemented in the same way, using lists, and therefore the observation functions are particularly simple. Using the current implementations, the observation functions just apply renaming functions to convert between the types. For example, in the sequential case, the observation function, $o\ s$, is *list2seqfs* (*fromFinSeq* s). This converts a finite sequence, s, into a list and then into an item of type *SeqFinSeq*. Because the implementations all use lists, both of these conversions are trivial.

Example A proof has the following form:

To prove $\forall f :: \alpha \rightarrow \beta,\ a : \alpha,\ s :: \mathit{FinSeq}\ \alpha . \mathit{foldl}\ f\ a \Rightarrow \mathit{foldl}_S\ f\ a$, we prove

$$\mathit{foldl}\ f\ a\ s\ =\ \mathit{foldl}_S\ f\ a\ (o\ s)\ =\ \mathit{foldl}_S\ f\ a\ (\mathit{list2seqfs}\ (\mathit{fromFinSeq}\ s))$$

This can be done as following, using the implementations of *foldl* and foldl_S, in terms of $\mathit{foldl}_{[]}$ and the fact that *fromSeqFinSeq* is an inverse of *list2seqfs*:

$$\begin{aligned} \mathit{foldl}\ f\ a\ s &= \mathit{foldl}_{[]}\ f\ a\ (\mathit{fromFinSeq}\ s) \\ &= \mathit{foldl}_{[]}\ f\ a\ (\mathit{fromSeqFinSeq}\ (\mathit{list2seqfs}\ (\mathit{fromFinSeq}\ s))) \\ &= \mathit{foldl}_{[]}\ f\ a\ (\mathit{fromSeqFinSeq}\ (o\ s)) \\ &= \mathit{foldl}_S\ f\ a\ (o\ s) \end{aligned}$$

3.4.2 Sequential case

All of the above comments apply when making data and functions explicitly sequential. In addition there is a main lemma that converts functions, making sure the types match up. It is specific to this sequential case.

In particular, nested data must make sense. If a data structure has type *SeqFinSeq* α, then *all* of its elements are stored in the same processor, and therefore any data nested in it must also be sequential—α cannot be *ParFinSeq* β.

This is reflected in the main lemma for the sequential case. It uses a function *changeseq* which makes functions and expressions sequential. In the following part of this function (written in a kind of pseudo-code), the second and fourth lines deal with nested data.

```
changeseq x       | $x$ scalar $= x$
changeseq xs      | $xs :: FinSeq\ \alpha$ $=$ fstoseqfs (map changeseq xs)
changeseq f       | $f$ is a FinSeq APM function $= f_S$
changeseq (f ps)  | $f$ is a FinSeq APM function $= f_S$ (map changeseq ps)
                  where
                    $ps$ $=$ the list of parameters of $f$
```

The operation of different functions on the same data should also be taken into consideration. If one function is made sequential, then either all functions acting on the same data must be made sequential or the data must be redistributed between the functions. The latter may cost a lot in communication time, but may be worth it if the advantage of executing the function in parallel is sufficiently great. Alternatively, the other functions can be left abstract just now and dealt with later. This option may lead to extra redistribution functions, but these can often be removed after all of the functions are transformed.

Therefore *changeseq* can be extended as follows. Each expression can be transformed in three ways, corresponding to the three options above. In the first, both functions are made sequential. In the second, there is a redistribution and, in the third, the second function is left abstract.

```
changeseq ($f_S \cdot g$)  = $f_S \cdot$ (changeseq g)
                         or $f_S \cdot$ partoseq $\cdot$ (changepar g)
                         or $f_S \cdot$ fstoseq $\cdot g$
changeseq ($f \cdot g_S$)  = (changeseq f) $\cdot g_S$
                         or (changepar f) $\cdot$ seqtopar $\cdot g_S$
                         or $f \cdot$ seqtofs $\cdot g_S$
```

This definition uses the function *changepar* which is described in the next section. It is similar to *changeseq*, but makes functions parallel. The other functions, for example, *partoseq* and *seqtopar*, are data redistribution functions, described in Section 2.6.

The function, *changeseq*, is neither complete nor designed to be applied automatically, but it illustrates some key situations and can be applied by hand.

The main lemma then reads:

Rule 42 (Make functions sequential)
For any expression *e*.

$$e \Rightarrow changeseq\ e$$

The $\Rightarrow$ sign indicates that this is a vertical transformation. The two expressions aren't equal in the usual sense. However, if an appropriate observation function is applied to them as described above, then it can be seen that they carry the same meaning.

This lemma can be applied to code that uses the specification APM, once the decisions about what is to be sequential and what not have been made.

3.4.3 Parallel case

The parallel case is very similar to the sequential one. Again, the main difference lies in the transformation lemma which uses a different function, *changepar*, that makes expressions parallel instead of sequential. Unlike sequential data, parallel data can have either parallel or sequential data nested inside it, so inner nesting levels remain abstract. In addition parallel data cannot be nested inside sequential data. To avoid this, the function must not be applied to inner nesting levels when the outer levels are still abstract. The outer levels should be made concrete first. The function itself is very similar to *changeseq*:

```
changepar x      | x scalar = x
changepar xs     | xs :: FinSeq α = fstoparfs xs
                   — Unlike the sequential case, xs is left abstract
changepar f      | f is a FinSeq APM function = f_P
changepar (f ps) | f is a FinSeq APM function = f_P (map' changepar ps)
                 where
                   ps = the list of parameters of f
                   map' only applies its function to arguments at the
                   same nesting level
```

As before, when functions are composed, other functions acting on the same data must also change. They can be made parallel or the data can be redistributed between the functions. This can be expressed using similar extensions to those used for *changeseq*.

Then the lemma reads:

Rule 43 (Make functions parallel)
For any expression e such that e does not involve any data nested inside items of type *FinSeq* α.

$$e \Rightarrow \mathit{changepar}\ e$$

3.4.4 Introducing data distributions

The introduction of an explicit data distribution as described in Section 2.7 is another example of a vertical transformation. As such it bears many similarities to the transformations just described. In fact, these can be seen as special cases of this transformation because both placing all of the data in a single processor and placing one piece of data in each processor are ways of distributing it.

As before, the data distribution is indicated using data types and annotations. The functions provided by the APMs are similar to those in the abstract, sequential and parallel APMs and have similar names, such as map_{Cyclic}, but different behaviours, in order to maintain the data distribution.

In common with other vertical transformations, the lemmas replace functions from the old APM with ones from the new. This time the old APM is not the abstract one, but the parallel APM because a specific data distribution provides more concrete information about data which is already known to be parallel. The lemmas, therefore, have the general form, $f_P = f_T$, where T indicates the distribution, e.g., $T = \mathit{Cyclic}$. An example lemma, therefore, is $map_P = map_{Cyclic}$.

As before, it may be necessary to give a large lemma that transforms the whole program and applies these individual lemmas. Such a lemma would introduce data distribution and redistribution where necessary. However, this is really only needed when several different distributions are used in the same program. If the same distribution is used throughout, then no redistribution is needed.

The lemmas and the rule are proved using an observation function, as was done in the sequential and parallel cases. However this function is more complicated than the previous observation functions because the types and functions used in the data distribution APMs are not implemented in the same way as those in the parallel APM (see Section 2.7). For example, the observation function for the Cyclic distribution needs to convert a two-dimensional Cyclic structure into the one-dimensional *ParFinSeq* used in the previous stage. Items from a single "processor" have to be moved to non-consecutive locations in the finite sequence.

Again, there is a decision to be made. There are a variety of standard data distributions available, and the programmer can also design more. The choice of distribution has an impact on the efficiency of the program, and should be made with care. This choice is considered in Section 4.5.

3.5 Horizontal transformations

Horizontal transformations, unlike vertical ones, use the same APM before and after the transformation. They manipulate functions from this APM, perhaps replacing combinations of functions with other combinations from the *same* APM. As with vertical transformations, they do this using lemmas and transformation rules. However, as these don't change the model of the parallel system used, observation functions are not needed and they can be proved using ordinary equational reasoning.

This section looks at some of the horizontal transformations in the methodology. It looks firstly at transformations that tidy the code. These are useful but fairly straight forward. Another type of horizontal transformation is optimisation, discussed later in this section.

3.5.1 Tidying up

One of the most common types of horizontal transformation tidies up the program. It is a good idea to do this at various points throughout the derivation, especially after and within particularly complicated steps.

Such transformations simplify and standardise programs so that they are easier both to read and to reason about. For example, some transformation rules expect the program to be in a certain form, (e.g., with only one APM function per line), and tidying up transformations can be used to get it into this form. In fact, in certain cases, a transformation rule may actually require particular tidying up steps to be performed before it is applied (see Section 3.6.1). Tidying up may also remove excess and unnecessary code, thus shortening the program as well as making it simpler.

The actual tidying up process is accomplished, as elsewhere in the methodology, via extensible sets of lemmas. There are many such lemmas, some of which are applicable to general Haskell code and others of which only apply at certain points in a derivation. Several of these lemmas are given in Appendix A, along with some sample proofs. This section considers just a few examples to illustrate the general points.

General tidying-up lemmas

Many of the lemmas involve the use of **let** expressions. These are common in Haskell and limit the scope of function and variable definitions. It is often useful to be able to manipulate such expressions to get them into forms to which other lemmas can be applied.

For example, the following lemma allows the programmer to rearrange the equations in a **let** expression. This lemma is very simple and arises from the basic properties of **let**, but it can still be useful. Used repeatedly, it can place two equations next to each other so that they can be combined.

Prop 5 (Rearrange equations in a let expression)
For all *Fs* let equations, $x, E :: \alpha, y, E2 :: \beta$ (any α, β), E3 an expression.

```
let Fs              ==        let Fs
    x = E                         y = E2
    y = E2                        x = E
in                            in
E3                            E3
```

Other lemmas involving **let** are given in Appendix A.2.

Monadic lemmas

In the later stages of the derivation, monads are often introduced to model side-effects (see Section 2.8). Although the general lemmas can still be used to tidy up such programs, monads introduce new notation and complexities, and therefore lemmas which explicitly deal with them can manipulate the programs more directly. There are very useful, even though they can only be used in part of the methodology.

Part of the complexity when dealing with monadic code is that the side-effects may not be immediately obvious. Therefore new notation may need to be introduced, if not for the monadic lemmas themselves, for reasoning about them and proving them. An example of such a proof and the associated notation is given in Section A.6.

A set of monadic lemmas is given in Appendix A.4. Although nowhere near exhaustive, this gives a fair number of lemmas, on which others could be based. The basic properties of monads are also described there, although these are rarely used directly as they are very basic and simple. They are described in further detail in papers such as [Wad92] and some examples of monadic lemmas occur in case studies such as [BF94].

Of particular use in this methodology are the lemmas in Appendix A that manipulate state monads. These allow one to remove or move expressions in state monads provided that certain conditions hold. They can also be extended to deal with other kinds of monads.

Example lemma For example, Lemma 39 tidies up a stateful monadic program by removing unnecessary lines:

For all
- M monadic expression
- x variable such that x does not appear in M before another $(x \leftarrow F)$ expression
- E monadic expression such that E does not change any variables in the state which are used by M or any following code.

```
do x ← E          ==          M
   M
```

This can used with the monad *IOPST* (see Section 2.8), if, in addition, E does not affect either the input or the output.

Example lemma application This lemma can be applied as in the following example:

```
do x ← pst_getChar          ⇒      do x ← pst_getChar
   y ← return x                       y ← pst_getChar
   y ← pst_getChar                    pst_putStr ([x] ++ [y])
   pst_putStr ([x] ++ [y])
```

since *return* x does not affect the state, the input or the output, and the value of y is reassigned before it's used.

Example of problems in lemma application The following example illustrates a situation in which the lemma cannot be applied because the conditions do not hold. Here *pst_getChar does* affect the input.

```
do x ← pst_getChar
   y ← pst_getChar
   y ← pst_getChar
   pst_putStr ([x] ++ [y])
```

As well as lemmas which can be used for many functions, it is often useful to have ones written for particular cases, especially if they are common. These are often instantiations of more general lemmas, in which some of the side-conditions have already been checked. They therefore save the programmer from having to perform some of the side-condition checks, and they also clarify the operations which can be performed.

Example For example, the following rule is a special case of Lemma 40 for reordering stateful monadic programs. It allows us to move a *create_var* to an earlier position in a program without changing the program's meaning, and is commonly used to group all of the *create_vars* together at the top of the program, as is done in a language such as C. This is particularly useful because most *IOPST* programs contain *create_vars*.

Rule 41 (Move *create_vars* to the top)
For all $M1, M2$ IOPST expressions and x_v a variable name such that x_v does not occur in $M1$.

```
do                          == do
  M1                             x_v ← create_var X
  x_v ← create_var X             M1
  M2                             M2
```

The condition on this is simpler and easier to check than the conditions in the general case, and therefore this is easier to apply. The lemma holds because the only variable which *create_var* modifies is the one whose name it returns, i.e., x_v. Therefore the second condition in Lemma 40 is automatically satisfied.

3.5.2 Algorithmic changes

Other types of horizontal transformations can be used to change the program's algorithm, for example, to introduce optimisations to a program. While some optimisations specify more parallel details, many do not need an extra degree of concreteness. They simply rearrange the data or functions at the *current* level of abstraction, and so are horizontal. They can occur at all levels of the methodology.

No detailed examples of such transformations are given in this section because the transformations are often complicated and require a lot of explanation. Instead a few examples are discussed in detail elsewhere in this thesis. For instance, load balancing is an algorithmic horizontal transformation which is examined in Section 4.4. In addition, Sections 6.5, 6.6 and 6.8 describe an example of such transformations later in a derivation, the manipulations of the communication to optimise a pipelined calculation.

Some optimisations are relatively straight-forward to introduce to a program, but most require a degree of thought. While optimisations are used to speed up programs, in certain cases program changes meant to do this actually slow the program down. For example, load balancing can slow down a program if communication costs are very high. Extra care must be taken in such cases. Further difficulties occur because a single optimisation has several different instances that can be introduced. For example, different data can be clustered together or different tasks moved to different processors. Which instance is best depends on the particular program. Therefore there are choices which need to be made, and Chapter 4 explains how this can be done.

Such algorithmic transformations can be carried out in the same way as other horizontal transformations. Lemmas transform the program from one version to the next, perhaps replacing a function by a combination of other functions. For example, Lemma 15, when applied to a program, breaks a task, *foldl*, down into two smaller parts. This reduces the granularity size and may, in some cases, improve or allow an improvement in the load balance.

The lemmas for the algorithmic transformations may be parametrised by variables, different values of which express different instances of the optimisation. For example, the load balancing lemmas can be parametrised by the size of the task to be moved to another processor and the processor to move it to. These lemmas are given and discussed in Section 4.4.

3.6 Introducing monads

As described in Section 2.8, monads are frequently used in a program in order to model imperative features such as side-effects on the state and IO. When targeting C+MPI, this thesis uses the designed monad, *IOPST*, described in Section 2.8. It manipulates a model of the state of a parallel system, *GlobalState*, and also provides IO functions.

This section and the following section describe how monads can be introduced into a program. This is a complicated transformation with many details and therefore it is split into two main parts. This section describes the initial introduction of monads, and the addition of IO if necessary, while the following section (3.7) describes how variables and state are introduced to a program.

3.6.1 Preparation

Before monads are introduced various modifications need to be made to the program. These are basically just tidying-up steps, as in Section 3.5.1, but may be more specialised. Some of them put the program into a form to which standard lemmas can be applied, and others make subsequent transformations easier.

First of all, each function should be written as a **let** expression, instead of one involving **where**s, because such expressions are easier to transform into monadic code using Lemmas 36 and 38. This can be achieved using Lemma 4 in Appendix A.

There should also be only one APM function per line. This is not absolutely necessary at this stage as code with more than one APM function per line can still be transformed. However this is often needed later on in the derivation, for example, in the conversion to the MPI APM (Section 3.8), and it's easier to do this transformation while still in the non-monadic world as side-effects don't have to be considered. This can be done by separating out nested APM functions and tuples, using, for example, Lemmas 8 and 13.

We also need to alter code containing APM functions, such as $take_P$, $drop_P$ and $takesites_P$, that partition the set of processors. Such functions do not reflect the parallel nature of further operations very well. However they can usually be transformed into code which does. This is considered for one particular case in the Gaussian Elimination case study in Section 7.4.2.

3.6.2 Getting started

Once this preparation has been done, monads can be introduced to the program. This is initially done at the highest level only, as expressed in the following lemma:

Rule 36 (Introduce monads)
For any expression E.

$$E \quad \Rightarrow \quad \mathit{return}\ E$$

The program remains essentially the same, but it is now monadic. The change of type necessary means that this is a vertical transformation. This lemma may seem rather boring and of limited use to the programmer, but the program is now in the monadic world, and can be manipulated using only horizontal transformations.

First of all, the monads are pushed into lower layers of the program, using the following lemma:

Lemma 38 (let into do)
For any $x, E1 :: \alpha, E2 :: \beta$ (any α, β) such that
- $E1$ does not depend on $E2$
- x has no parameters

$$\begin{array}{lcl} \mathit{return}\,(\,\mathbf{let}\; x = E1 & == & \mathbf{do}\; x \leftarrow \mathit{return}\; E1 \\ \quad \mathbf{in} & & \quad \mathit{return}\; E2 \\ \quad E2) & & \end{array}$$

By applying this lemma several times to different expressions and nesting levels, it is possible to remove all **let** expressions from the program. It is not, however, always necessary to do this, especially for small **let** expressions which only define temporary intermediate values which do no use APM functions and do not need to be stored in the state, passed to the world or use values from the outside world. In addition, **let** expressions that define functions are a special case and are dealt with separately, as described in the following section.

When Lemma 38 has been applied multiple times, the program may contain nested **do**'s. This can be untidy, but the program can be easily converted to an un-nested form using Law 35.

3.6.3 Auxiliary functions

When monads are introduced to a program, they affect not only the main function, but also any local auxiliary functions. These may be defined within the main program in **let** expressions or given separately at the top level of the program. They cannot be treated in the same way as the constant **let** expressions in the last section: more care needs to be taken because functions have parameters whose types may also change.

Not all such functions need to be made monadic. However, functions which use APM functions, do input or output or manipulate the state will need to be monadic. It is not always yet clear which functions this will affect, and therefore some functions may be left until variables are introduced to the program before they are converted. This is not a problem as monadic programs can call non-monadic functions.

The way in which a function is used affects how it is converted.

In the simple general case, a function, $\mathit{local\_fn} :: T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$, with parameters ps, is called directly by the rest of the code, using, for example,

$$x \leftarrow \mathit{return}(\mathit{local\_fn}\; ps).$$

It itself is not used as a parameter to any higher-order functions.

This can be transformed relatively simply. The local function becomes:

$$\begin{array}{rcl} \mathit{local\_fn}' & :: & T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow \mathit{IOPST}\; T_n \\ \mathit{local\_fn}'\; ps & = & \mathit{return}(\mathit{local\_fn}\; ps) \end{array}$$

which can then be transformed as described above (in Section 3.6.2).

The code which calls *local_fn* also changes. The function can now be called directly instead of via a *return* function. For example, the above call becomes

$$x \leftarrow \mathit{local\_fn}'\; ps.$$

However, there are more complicated cases, when the function's parameters also become monadic. This occurs most frequently when the function is used as a parameter to a higher-order function (h.o.f.) such as *foldl*. The h.o.f. may impose constraints on the types of the function, its parameters and the other parameters involved, so that making the function monadic requires that some or all of these other variables also become monadic.

Example For example, foldl_P, a typical h.o.f., has the type,

$$(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \mathit{ParFinSeq}\; \beta \rightarrow \alpha.$$

This imposes some constraints on the types of values it can be used with. If the function parameter,

f, returns a monadic value, α becomes monadic, and all the parameters with type α must therefore also become monadic.

Consider the following piece of code (in which $s :: ParFinSeq\ \beta$):

$$\begin{array}{l} f :: \alpha \to \beta \to \alpha \\ a :: \alpha \\ x \leftarrow return(foldl_P\ f\ a\ s) \end{array}$$

Although part of a monadic program, the calculation is non-monadic. The result is simply converted into a monadic value using *return*. However, it may be useful to make f return a monadic value, for example, if f uses any variables or alters the state. Its return type would then be monadic, for example, $IOPST\ \alpha'$. In order for the types to match up, everything else that used to be of type α must now have type $IOPST\ \alpha'$ and therefore become monadic, as shown below:

$$\begin{array}{l} f' :: IOPST\,\alpha' \to \beta \to IOPST\,\alpha' \\ a' :: IOPST\alpha' \\ x \leftarrow foldl_P\ f'\ a'\ s \end{array}$$

This affects both f's parameters and a, a separate parameter of $foldl_P$. Care must therefore be taken in other places where f and a are used, and in the definition of f itself. The modified version of f accesses its monadic parameters before it continues as before:

$$\begin{array}{lll} f'\ x\ y & = & \mathbf{do}\ xval \leftarrow x \\ & & \quad return(f\ xval\ y) \end{array}$$

The second line of f' can now be transformed as in Section 3.6.2.
This change in f's parameter also affects the use of f elsewhere in the program. For example, $a \leftarrow return(f\ x\ y)$ becomes $a \leftarrow f'\ x'\ y$ where x' is the monadic version of x.

These various changes can be carried out in any order, but it is recommended that a consistent order is used to prevent confusion and omission of any of the changes. In the case studies in this thesis, the changes started with the modification of a function to allow it to access the state. The places in which this function was used were then noted, especially places where it was used as a parameter, for example to $foldl_P$ as above. Each of these was taken in turn, and the other functions and expressions that must change were discovered by examining the types as in the example above. Each of them was then transformed.

3.6.4 Adding IO

If the program is going to be used as a function, called by another program, then it does not need to use IO to obtain values or display its results—this is dealt with by the other program. However, if it will be run in isolation, either in its final version or in intermediate versions for testing purposes, then a *main* program, which deals with IO, needs to be given. This program takes values from the input, calls the function with these values as its parameters, and outputs the result.

Input functions

The *main* function can make use of specific input functions to read in the data in the correct formats. This reduces the amount of detail needed in the *main* function itself.

These input functions prompt the user for input, receive the data, convert it to the required parallel datatype, and then pass it on to the rest of the program. The prompt is not always needed—this depends on whether the data is to come from the keyboard or from a file. The data is often initially a list or sequential finite sequence, and is then converted to the parallel type. This mimics the way input is dealt with in C+MPI in which the data is read in to a single processor and then distributed across the parallel system. It is easy to modify this to cope with specific data distribution formats (see Section 3.10).

There are many variations in the input functions. Data of different types is read in in slightly different ways, and may be distributed across the system in more than one way. If user prompts are used, these are also dependent, not only on the type, but also on the program's context. Therefore the input functions are often program-specific. However, they do not usually have to be written from scratch. Example functions or templates for data of particular types can be modified with different prompts and details for each situation.

These input functions are usually based on the *IOPST* functions, *pst_getChar* and *pst_getLine*, which obtain characters and strings from the input, as described in Section 2.8.3. These input values can then be converted into the required types using the standard function, *read*, provided by the Prelude. This works for most standard types, although it may insist on particular input formats. For example, it insists that the character '%' is used to separate the numerator and denominator of a rational number. Conversion functions for other types and formats can be written based on these.

Example This can be illustrated with a simple example. The function *enter_int* fetches an integer from the input, using *pst_getLine* and *read*:

```
enter_int :: IOPST Int
enter_int = do pst_putStr "Enter a : "
               aChs ← pst_getLine
               return (read aChs)
```

More complicated example Functions for obtaining more complicated data structures can be written in a similar way. They can use recursion to obtain multiple values. For example, *enter_vector* uses a subsidiary recursive function, *enter*, to obtain a list of Floats, which it then converts into a parallel finite sequence. It takes one parameter *n* which describes the length of the input sequence. This can either be built into the program or obtained from the input using *enter_int*.

```
enter_vector :: Int → IOPST(ParFinSeq Float)
  — Reads in the elements for a vector of length n,
  — then distributes them across the processors
enter_vector n = do xs ← enter 1
                    return (list2parfs xs)
  where
    enter :: Int → IOPST [Float]
      — subsidiary function—takes in values i to n
    enter i | i > n = return []
            | i ≤ n = do
                        pst_putStr("Enter b["++ (show i) ++ "]" :)
                        xChs ← pst_getLine
                        x ← return (read xChs)
                        xs ← enter (i + 1)
                        return (x : xs)
```

Transformation rule

The addition of these input functions and a *main* function, which calls them, can be encapsulated in a transformation rule as follows:

Rule 44 (Introduce main)
For all $prog :: T_1 \to T_2 \to \ldots \to T_n \to T'$, $p_i :: T_i$ $(i = 1, \ldots, n)$, $F :: T'$.

```
pst_putStr (show (prog p1 ... pn))
  where
    prog x1 ... xn = F
⇒
main p = do let enter1 = ...
                ...
                entern = ...
            let prog x1 ... xn = F
            q1 ← enter1
            ...
            qn ← entern
            result ← prog q1 ... qn
            pst_putStr (show result)
  where
    enteri = the appropriate input function for pi.
             It may have parameters.
```

The first program produces the same output as the second, when it is given as input a sequence of values, $p_1, \ldots, p_n$. The parameter, p, to *main*, represents the number of processors in the system and can be any positive integer.

3.7 Introducing variables

Variables can be used in the Haskell program as explained in Section 2.8.2. But, before they are introduced, the model of the parallel system must be set up with the required number of processors. This is the first thing to be done.

This can be done in the *IOPST* monad using the function *start* at the beginning of *main*. The number of processors, p, is one of *main*'s parameters and can therefore be easily passed to *start*, as follows:

Rule 45 (Set up the parallel system)
For all IOPST expressions E.

```
main p = E        ⇒        main p = do start p
                                       E
```

The remainder of this section shows how variables can be introduced once the system has been set up. It is split into several sections which discuss different aspects of the transformation.

3.7.1 Local variables

Local variables usually occur frequently in a program. They are located within *main* and the local functions, including the core function itself, and are used to hold intermediate values during a calculation. They can be introduced as follows:

Rule 46 (New variable)
For all x, E :: *ParFinSeq* T for some type T,
there exists $T0 :: T$ which can be used for initialisation and x_v :: *VarFn* T such that

$$\begin{array}{rcl} x \leftarrow \textit{return } E & \Rightarrow & x\_v \leftarrow \textit{create\_var } E \\ & & x \leftarrow \textit{retrieve } x\_v \\ \text{or} & \Rightarrow & x\_v \leftarrow \textit{create\_var } (\textit{repeat}_P\ (T0 :: T)) \\ & & \textit{store } x\_v\ E \\ & & x \leftarrow \textit{retrieve } x\_v \end{array}$$

A variable with a similar name to the intermediate value is created and initialised. It may be initialised with its value straight-away, if this does not involve much calculation. Otherwise, a dummy value can be used, followed by an explicit *store*. Its value must then be retrieved before any reference to it is made. For simplicity the rule above retrieves values directly after they are stored, but the retrieval code can often be moved later in the program using Lemma 40. The creation of the variable can also be moved to the start of the function, using Lemma 41.

The above lemma only applies if x has type *ParFinSeq* T, but it is possible to use variables for scalar quantities as well, as discussed in Section 3.7.3.

The use of this lemma often creates many variables, more than are actually needed, as each is assigned to only once. This is useful if targeting a single-assignment language, such as SAC [GSSW98], but otherwise can be inefficient, especially if there is no garbage-collector. Some of these extra variables can be removed, using, for example, Rule 47 below.

Rule 47 (Remove an extra variable)
For all x_v, y_v :: *VarFn* T, V, $V2$ :: *ParFinSeq* T (for some type T), Fs, Gs IOPST expressions such that
- x and x_v don't occur in Gs,
- x, y refer to the values obtained from x_v,y_v respectively.

$$\begin{array}{lcl} x\_v \leftarrow \textit{create\_var } V & \Rightarrow & y\_v \leftarrow \textit{create\_var } V \\ Fs & & Fs[y\_v/x\_v,\ y/x] \\ y\_v \leftarrow \textit{create\_var } V2 & & Gs \\ Gs & & \end{array}$$

3.7.2 Global variables

Global variables are also important. They lie at the outer-most level of a program, and so can be accessed by any function. Some global variables, especially those holding information about the system state, occur in many programs. For C+MPI, such variables include p, the number of processors, and *pid*, the processor ids. Their values can be set using the special functions *get_size* and *get_pid*, described in Section 2.8.2. Many other global variables are program-dependent. The programmer may know which variables should be global at this stage, but in some cases, this doesn't become obvious until later. In these cases, they can be introduced in a similar way later in the derivation.

Global variables are modelled in Haskell in the same way as local variables, as described in Section 2.8.2. However, they are created before the local function definitions, so that these local functions, including the core function, can access them. They are also initialised at this point, but any calculations involving their values lie in the functions themselves or in *main* after all the local function definitions. This separates global variable declarations from the main code, following the pattern in C.

A global variable, x_v, of type T, whose value is initially set using E at the outermost nesting level, can therefore be introduced using the following rule:

Rule 48 (Add a global variable)
For all $x, E, T0$ :: T (for any type T), p :: Int^+, $F, F2$ IOPST expressions and Fs local function definitions.

```
        x = E
        main p = do F
                    let Fs    — local function definitions
                    F2
  ⇒

        main p = do F
                    x_v ← create_var (repeat_P (T0 :: T))
                    let Fs    — local function definitions
                    x ← return E
                    store x_v x
                    x ← retrieve x_v
                    F2
```

If E returns only a single value, then this value must be replicated across the processors so that they can all access it, as discussed in the next section (3.7.3).

It is sometimes the case that the value of a global variable, such as *x_v*'s is set instead in one of the local functions. In this case the extra code after *Fs* is moved into the appropriate function.

3.7.3 Duplicating values across the processors

There is some difficulty when a variable's value is scalar because *store* expects a parallel finite sequence of values, one for each processor in the system. This difficulty can be overcome by replicating the scalar value to give p copies of it, one for each processor:

Rule 49 (Duplicate a value across the processors)
For $T \neq D\alpha$ for $D = \mathit{ParFinSeq}$ or any data distribution type.
For all $E, x :: T$ (on the left hand side)

$$x = E \quad \Rightarrow \quad x = \mathit{repeat}_P\ E$$

and
For all $E :: \mathit{IOPST}\ T$, $x, x' :: T$ (on the left hand side).

$$x \leftarrow E \quad \Rightarrow \quad \begin{array}{l} x' \leftarrow E \\ x \leftarrow \mathit{return}\ (\mathit{repeat}_P\ x') \end{array}$$

This transformation changes the type of x, and so, to maintain correctness, any references to x in the rest of the code must also change. There are two main ways in which this can be done.

Normal case A parallel finite sequence representation is usually used because it expresses that there is a sequence of values, one in each processor and allows different values to be stored in different processors. Therefore, when x occurs in an expression, each of x's values should be operated on separately. This can be done using variants of the standard function map_P, which apply the given function to *each* element of the sequence, as follows:

Lemma 25 (*repeat* and *map*)
For all $x :: T$, $f :: T \rightarrow T'$ (for any T, T').

$$\mathit{repeat}(f\ x) \ == \ \mathit{map}_P\ f\ (\mathit{repeat}_P\ x)$$

And therefore

$$f\ x \Rightarrow \mathit{map}_P\ f\ (\mathit{repeat}_P\ x)$$

A more general version of this lemma is Lemma 24 in Appendix A.

Alternatively, instead of using map_P, the definition and type of f itself may change to take a parallel sequence of values.

Exceptions Sometimes, however, the parallel finite sequence representation gets in the way. When x is always constant across all the processors, it can be inconvenient and inefficient to use a sequence instead of this single value. This is the case with the standard MPI global variable, p.

This can be dealt with by selecting a single representative value from the sequence, using $!!_P$ or $head_P$. This value is then used in the program. The following lemma states that this can be done without changing the meaning of the program:

Lemma 26 (*repeat* and *head*)
For all x.

$$x \;==\; head_P\ (repeat_P\ x)$$

However, this method should not be used if x's variable, x_v, is ever modified so that it is no longer constant across the system.

A similar situation may occur later in the derivation if, at any point, the program deals with a single processor instead of the whole system. Then only one of x's values is needed. This commonly occurs after the introduction of communications, especially during input and output and point-to-point communications.

This situation can be dealt with in a similar way to the above by selecting a single value from the sequence. If processor i is used, x's ith value must be selected,using $!!_P\ (i-1)$.

3.7.4 Variables as parameters

Although variables have now been introduced, they haven't been often used directly in expressions. Their values, obtained using *retrieve*, are used in their stead. However, it can be useful to use the variables themselves as parameters to functions. If this is so, the variable is said to be passed by reference, whereas, if only the variable's value is given, it is passed by value.

Reference parameters allow the function to modify the variable's value. They also model the situation in C more closely, where reference parameters are also used for array parameters—otherwise the C's array representation would prevent all the array elements from being accessed. It is also often more efficient to pass records (structures) by reference.

In Haskell, reference parameters have type *VarFn* α. For example, the following IOPST function takes one value Integer parameter and one reference Integer parameter and returns an Integer:

$$f \;::\; ParFinSeq\ Int \rightarrow VarFn\ Int \rightarrow IOPST\ Int$$

At this point in the derivation, we may know some of the parameters which it is useful to pass by reference. For example, all values of type *SeqFinSeq* α should be passed by reference. However, some cases (e.g., scalar values) may still be unclear. If the variable is to be changed by the function then it needs to be passed by reference, but this will usually not become obvious until the program has been tidied up. In these cases, the code can be changed later in the derivation.

If a parameter becomes a reference parameter, then the type of the function, the function's code and any calls to the function must change as follows:

Rule 50 (Reference parameters)
For all f, x, y, T of appropriate types.

$$\begin{array}{ll} f \;::\; \ldots \rightarrow ParFinSeq\ T \rightarrow \ldots & \Rightarrow \quad f \;::\; \ldots \rightarrow VarFn\ T \rightarrow \ldots \\ f\ \ldots\ x\ \ldots\ = & \qquad f\ \ldots\ x\_v\ \ldots\ = \\ \quad \textbf{do}\ \ldots & \qquad\quad \textbf{do}\ x \leftarrow retrieve\ x\_v \\ & \qquad\qquad \ldots \\ E[f\ \ldots\ y\ \ldots] & \qquad E[f\ \ldots\ y\_v\ \ldots] \end{array}$$

A value passed by reference is now accessed using *retrieve* at the start of the function. Transformations can then further manipulate the variable and its value within the function.

Variables for function results

Variables can be used for a value returned by a function, as well as for values passed to a function. If this is the case, the variable to be used should be passed as a parameter to the function. Then, instead of returning the value directly, the function should store it in the variable.

Rule 51 (Returning a value from a function)
For all $x :: T$, f, ps, E of appropriate types and $x\_v :: \mathit{VarFn}\ T$ such that x_v is either a new variable (create it using *create_var* before f is called) or the variable corresponding to x.

$$\begin{array}{lll} x \leftarrow f\ ps & \Rightarrow & f\ ps\ x\_v \\ & & x \leftarrow \mathit{retrieve}\ x\_v \end{array}$$

and

$$\begin{array}{lll} f\ ps = \mathbf{do}\ \ldots & \Rightarrow & f\ ps\ x\_v = \mathbf{do}\ \ldots \\ \quad \mathit{return}\ E & & \quad \mathit{store}\ x\_v\ E \end{array}$$

Sequential functions

Some functions only operate on data which is local to one processor. They are passed local values with scalar or *SeqFinSeq* α types. For example, sequential functions, annotated with S, operate on data stored in a single processor. These are often called by map_P which applies a sequential function to each processor's data.

It is useful to be able to easily identify such functions by noting that they only have scalar or sequential parameters. However if they are passed reference parameters, these have types such as *VarFn* (*SeqFinSeq* α), and contain data for the whole parallel system. This therefore makes it harder to identify the functions that are sequential.

Therefore, it may be useful to wait before modifying such parameters. Once the program is at an individual level (see Section 3.9), the local nature of the functions is obvious, and the transformation can be carried out without causing these problems.

3.8 Moving to the MPI APM

After the introduction of monads, a language-specific APM can be used. As this is highly language-dependent, this section focuses on just one example, an APM for MPI. This allows real programs to be produced, in particular the examples in the case studies, and provides insight into this stage of the methodology in general.

The transformation to a language-specific APM is a vertical transformation, as discussed in Section 3.4, because it uses a different APM after the transformation than before. However this APM uses the same model of the parallel system so an observation function is not needed.

The current section firstly presents some transformation rules used when moving to the MPI APM, and then discusses some special cases.

3.8.1 Transformation rules

A set of rules can be used to transform a program that uses the previous general monadic APM to one that uses the MPI APM. However before they are used, the program should be tidied up as described in Section 3.5.1. The rules should then be applied to all occurrences of the APM functions.

This section looks at some representative examples of these rules to show their form, demonstrate their operation and support the case studies. Other rules are similar.

The following is an example of a rule for a basic MPI APM function:

Rule 52 (*mpi_comm_size*)
For all *x_v* :: *VarFn* α, *xs* :: *ParFinSeq* α (any α) such that
xs hasn't been shortened using, for example, *take* or *filter* (the initial data structure on which the program operates is a good choice for *xs*).

$$\textit{store x\_v } (\textit{size}_P \textit{ xs}) \quad \Rightarrow \quad \textit{mpi\_comm\_size x\_v}$$

This rule can be proved using the lemmas from Appendix A.4 and the definition of *mpi_comm_size*.

The rules for collective communication functions are similar to the following:

Rule 53 (repeat/broadcast)
For all *val* :: α, *var_v* :: *VarFn* α.

$$\begin{array}{lll} \textit{store var\_v } (\textit{repeat}_P \textit{ val}) & \Rightarrow & \textbf{do } \textit{store}_{indiv} \ 0 \ \textit{var\_v val} \\ & & \quad \textit{mpi\_bcast\_simple var\_v } T \ 0 \end{array}$$

Here *repeat*$_P$ signals that a value is broadcast across the system. Unless otherwise indicated the root processor is assumed to be 0, and so *val* is initially only stored in processor 0, before it is broadcast using *mpi_bcast_simple*. This function is a simplified broadcast function that sends one value from *var_v* to each processor. Such broadcasts are often used in input functions.

Rule 54 (scatter)
For all *xs* :: $[\alpha]$, *var_v* :: *VarFn* α.

$$\begin{array}{lll} \textit{store var\_v } (\textit{list2parfs xs}) & \Rightarrow & \textit{tmp\_v} \leftarrow \textit{create\_var } (\textit{repeat}_P \ (T0 :: T)) \\ & & \textit{store}_{indiv} \ i \ \textit{tmp\_v } (\textit{list2seqfs xs}) \\ & & \textit{mpi\_scatter tmp\_v } 1 \ T \ \textit{tmp\_v } T \ i \\ & & \textit{tmp} \leftarrow \textit{retrieve tmp\_v} \\ & & \textit{new\_vals} \leftarrow \textit{return } (\textit{map}_P \ \textit{heads}_S \ \textit{tmp}) \\ & & \textit{store var\_v new\_vals} \end{array}$$

where
- $T = \textit{SeqFinSeq}\alpha$
- $T0$ is any value of type T

Here *list2parfs* stores one value of *xs* in each processor, effectively scattering *xs*. Processor *i* is used as the root, although, as before, *i* is commonly 0. The rule contains extra code because *mpi_scatter* can be used to place multiple values in each processor, and so the receiving variable must have type *VarFn*(*SeqFinSeq* α). *var_v* doesn't have this type, and so a new, temporary variable, *tmp_v*, is created for this purpose.

A related function, *mpi_scatterv* is often also used to scatter values. It can send different numbers of elements to different processors, and needs extra variables *displs_v* and *sizes_v* to indicate which values should be sent. It can therefore be used to distribute data with more complicated distributions. For example, in the Gaussian Elimination case study in Chapter 7 it is used to implement a scatter with a cyclic data distribution.

Less standard APM functions

Other APM functions, including user-defined functions, don't correspond to any single MPI APM function, and so need to be implemented with a set of communication functions. The MPI APM functions *mpi_spt2pt* and *mpi_jointpt2pt* are particularly useful because they can express specialised individual and collective patterns of communication. The former is used when the communication is extremely limited, typically between only 2 sites, and the latter when it is larger scale, involving most or all of the processors. These functions are described in Section 2.10.

The transformation of these less standard APM functions is highly dependent on the Haskell function being transformed, because each function may use a different pattern of communication.

The function *reverse* is given as an example of such a function, together with a description of how the transformation rule was produced. This is followed by a discussion of more general cases.

Example: reverse The APM function $reverse_P$ expresses a specialised pattern of communication in which each processor sends and receives exactly one value. It doesn't translate directly to a fixed MPI APM function, but rather to *mpi_jointpt2pt* which models a combination of point-to-point MPI sends and receives. It can be converted into an instance of *mpi_jointpt2pt*. Alternatively a new function, corresponding to $reverse_P$ can be defined at the MPI APM level and implemented using *mpi_jointpt2pt*. The latter option is described in Section 2.10.5 where the function *requestrev* is defined.

However, no matter which option is chosen, it is necessary to calculate the combination of sends and receives needed to implement $reverse_P$, i.e., the parameters that *mpi_jointpt2pt* takes. These parameters are calculated by the programmer, or, when a new function is implemented at the MPI APM level, by the implementor of that function.

In the case of *reverse*, the parameters can be determined by calculating the processor to which processor i (for any i) sends its values and that from which it receives other values. These can be calculated using Lemma 23 as follows, where p is the number of processors:

$xs!!_P\ i\ =\ xs\ !!_P\ (p-(p-i-1)-1)\ =\ (reverse_P\ xs)!!_P\ (p-i-1)$

And so processor i sends to processor $p-i-1$.

$\Rightarrow\ sendfn\ i\ =\ p-i-1$

Similarly, $recvfn\ i\ =\ p-i-1$.

Therefore $reverse_P$ can be transformed as follows:

Rule 55 (reverse)

For all $xs :: ParFinSeq\ \alpha,\ xs\_v :: VarFn\ \alpha$ (any α).

$$\begin{array}{lll} xs \leftarrow retrieve\ xs\_v & \Rightarrow & mpi\_jointpt2pt\ xs\_v\ x\_v\ T\ (\lambda\, i \rightarrow (p-1-i))\ (\lambda\, i \rightarrow (p-1-i)) \\ store\ x\_v\ (reverse_P\ xs) & & \end{array}$$

where
- p is the number of processors. It can be set using: **do** $p \leftarrow retrieve\ p\_v$
  $p \leftarrow return\ (head_P\ p)$
- T is the *ItemType* corresponding to α (*ItemType* is an enumerated Haskell type described in Section 2.8.2.)

This can be written as a specialised MPI APM function, *requestrev*, which is described in Section 2.10.5.

General comments In a similar way, a function, f_P, which involves one send and receive per processor can be expressed using *mpi_jointpt2pt* with functions *sendfn* and *recvfn* as follows:

$$mpi\_jointpt2pt\ xs\_v\ x\_v\ T\ sendfn\ recvfn$$

where
- $xs\ !!_P\ i\ =\ (f_P\ xs)\ !!_P\ (sendfn\ i)$ and
- $(f_P\ xs)\ !!_P\ i\ =\ xs\ !!_P\ (recvfn\ i)$.

Functions which involve only one send/receive pairing are similar but use *mpi_spt2pt*. Other functions may however require several communications per processor. These can be transformed by splitting them into several parts, each of which corresponds to a sequential calculation, or a standard MPI APM function, such as *mpi_jointpt2pt* or *mpi_spt2pt*.

3.8.2 Special cases in the use of the MPI APM

Input functions

As described in Sections 3.6 and 3.7, the input functions distribute the input data using $repeat_P$ or type conversion. In the MPI APM, these become broadcast and scatter MPI APM operations respectively, using the rules given above.

Nested parallelism

In MPI nested parallelism can be implemented using communicators to cluster the processors into groups. Each group is then assigned a subset of the task, often in the form of a subset of the data, such as a row or column of a matrix. The groups can work on their subsets in parallel with each other, and the processors within an individual group can work on items of data from its subset in parallel. However communicators are not yet implemented in the MPI APM, so nested parallelism cannot at present be implemented in this stage of the methodology.

Implicit communication

The rules in Section 3.8.1 apply when the communication in the program at the stage prior to the MPI APM is given by APM functions. However, sometimes there is implicit communication in a function. For example, a sequential function may use a value from another processor directly, without the value being explicitly moved between the processors.

It is important to discover such implicit communication in a program and make it explicit as attempts to access remote data directly will fail. Section 7.5.4 gives an example of this process.

In general, it involves several steps:

- Examine the sequential function to see if it uses data from another processor. For example, a function with parameter i (the current processor id) uses remote data if it accesses $xs \;!!_P\; j$, for $j \neq i$.

  Sometimes the function does not have a parameter giving the current processor. In this case, the processor id can often be calculated. The example in Section 7.5.4 has a parameter *rowno* which can be used to calculate the processor id.

  In addition, the parameters, which should provide local data, should be examined. If any of them give data from another processor or if any global data is used in the function, this should be noted.

- Store the implicitly communicated data in a local variable in the current processor using $store_{indiv}$ before the sequential function is called.

- Send it to the processors in which it is used. This can be done using *mpi_spt2pt*, or, if it is used in many processors, *mpi_bcast*.

- Access the value and pass it as a parameter to the function. In certain cases, if the sequential function is run in many or all of the processors, this can be done concisely by mapping the function over the distributed values of the variable.

3.9 The individual level

MPI views the parallel system from the viewpoint of an individual processor, whereas the Haskell programs so far have viewed it from a collective level, describing the actions of the whole system. This individual level viewpoint and how it can be modelled in Haskell are described in more detail in Section 2.11. The transformation from the collective viewpoint to the individual is very language-specific, and so this section focuses on C+MPI, although some of the principles can also apply to other languages.

3.9.1 Transformation to the individual level

The transformation is guided by a set of guidelines and rules as follows. In these, *pid* is the processor id. It is assumed to have been set at the beginning of the program using *get_pid* or *mpi_comm_rank*.

Rule 56 (Transformation to the individual level)

- The system initialisation no longer needs the number of processors as a parameter, so:
  $$\mathit{mpi\_init}\ p \ \Rightarrow\ \mathit{mpi\_init}\ \text{(for any } p :: \mathit{Int}^{+}\text{)}$$

- MPI functions can be used to obtain the size and processor ids:
  $$\mathit{get\_size} \Rightarrow \mathit{mpi\_comm\_size}$$
  $$\mathit{get\_pid} \Rightarrow \mathit{mpi\_comm\_rank}$$

- *create_var*, the function which creates a parallel variable, needs only the data for the current processor as an initialisation parameter:
  For all $x\_v :: \mathit{VarFn}\ T,\ v :: T$.
  $$x\_v \leftarrow \mathit{create\_var}\ (\mathit{repeat}_P\ v :: \mathit{ParFinSeq}\ T) \quad \Rightarrow \quad x\_v \leftarrow \mathit{create\_var}\ (v :: T)$$

- Individual stores and retrieves using store_{indiv} and $\mathit{retrieve}_{indiv}$ are replaced by **if** expressions on the processor id, e.g.,
  For all *proc*, *var*, *value* of appropriate types.
  $$\mathit{store}_{indiv}\ \mathit{proc}\ \mathit{var}\ \mathit{value} \quad \Rightarrow \quad \textbf{if}\ (\mathit{pid} == \mathit{proc})\ \textbf{then}\ \mathit{store}\ \mathit{var}\ \mathit{value}\ \textbf{else}\ \mathit{return}\ ()$$

  This is used in the input functions, among other places, since input is usually done in only one processor.

- Collective communication functions are not changed. This works because the MPI APM functions were specifically designed to ease the conversion to the individual level. Their implementation changes at this level, but their names and parameters remain the same. For other languages, there may be modifications which need to be made.

- Individual communication functions, *mpi_jointpt2pt* and *mpi_spt2pt*, are separated out into calls to sends and receives. This may be different in other languages depending on their individual communication mechanisms.

  The separation can be done using the following rules:
  For all $s\_v,\ r\_v :: \mathit{VarFn}\ T$ (any type T), $\mathit{sendfn},\ \mathit{recvfn} :: \mathit{Int}^{+} \to \mathit{Int}^{+}$, $\mathit{source},\ \mathit{dest} :: \mathit{Int}^{+}$.

  $$\begin{array}{lll} \mathit{mpi\_jointpt2pt}\ s\_v\ r\_v\ T\ \mathit{sendfn}\ \mathit{recvfn} & \Rightarrow & \textbf{do}\ \mathit{mpi\_send}\ s\_v\ T\ (\mathit{sendfn}\ \mathit{pid}) \\ & & \quad\ \mathit{mpi\_recv}\ r\_v\ T\ (\mathit{recvfn}\ \mathit{pid}) \end{array}$$

  and

  $$\begin{array}{lll} \mathit{mpi\_spt2pt}\ s\_v\ T\ \mathit{source}\ r\_v\ T\ \mathit{dest} & \Rightarrow & \textbf{if}\ (\mathit{pid} == \mathit{source})\ \textbf{then}\ \mathit{mpi\_send}\ s\_v\ T\ \mathit{dest} \\ & & \textbf{else if}\ (\mathit{pid} == \mathit{dest})\ \textbf{then}\ \mathit{mpi\_recv}\ r\_v\ T\ \mathit{source} \\ & & \textbf{else}\ \mathit{return}\ () \end{array}$$

- Parallel maps (e.g., in storage functions) turn into single function applications:
  For all $x_i :: T_i,\ f :: T_1 \to \ldots \to T_n \to T$.
  $\mathit{mapn}_P\ f\ x_1 \ldots x_n \Rightarrow f\ x_1 \ldots x_n$

3.9.2 Transformation within the individual level

Individual level code can be transformed as can code at other levels in the methodology. It can tidied up, optimised, and brought closer to the target language. However, due to the present lack of a proper semantics for the individual level (see Section 2.11), these transformations cannot at present be proved correct, although this is possible in principle. Despite this problem, it is sometimes better to perform transformations at the individual level instead of at the collective level.

Individual level code is often simpler and cleaner as it does not have to contain details for manipulating the whole system. This makes certain types of code clearer to understand and manipulate and makes it easier to transform the program. For example, when different processors do different things, it may be clearer to deal with each processor separately. It is also easier to express and manipulate overlapping communications at the individual level. This *can* be done at the collective level using collective communication functions or *mpi_jointpt2pt*, but in these cases the overlaps must be calculated and then combined into the right functions. Sometimes this is useful, but it is often easier to manipulate the communications at the individual level, and leave it up to the semantics and run-time system to work out exactly which ones are done at the same time and the pattern of communication used. An example of this occurs later in this thesis in Sections 6.5.2 and 6.6.

It is also useful to bring the program closer to the target language while at the individual level. These transformations are very language-specific and therefore make more sense when close to the target, at the individual level rather than the collective. Some examples of such transformations were developed for one of the case studies in this thesis. They are fairly specific to that case study, although they can be easily extended to apply to other programs as well. Therefore they are described fully in Section 7.6.5 and only a few examples are given here.

Some of the transformations concern the replacement of implicit loops, e.g., in $maps_S$, by explicit loops, using Haskell functions, such as *for*, to mimic loops in C. For example, code containing $maps_S$ can be transformed as shown in Figure 3.4.

For all *res*, *f*, *xs*, *var_v* of appropriate types, and new names *elt* and *var*.

```
do res ← return(maps_S f xs)   ⇒   for [0..(length xs − 1)]
   store var_v res                  (λi → do elt ← return (f(xs!!i))
                                              var ← retrieve var_v
                                              store var_v (replace var i elt))

                               ⇒   for (i=0; i<n;i++)
                                          var[i] = f (xs[i]);
```

Figure 3.4: Transformation of map to a for loop. replace xs i v returns xs with its ith element replaced by v. List notation is used for simplicity.

Recursive functions can also sometimes be implemented using the function *for*. For example, functions that recurse, or can be rewritten to recurse, on an integer parameter can often be written as a loop through the values of this integer. The transformation of such functions is situation-specific, and an example is given in Section 7.6.5. This is useful as it often improves a program, although recursive functions can also be written directly in C.

There are also places in which the functions provided by the MPI APM are not as close as possible to actual MPI functions. This has been done for a variety of reasons, partly to simplify code and allow greater abstraction at earlier levels. However, at this point, they should be replaced by more concrete versions. Section 7.6.5 considers one such situation, the communication of arrays.

3.10 Data distributions in the monadic levels

Explicit data distributions can be introduced to a program during the non-monadic levels, as described in Section 3.4.4. They can be manipulated using specific data distribution APMs, which make it much easier to deal with them. They can also be used in the monadic levels, and Section 2.12 describes functions which deal with distributed data at the general monadic, MPI APM and individual levels. However, the use of specific functions and APMs affects the transformations used within and between these levels. This section looks at these effects on each of these levels in turn, and lastly at how the explicit data distributions are removed from the program.

3.10.1 Introducing monads

The introduction of monads to a program in general was considered in Sections 3.6 and 3.7. The addition of data distributions doesn't change this very much, in fact, the initial introduction of monads, described in Section 3.6, remains the same.

There are some changes when variables are introduced, but again these are mostly minor. They involve distributed variables, e.g., Cyclic variables, as described in Section 2.12.1. The transformation proceeds as before except that data with type *Cyclic* α is now stored in a cyclic variable of type $VarFn_{Cyclic}\alpha$. Such variables can be manipulated using the cyclic counterparts of the variable manipulation functions, and can be passed as parameters to functions in the same way as other variables.

Duplicated values, discussed in Section 3.7.3, still have to be duplicated across the system. However, there are now two possibilities. One value may be stored in each processor, or several can be stored in each processor, one corresponding to each data element. The former saves space, and is frequently used in target programs, but is less intuitive at this stage.

One place where there must be a change is in the input, since the initial distribution of values across the system determines the data distribution in the program. Input functions can use distribution-specific functions instead of *list2parfs* to convert the data from a list into the form to be used in the program. For example, input data can be distributed cyclicly using *makecyclic*.

Example This is illustrated in the following variant of *enter_vector*:

```
enter_vector_Cyclic :: Int → Int → IOPST(Cyclic Float)
  — Reads in the elements for a vector of length n,
  — distributing them cyclicly on p processors
enter_vector_Cyclic n p = do xs ← enter 1
                             return (makecyclic p xs)
```

As in the non-cyclic version, *enter* is used to take in the vector's values. The only change is the use of *makecyclic* instead of *list2parfs*.

3.10.2 The MPI APM

The MPI APM, described in Sections 2.10 and 3.8, provides data distributed versions of communication functions (see Section 2.12.2), which maintain the distribution. For example, $mpi\_scatter_{Cyclic}$ scatters values from a root processor to all of the processors, distributing them in a cyclic fashion. The existence of such functions means that the transformation to the MPI APM is much the same whether or not data distributions are involved. The complexities of the distribution are hidden within the provided functions. There may, however, be some details which differ.

Example For example, $mpi\_scatter_{Cyclic}$, mentioned above, has the following type:

$$\begin{array}{ll} mpi\_scatter_{Cyclic} & ::(Dyn\ \alpha)\ \Rightarrow\ VarFn\ (SeqFinSeq\ \alpha)\ \rightarrow \\ & ItemType\ \rightarrow\ VarFn_{Cyclic}\ \alpha\ \rightarrow \\ & ItemType\ \rightarrow\ Int\ \rightarrow\ IOPST() \\ instead\ of: & \\ mpi\_scatter & ::(Dyn\ \alpha)\ \Rightarrow\ VarFn\ (SeqFinSeq\ \alpha)\ \rightarrow\ Int\ \rightarrow \\ & ItemType\ \rightarrow\ VarFn\ (SeqFinSeq\ \alpha)\ \rightarrow \\ & ItemType\ \rightarrow\ Int\ \rightarrow\ IOPST\ () \end{array}$$

Care must be taken as the former involves a change in the type of the variables used, while the latter does not. It also lacks the integer parameter specifying the number of values to send.

3.10.3 Individual level

It is also useful to represent and manipulate data distributions explicitly in individual level code. Section 2.12.3 describes how this can be done, and presents some cyclic state functions for the individual level. These functions are lower-level than their collective counterparts and so ease the removal of data distributions later. However, because they *are* lower level, the transformation from the collective level is rather more complicated than the non-data-distribution version. Extra code may have to be added.

This section gives some transformation rules, focusing on the cyclic data distribution as an example. The code at the start of each rule is at the collective level, and the code at the end is at the individual level. This latter code may contain references to p, the number of processors, and *pid*, the processor identification number, usually set at the start of the program using *mpi_comm_size* and *mpi_comm_rank*. Counting starts at 0 as does the indexing of values within processors.

Cyclic variables, with type $VarFn_{Cyclic}\ \alpha$, are now seen from the viewpoint of one processor only. Each processor has a sequence of values from the variable stored in it, and so the variable is equivalent to one with type $VarFn\ (SeqFinSeq\ \alpha)$.

Function application is a key operation, given at the collective level using $map_{Distribution}\ f$. This does not transform to f directly as it does in the *SeqFinSeq* case. There are several values stored in each processor and f must be applied to each of them. This can be done using $map_S\ f$ for the cyclic distribution, or variants on this if the distribution is more complicated. It is also possible to provide an individual level version of map_{Cyclic}, but this adds little to the APM and so is not done here.

The collective versions of cyclic state functions change to individual level cyclic and non-cyclic functions. $retrieve_{Cyclic}$ and $store_{Cyclic}$ are simply transformed into individual level functions of the same name. However, $retrieve_{Cyclicindiv}$ and $store_{Cyclicindiv}$, which access single values in the state, transform differently. As they deal with only a single processor, the standard individual level functions, $retrieve_{indiv}$ and $store_{indiv}$, can be used:

Rule 57 (Individual level cyclic state functions)
For all $j :: Int^+$, *pid* containing the processor id and p the number of processors.

$$\begin{array}{l} retrieve_{Cyclicindiv}\ j \\ \quad\Rightarrow \\ \mathbf{if}(pid\ ==\ j\ `mod`\ p)\ \mathbf{then}\ retrieve_{indiv}\ (j\ `div`\ p) \\ \mathbf{else}\ return\ () \end{array}$$

$$\begin{array}{l} store_{Cyclicindiv}\ j \\ \quad\Rightarrow \\ \mathbf{if}(pid\ ==\ j\ `mod`\ p)\ \mathbf{then}\ store_{indiv}\ (j\ `div`\ p) \\ \mathbf{else}\ return\ () \end{array}$$

The relevant processor is identified using j '*mod*' p and then j'*div*' p gives the location of the element in the local sequence.

Communication functions behave in basically the same way as their non-data-distributed counterparts, except that they use new functions, *mpi_send'* and *mpi_recv'*, described in more detail in Section 2.12.3. These functions take an additional integer parameter, specifying the offset of the value to be sent. They are also used to access values in arrays.

Rule 58 (Cyclic point to point send)
For all *var_v* :: *VarFn T*1, *var2_v* :: *VarFn T*2, *source*, *dest* :: Int^+ (for some types $T1$, $T2$).

```
$mpi\_spt2pt_{Cyclicindiv}\ var\_v\ T1\ source\ var2\_v\ T2\ dest$
  $\Rightarrow$
$\mathbf{if}(pid\ ==\ source`mod`p)$
  $\mathbf{then}\ mpi\_send'\ var\_v\ T1\ (dest`mod`p)\ (source`div`p)$
$\mathbf{else\ if}\ (pid\ ==\ dest`mod`p)$
  $\mathbf{then}\ mpi\_recv'\ var2\_v\ T2\ (source`mod`p)\ (dest`div`p)$
$\mathbf{else}\ return\ ()$
```

The related function, *mpi_jointpt2pt*, transforms in a similar way but involves several communications per processor.

3.10.4 Removing data distributions

Although some languages, such as HPF [MH95], provide support for explicit data distributions, the majority, including C+MPI, the focus of this thesis, don't. Therefore functions which explicitly manipulate data distributions usually have to be removed and replaced by combinations of the ordinary state manipulation and communication functions provided by the language.

This does not have to be done at this point in the derivation. It can be done earlier, at the collective level, or even in non-monadic stages. There *are* advantages to doing it there. The transformations are easier to prove correct at the collective level because this level is easier to reason about than the individual level. In addition, at present, programs at the individual level cannot be run and so the new versions of the program cannot be checked by running them. However there are also advantages to leaving the removal of data distributions until the end of the methodology. In particular, it makes it easier to deal with and reason about data distributions in the rest of a derivation.

The conversion to the individual level has in some ways prepared the way for the removal of data distributions, because some of the explicit distributed functions, such $mpi\_spt2pt_{Cyclicindiv}$, have been replaced by more standard operations. However, there are still various changes to be made. This section presents some of these rules for the cyclic distribution.

First of all, any remaining cyclic state functions can be changed into ordinary state functions, as follows.

Rule 59 (Replacement of cyclic state functions)
For any α, s, *val*, *vals*, i, *var_v* of appropriate types and new name *old_vals*.

- $VarFn_{Cyclic}\ \alpha\ \Rightarrow\ VarFn(SeqFinSeq\ \alpha)$
- $create\_var_{Cyclic}\ s\ \Rightarrow\ create\_var\ s$
- $val\ \leftarrow\ retrieve_{indiv}\ i\ var\_v$

  $\Rightarrow$

  $old\_vals\ \leftarrow\ retrieve\ var\_v$

  $val\ \leftarrow\ return\ (vals\ !!_S\ i)$

- $store_{indiv}$ *i var_v val*
  $\Rightarrow$
  old_vals $\leftarrow$ *retrieve var_v*
  store var_v (*replace*$_S$ *old_vals i val*)
  where *replace*$_S$ *xs i x* returns *xs* with its *i*th element set to *x* and its other elements as before

- *val* $\leftarrow$ $retrieve_{Cyclic}$ *var_v* $\Rightarrow$ *val* $\leftarrow$ *retrieve var_v*

- $store_{Cyclic}$ *var_v vals* $\Rightarrow$ *store var_v vals*

As mentioned above, the point-to-point communication functions already no longer use explicit data distributions. However, the collective functions still do. These have to be converted and may require a fair amount of complex code. Therefore each function should have a transformation rule so that the programmer doesn't need to work out this code for himself. One example is outlined in the following rule.

Rule 60(Replacement of $mpi\_scatter_{Cyclic}$**)**
For any *sendbuf* :: *VarFn T*, *recvbuf*, *sendtype*, *recvtype*, *root* of appropriate types
and new names, *p*, *n*, *pid*, *tmp_v*, etc.
p_v storing the number of processors, *matrix_size_v* the matrix size and *pid_v* the processor id.

```
mpi_scatter_{Cyclic} sendbuf sendtype recvbuf recvtype root
    ⇒
    — access global variables
  p ← retrieve p_v
  n ← retrieve matrix_size_v
  pid ← retrieve pid_v
    — create new variables
  tmp_v ← create_var (X :: T)
  sizes_v ← create_var (repeat_S (0 :: Int))
  displs_v ← create_var (repeat_S (0 :: Int))
    — set up variables for sending the values
  if (pid == root) then
      do   — set up tmp_v with the cyclic layout
        scattervals' ← retrieve sendbuf
        scattervals ← return (makecyclic p(seqfstoList scattervals'))
        store tmp_v (concat_S scattervals)
          — set up size and displs
          — the first few procs may have one more item than the others
        store sizes_v (cat_S (replicate_S (n`mod`p) (n`div`p + 1))
                             (replicate_S (p − (n`mod`p)) (n`div`p)))
        sizes ← retrieve sizes_v
        let displvals = (0 : [displvals !!_S i + (sizes !!_S i)
                             | i ← [0..n − 2]])
        store displs_v (toSeqFinSeq displvals)
  else return ()
    — send the values
  mpi_scatterv tmp_v sizes_v displs_v sendtype recvbuf recvtype root
```

mpi_scatter_cyclic can be implemented using *mpi_scatterv*. Different numbers of values are sent to different processors, as processors near the start of the system may receive one more value than processors at the end. This requires two additional variables, *sizes_v* and *displs_v*, to specify how many values are to be sent and from where. The values of these variables must be calculated using index modulo arithmetic. In addition, the variable from which the results are to be scattered must be set up so that values to go to processor *i* are all next to each other. Therefore an extra variable, *tmp_v*, is created and the values stored in it using *makecyclic*.

3.11 Target code

Finally, the code must be transformed into the target language so that it can be run in parallel on the target machine. By its nature, this transformation is very dependent on the target, so that little can be said about it in general. Nevertheless, this section makes a few general comments before examining one particular target, C+MPI, which is used throughout this thesis.

3.11.1 General comments

It isn't, in general, possible to prove the final transformation to the target correct because many target languages, including C+MPI, lack a formal semantics, and it is outside the scope of this thesis to provide one. However, for some target languages, such as GpH, such semantics *do* exist, and such proofs can be given.

To lessen the effect of this general "unprovability", the final transformation is kept as simple and straightforward as possible. This helps to convince programmers of the correctness of this transformation even if it cannot be proved correct. This is achieved by introducing more detail to the final Haskell level of the methodology, thus bringing this level close to the target language. If this final Haskell stage gets too complicated, an extra stage in the methodology can be introduced, as suggested in Section 2.2.

Sometimes, however, it is not possible to remove all of the non-trivial transformations. In such cases, great care should be taken to ensure the correctness of the transformations even if they cannot be proved formally.

It is also sometimes useful to further optimise the program within the target language even though such optimisations cannot usually be reasoned about formally. For example, some optimisations only become apparent at the target level, and others cannot be expressed in the Haskell code as it stands. The system is still in its preliminary stages, and the Haskell models are not as expressive as they could be. In these cases, transformation rules can be specified, though not proven, within the target language.

3.11.2 C+MPI

C+MPI [KR88, MPI97] is the target language used in this thesis, and the transformations and stages presented in this chapter and in Chapter 2 are geared towards it. In particular, the structure of the Haskell program in its final stages mostly follows that of a C+MPI program. However, there are some differences which are necessary to allow some features to be modelled. The main difference is that the Haskell program contains local function definitions within *main* whereas C does not allow nested functions. This was done in the Haskell to allow local functions to manipulate the state monad, but this is not necessary in C. When the program is transformed to C, these functions are moved out of *main* and then treated in the same way as ordinary C functions.

Other parts of the Haskell code correspond to set parts of the C program. For example, variables and constants declared and defined in Haskell before the local function definitions correspond to global variables in C. The declaration for a Haskell function,

$$f :: T_1 \to T_2 \to \cdots T_n \to T, \text{ with } f\ p_1\ \ldots\ p_n = \ldots,$$

transforms to the form, `T f (T1 p1, ..., Tn pn)`

where `Ti` is the C type corresponding to the Haskell type T_i.

The Haskell *main* function becomes the C `main` function once these declarations and functions have been removed.

This can be expressed in the following rule which converts the structure of a program. Individual lines can then be transformed using the rules and guidelines in the next section.

Rule 61 (C+MPI top level)

```
main = do start
          variable declarations
          local function definitions
          code
⇒
 variable declarations
 local function declarations

 main(int arc, char *argv[])
   {
     int errcode;

     errcode = MPI_Init (&argc, &argv);
     code
     errcode = MPI_Finalize ();
   }

 local function definitions
```

3.11.3 C+MPI individual transformation rules

Table 3.2 summarises a selection of the transformation rules for different kinds of Haskell and C+MPI expressions. It is not complete, but gives a flavour of the transformations. They can be seen in practice in the case studies in Section 5.10 and (to a lesser extent) in Section 7.7.

Some of these rules involve a change from Haskell types to their C or MPI equivalents. These are indicated in the table by adding the letters C or MPI respectively to the type variable.

Many of the functions also involve reference parameters. In these cases, the address of a variable should be passed to the function instead of the variable itself. This is indicated in the table by attaching an apostrophe or prime to the variable, with the following meaning:

- `x'` = `&x` (the address of `x`) if x is a scalar value
  = `x` if it's an array.

Variables

In C, variables are created when their types are declared. If required, they can also be initialised at this point. In addition, array variables must be given a length or size after their name. This is expressed using an extra variable, `Y`, in rule 4 in the table. Variables need not be retrieved explicitly, so Haskell code which does this produces no C code. Variable names can also be changed. There is no longer any need to use two names for each variable, one for the variable and one for its value, because C makes no distinction between these in its code. In Haskell, x_v was used for the variable, and x for its variable. These can now both be replaced by `x`.

Arrays

Arrays, or finite sequences, are manipulated in Haskell using APM functions such as map_S and $index_S$, and values are stored in them using auxiliary functions $replace_S$ and $replaceslice_S$. $replace_S\ xs\ i\ x$ replaces the ith value of xs with x and leaves the rest of the array the same. $replaceslice_S$ is similar but it replaces a whole subsection or slice of xs starting at the ith location with another (usually shorter) array. The same behaviour is expressed in C using direct array indexing and `for` loops.

	Haskell expression	C+MPI version
	General expressions	
1	$x \leftarrow$ *return E*	`x = E;`
2	*return x_v*	`return x;`
3	... *f parameters* ...	`...f(parameters) ...`
	Variables	
4	*x_v* $\leftarrow$ *create_var* $(X :: T)$	`TC xY = X;` where `Y` = `[SIZE]`, if T = *SeqFinSeq* α = `*nothing*`, otherwise
5	$x \leftarrow$ *retrieve x_v*	*nothing*
6	*store x_v E*	`x = E;`
	Arrays	
7	*return*$(xs\ !!_S\ i)$	`xs[i]`
8	*xs* $\leftarrow$ *retrieve xs_v* *store xs_v* (*replace*$_S$ *xs i x*)	`xs[i] := x;`
9	*xs* $\leftarrow$ *retrieve xs_v* *store xs_v* (*replaceslice*$_s$ *xs i ys*)	`for (j=0;j<size of ys;j++)` `xs[i+j] := ys[j];`
	MPI functions	
10	*mpi_comm_size n_v*	`MPI_Comm_size(MPI_COMM_WORLD, &n);`
11	*mpi_comm_rank pid_v*	`MPI_Comm_rank(MPI_COMM_WORLD, &pid);`
12	*mpi_bcast x_v T root*	`MPI_Bcast(x', size, TMPI, root,` `MPI_COMM_WORLD);` where `size` is a variable giving the number of elements in *x_v*. Its value should be updated every time this number changes.
13	*mpi_scatterv x_v sizes displs* *T y_v T2 root*	`MPI_Scatterv(x', sizes, displs,` `TMPI, y', SIZE, T2MPI, root,` `MPI_COMM_WORLD);`
14	*mpi_send x_v T dest*	`MPI_Send(x', count, TMPI, dest,` `tag, MPI_COMM_WORLD)`
	Standard Haskell functions	
15	*store x_v* (*foldl*$_S$ *f a xs*)	`x_v = foldl(f,a,xs,xs_size);`
	Input/output functions	
16	*a* $\leftarrow$ *enter_int*	`a = enterint;`
17	*xss* $\leftarrow$ *enter_dmatrix n* 0	`enterdmatrix(xss,n);`
	Loops	
18	*for* $[a..b]$ $(\lambda\, i \rightarrow M)$	`for (i=a;i` $\leq$ `b; i++)` `M'`

Table 3.2: Transformation rules for converting expressions from individual level Haskell code using the MPI APM into C+MPI. These hold for all values of the appropriate types.

MPI functions

MPI APM functions model MPI ones with similar names, and use similar parameters. However, the MPI APM simulation is not complete, and there may be extra parameters in the C+MPI version. For example, the Haskell version does not as yet model communicators. These are used in MPI to divide the set of processors into subunits that can carry out operations independently of each other. At present, instead, all the MPI APM operations are transformed to use the standard communicator, `MPI_COMM_WORLD` that represents the set of all the processors.

Other parameters that aren't given explicitly in the Haskell version include `count` which tells the system how large the value being sent is. This can be calculated using $size_S$ *var* if *var* is a sequence. Another parameter is the integer `tag` which can be used to ensure that the right sends and receives match up. Each send/receive pairing can be given a distinct generated integer. This could be modelled in later versions of the Haskell MPI APM.

The individual send and receive MPI functions actually have many variants corresponding to buffered, synchronous and ready communication. However, this thesis only considers the standard send. Future work may consider how the other variants can be modelled in Haskell.

Other functions

Many sequential Haskell functions, both standard and user-defined do not correspond to standard C+MPI functions. They can be defined locally in Haskell and then transformed into C functions using the techniques in this chapter. Common functions can be written in advance and kept in a library.

For example, Haskell Prelude functions, such as $foldl_S$, $take_S$ and $drop_S$, are common in Haskell programs. C+MPI versions of these can be written and called as illustrated in rule 15 in Table 3.2, although array iterations are often used instead of the recursion and pattern-matching common for list functions, as discussed in Section 3.9.2. Such functions will usually need to know the size of the array in addition to the other parameters. This can be done by introducing an extra size variable, *xs_size*, for each array, *xs*. This is updated every time the number of elements in the array changes.

Input and output functions also need to change. C+MPI versions of them can be written using the standard C IO functions, *scanf* and *printf*, and can be called as in the table.

In addition, some C versions of standard functions don't work in exactly the same way as the Haskell versions. For example, some have a smaller domain, or produce different results on non-standard input. This initially caused problems in the Gaussian elimination case study when `%(mod)` did not work as expected for negative numbers. A new `mod` function was needed that called the standard function for positive numbers, but was implemented in a different way for negative ones.

Loops

Standard C loops, such as `for` and `while` can be simulated in Haskell as in Rule 18 in the table, in which `M'` is the C+MPI transformation of *M*. Section 3.9.2 gives an example of the use of the Haskell function *for* to simulate a C `for` loop. It can be implemented using recursion and pattern-matching on its list argument.

3.11.4 Optimisations

As mentioned above (in Section 3.11.1), it is possible to further transform the code even after it's in the target language, both to optimise the code and to tidy it up. Section 5.10 gives two examples of such transformations for a particular case study. As an additional example, here is the rule for removing unnecessary initialisations. Such initialisation may occur because the Haskell function *create_var* assumes that an initial value is given for each new variable, but this is not always necessary in C.

Rule 62 (Remove initialisations)
For all types T, variables x and values y of type T.

```
T x = y;        ⇒        T x;
```

Provided • the value of x is updated before the first time it is used.

3.12 Summary

This chapter has presented and discussed key transformations in the methodology for a derivation targeting C+MPI. As well as the basic transformations needed for deriving any program, it has considered the use of explicit data distributions in programs. Many of the transformations are detailed, and it is unlikely that a programmer would want to apply them all by hand, but tool support or automation could be added in the future.

This chapter has focused on how the transformations are carried out, rather than on the choice of which transformations should be done. This is the topic of the next chapter.

Chapter 4

Making Decisions

4.1 Introduction

There are many decisions to be made in the course of deriving a parallel program: the placement of the data, the division of the tasks, and the choice of optimisations to use, to name but a few. Some of these decisions are fairly straightforward and may be determined by the target language and architecture, but many are not so easy to make.

This chapter describes how such decisions fit into the framework of the APM methodology, described in Chapters 2 and 3, and examines how they are made, using a mixture of methods, including cost models. This is supported by the examination of three decisions in detail: the choice of what to do in parallel and what sequentially; static load balancing, a parallel optimisation; and the choice of data distributions. Key methods by which these decisions are made, the criteria to take into account, and the incorporation of the results of the decisions are discussed. All three decisions are illustrated in the case studies that follow in Chapters 5 to 7.

4.2 Decisions in the methodology

When transforming a program from one stage to the next, decisions have to be made. One has to decide which APM to go to and which instance of the transformation to apply. Some of these decisions are determined easily, using key factors such as the target. For example, when the target language is C+MPI, the language-specific APM to be used must be the MPI APM. Other choices are more difficult, and there may be several relevant factors to consider. For example, the distribution of the data affects the placement of the tasks, the communications needed, and the cost of global communications.

For such decisions, the transformation contains three main parts. First of all the decision must be specified clearly so that we know precisely what we're talking about. Once this is done, the decision can be considered and made. Finally the result of this decision must be incorporated into the program. The remainder of this section considers each of these parts separately.

4.2.1 Specifying a decision

The specification of a decision depends, among other factors, on the type of the decision. Decisions can be categorised into two main types, corresponding to the two main types of transformations.

Horizontal transformations

Horizontal transformations use the same APM before and after the transformation, and manipulate functions belonging to that APM. Decisions in such situations correspond to a choice of possible program modifications, each of which can be expressed in a transformation lemma or set of lemmas. The decision can be specified by giving the set of possible lemmas.

Sometimes it is useful to express a range of different possibilities in a single lemma. To do this, the lemma can be parametrised, different values of the parameters representing different possibilities.

For example, in load balancing, one must decide where to move excess work from heavily-loaded processors to. This data movement can be described by a function, f, using the convention that data in processor i is sent to processor $f i$. By using f as a parameter in the load-balancing lemmas, they represent a variety of possible redistributions of the data.

Such lemmas can be applied to the program before the decisions are made to produce a generalised program that includes variables. This program can be used in a variety of situations, but it cannot yet be run, as it includes free variables. The values of these variables need to be chosen and substituted into the program for it to run. Alternatively the parameter values can be decided first, and substituted into the lemmas which are then applied to the program.

Vertical transformations

The other main kind of transformation is the vertical transformation, which replaces the APMs used by the program. The decision in this case determines which APM or APMs are used in their place. For example, Section 4.3 discusses the replacement of the abstract APM with either the parallel or sequential APM. The decision can be specified by listing the set of possible new APMs.

Such decisions are closely related to the horizontal ones. Although they concern a choice of APMs, each APM has a set of transformation rules and lemmas from the old APM to the new. The decision therefore corresponds to a choice of a set of lemmas and rules to apply, as in the horizontal case.

Variables can also sometimes be used to represent the choices in this situation, by encapsulating the properties of new APMs. For example, a block-cyclic data distribution (see Section 2.7) can be used with a variety of block sizes. With certain sizes, it represents the ordinary cyclic and blockwise distributions. A variable can be used for this block size in the lemmas and program.

4.2.2 Making a decision

Once the decision is precisely specified, it can be made. There are many possible methods of doing this and many criteria to be taken into account.

One common method is the use of *cost models*, which are used directly within this thesis in Sections 6.3.2 and 7.3. This thesis also uses a more unusual method involving the specification of a set of constraints. Although related to other work, this contains some novel aspects. Both of these methods are described in more detail later in this section, along with some factors apart from cost which should be taken into account. Their use in and relation to other work is described in Section 8.1.4 of Chapter 8 later in this thesis. That section also describes some of the other methods that are commonly used to make decisions in parallel programming. These include general guidelines, profiling, and the use of prewritten packages. The section also discusses the advantages and disadvantages of each method and the situations in which it works well.

This thesis allows different methods to be used in different situations, at the discretion of the programmer. For example, a cost model can be used when its formalism and increased accuracy are useful, and more ad-hoc methods when they are not.

Cost models

A cost model gives estimates of the cost of a program. Usually this is the time that it takes, but sometimes other measures are used. For example, Nesl's cost model calculates the work (total number of operations executed) and depth (longest chain of sequential dependencies) in a computation [Ble96]. To calculate the cost, a cost model takes a set of parameters. These usually describe key costs on the target machine and other relevant factors, such as the number of data elements and the number of processors. However in some cases it is also possible to parametrise the model with descriptions of the decision which is to be made.

A cost model can be employed to help make decisions by estimating the cost of the program under different choices. These costs can then be compared to determine the best choice.

There are a variety of cost models available. Skillicorn in [Ski99] explains some of the difficulties involved and some of the approaches taken in these models. A cost model suitable for the methodology is given in Section 6.3.1, and a family of cost models corresponding to the APMs has also been considered in [ORR01]. There are different levels of abstraction at different stages of the methodology, and so it can be useful to use a variety of cost models within a derivation.

Cost models are considered in more detail in Section 8.1.4 of Chapter 8, which also discusses how particular cost models could be used within the methodology.

Cost constraints

As well as being used directly to compare the costs of different alternatives, cost models can be used in more oblique ways. It is sometimes possible to make decisions by identifying properties which one wants satisfied, perhaps because they will produce a correct program, give good performance or will be useful in other ways. Rather than using cost estimates directly, these properties can be manipulated algebraically or in some other way, perhaps using cost models.

Section 8.1.4 describes ways in which this method is used in related work, and the method is discussed further in Section 4.4. The latter considers an example, the choice of some load balancing details. In some cases, the best possible speed-up of the program can be achieved by a perfect load balance without excess communication. This is therefore chosen as the constraint to be satisfied.

As in that case, the constraints can often be written algebraically, involving the parameters whose values are to be found. By manipulating these equations and inequations, values for them can be obtained.

Sometimes a mixture of cost models and constraints is used, especially if the constraints by themselves are not restrictive enough. Explicit cost concerns can be used to pick a value from the set of possibilities allowed by the constraints, or, if their costs are not too different, a value can be picked at random from this set.

Although cost models use details from the target machine, this doesn't mean that all the details of the target need to be known whenever a decision is made. If this were so, then the target would be fixed right from the very first decision, and the approach wouldn't be very portable. Instead the information available about the target increases as the derivation progresses so that early stages can be shared by many targets.

Therefore, decisions in the early stages only use information about a few aspects of the target. They may partition targets broadly into classes of machines with parameters within certain ranges, instead of using exact parameter values. For example, Section 4.3 examines the order of magnitude of p, the number of processors, rather than its exact value.

Other factors

Although cost is an extremely important factor in the design of parallel programs, it is not the only one. There are other criteria which should be taken into account when making a decision.

In particular, the target of the derivation should be considered. This will affect the cost, and indeed most cost models are parametrised with details of the target machine. However, it also affects the ease with which certain parallel features can be incorporated into the final program, and, in some cases, whether they can be incorporated at all. For example, few languages provide explicit support for nested parallelism.

Other factors that may be taken into consideration include the ease of maintaining the final code, the initial and final layout of the data and the data dependencies. Section 4.3.2 discusses some of the factors in one example decision.

4.2.3 Incorporating the result of a decision

Once the decisions have been made, the chosen lemmas are applied to the program to transform it as in Chapter 3. If these lemmas are parametrised, then the chosen values of the parameters can be substituted into the lemmas, and the modified lemmas then applied to the program. Alternatively the program can be transformed using the parametrised lemmas, perhaps before the decisions are completely made. The values of the parameters are substituted into the program itself afterwards.

4.3 Parallel and sequential implementations

This section presents a common example of a decision in the methodology—the choice of whether to make a function parallel or sequential (see Section 3.4.1). This is connected to the distribution of the data on which the function operates. The function can be executed in parallel if the data is spread over more than one processor, but must be done sequentially if the data is all in a single processor. Therefore this decision can be viewed as a more abstract version of the choice of data distributions in Section 4.5. It is made first and simplifies the choice of distributions later.

The transformation is vertical, and therefore the decision is a choice between new APMs, the sequential and parallel APMs, both described in Section 2.6. Each APM has a corresponding set of lemmas, described in Section 3.4.1. These can be applied to the program to carry out the transformation.

The choice can be made by considering several criteria, one of which is, as usual, cost. As in the general case, a cost model can be used to help make this decision. However it can be hard to find a useful model because this decision usually occurs near the start of a derivation before much information about the target has been given. It is more useful to consider other related criteria that have a large impact on the cost, and don't require precise details of the target system. Some may, however, require general information about the target, and some decisions about it may have to be made at this point.

4.3.1 Data dependencies

Data dependencies occur when one piece of code requires results from another piece. The two pieces cannot be executed in parallel, even if they are placed on different processors. The later code must wait until the first finishes its calculation, making the required results available. Thus data dependencies force a sequential order of execution between the two pieces of code. The code and their data can still be stored in different processors, but this produces no direct parallel benefit and incurs communication costs. This might suggest that such code and data be placed in the same processor, and this is sometimes true.

However if the same data or the results of these calculations are used by another function, it may not be true. If the new function can be executed in parallel, then storing the relevant pieces of data in separate processors may speed up its execution. This data distribution can be achieved by redistributing the data between the parts of the program, or by using the latter distribution throughout, despite the resultant suboptimal performance of the first function. This situation is similar to that in general data distributions (see Section 4.5.1) when deciding which data distribution to use for each part of a program.

The decision can be made using cost models as in Section 4.5, although, as mentioned above, these are often too concrete to be used at this stage in the derivation. It may be better to examine the program in more general terms, noting the parallelism of each part of the program, and its size in relation to the other parts, in order to determine the relative benefits and costs of placing the data in separate processors.

To do this, and to determine where decisions need to be made, it is important to know the program's data dependencies. Some work has been done in this area in fields such as compilation (e.g., [AK87]), and similar methods can be used. Section 7.2.6 uses a simpler, more ad-hoc method—it observes the data dependencies between APM functions. These functions are standard and so their individual data dependencies can be analysed in advance, as shown in the examples in Table 4.1.

Function	Data dependencies
foldl f	In general, linear
	If *f* is associative, tree-structured
map	None
take	None

Table 4.1: Examples of data dependencies in abstract APM functions

For user-defined APM functions, including application-specific functions, it may be necessary to go back to the mathematics or the application area underlying the problem in order to determine data dependencies and uncover the parallelism.

As shown in Table 4.1, the data dependencies may depend on the parameters to functions in the program. For example, when the function parameter *f* is associative, the data dependencies in *foldl* reduce from linear to tree-structured, allowing a limited parallel execution. Otherwise each part of *foldl* must be executed before the next in a sequential fashion. This introduces further branching into a program's derivation. If the program fulfils the necessary conditions, the derivation may proceed in one way, but otherwise it has to go another way. One should keep note of such conditions that have been introduced for purposes like this.

4.3.2 Other criteria

Other criteria, noticeably key features of the target architecture and the program context, also affect the cost of the program.

For example, the communication/calculation cost ratio affects the best grain size of parallelism, and thus the number and size of parallel tasks. This ratio need not be known exactly, but can be broadly estimated from the type of the target machine. Tightly-coupled machines have low ratios and therefore can implement fine-grained parallelism efficiently, while networks of workstations and Beowulf clusters have high communication costs, and work best with large-grain parallelism.

The number of available processors is also important, because it determines how much of the specified parallelism can actually *be* implemented in parallel. Although the exact number of processors is often not known, the type of target machine can give a rough idea of it, or at least, of its order of magnitude.

If there are less processors than there are tasks in the outer nesting levels, implementing further nested parallelism can be a bad idea. It reduces the degree of parallelism which can be implemented at the outer levels, and although it is possible for the parallelism in the inner levels to make up for this loss, it often does not. The benefit and cost of communication at each level needs to be examined.

The initial and final data layouts also affect the decision. If the function is used as part of a larger program, and sometimes if it's a stand-alone program, these may be specified. They affect the choice of data distributions (see Section 4.5), and also the current decision as they specify which data is stored in different processors. Different data layouts can be used within the program but this requires a redistribution after the input or before the output. This is often expensive but may be worth-while if the increase in efficiency within the program is sufficiently high.

However, cost is not the only factor to be considered, and in fact, it is not always the most important factor. The complexity and readability of the final code may be important, for example, for maintenance purposes. A drop in efficiency may be acceptable to avoid an overly confusing program.

For the current decision, this primarily concerns nested parallelism. Nested parallelism increases the complexity of a program, often by a significant amount, unless a language, such as Nesl [Ble95] or Nepal [CKLP01], is used that supports it. It may therefore be desirable to avoid nested parallelism even if its use will increase the efficiency of the program. If so, only one level of parallelism is allowed in the program, and the remaining levels must be sequential.

4.4 Static load balancing

Static load balancing is an example of an optimisation that can be introduced to a program. It requires decisions to be made about how the tasks are split into smaller pieces and how these pieces are moved around. It can be expressed in lemmas that transform the program and are parametrised by the functions for splitting and moving the tasks. These functions can be simplified for particular programs. A full length example of this decision is given in Section 5.4 in the map-triangle case study.

4.4.1 Load balancing in general

Load balancing is a well-known technique designed to improve the efficiency of a program in cases in which the amount of work done by different processors varies widely. In such cases, processors with a small amount of work or load have to wait for those with a large load, thus wasting processing power. Load balancing transfers work from processors with a large load to those with a small one, so that the load on the processors is better balanced, as illustrated in Figure 4.1. Sometimes, before this can be done, large tasks have to be split into smaller pieces that can be executed separately. A more comprehensive description of load balancing can be found in [WA99].

There are two main kinds of load balancing—dynamic and static. This thesis focuses on static load balancing in which the movement of the tasks is determined before run-time. Static load balancing is effective when the program has a regular structure and predictable task sizes and the relative speeds of the processors are known in advance. In other cases, a better solution is *dynamic load balancing* which reorganises the tasks at run-time. This is usually supported by the run-time system rather than the program itself, and so is not so immediately relevant to this thesis.

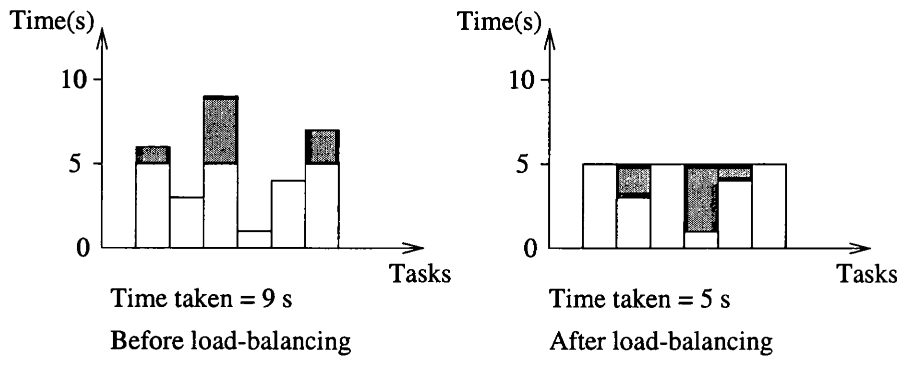


Figure 4.1: An example of load balancing

Related work

Load balancing is an active research field. There has been a lot of work on the design of algorithms for balancing tasks between processors. However much of this has focused on dynamic load balancing rather than static (see, for example, the load balancing papers in [SKGF01]).

Static load balancing determines the movement of tasks before run-time, but the actual movement can occur at different times. The tasks can be moved at the beginning of the program when the data is first distributed, or part-way through the program. The latter is better if the load distribution changes between parts of the program. Much of the work on static load balancing has been on the initial distribution of work, rather than redistribution in the middle of a program. The choice of a good data distribution, if there is one that fits the load pattern, can often help here. For example, cyclic and block-cyclic distributions work fairly well for triangular load patterns.

Alternatively, algorithms can be used. These vary and are usually restricted to a particular class of programs or architectures. They are often related to other algorithms, such as graph partitioning ones, since the program can be modelled by a directed acyclic graph (see, for example, [GY92]). These algorithms use measures of load from a cost model. In many of the algorithms that deal with initial distribution, the additional costs of communication to obtain the load balance are not very high, and using one particular distribution over another may not affect communication costs by a significant amount. Therefore some algorithms (e.g., [GA00]) don't take communication into account.

4.4.2 Specifying the decision

Load balancing is not concerned with changing the level of abstraction, but rather with the details of the algorithm. It performs extra operations on the data within the same abstraction level. Therefore the same APM is used before and after the transformation and this is a horizontal transformation.

It is expressed by a set of lemmas that can be used to change the program. It is possible to have a different such set for each possible load balancing algorithm, but it is simpler and the lemmas are easier to manipulate if fewer parametrised sets are used, as described in Section 4.2.1. Each set applies in a different situation, for example, with certain classes of problems. Each of its lemmas captures a key part of the transformation, and different values of its parameters give different instances of the transformation.

To identify what these lemmas should be, we need to examine the main actions performed in load balancing. Firstly the tasks are split into smaller pieces. If there are enough suitably sized tasks in the program already, this need not be done. Then some of these pieces are moved from the heavily loaded processors to the lightly loaded ones. All of the tasks are then carried out, and the results are finally moved back to their original processors, where they are combined with the results from the other pieces of the task to give the final results.

There are two key actions here—the splitting up of the tasks and their movement. One lemma can be written for each of these, as is considered below.

Division of the tasks

If there are several tasks in each processor, some of these can be moved to other processors to help balance the load. However, if there's only a few tasks in each processor, it may be necessary to divide them into smaller pieces which can be calculated separately, on different processors.

This process can be expressed in a parametrised lemma. This lemma varies according to the problem, although it may be possible to combine different cases into a single (probably rather complicated) lemma. For the sake of simplicity, this section examines one fairly general instance of the lemma, and shows how it can be made more concrete to apply to a particular program.

In some instances, the task can be divided by dividing up the data. If we want to divide the task in two, the lemma would therefore have the following form:

Lemma Form 1 (Division of a task and data into two parts)

$$F\,X \;=\; G\,(F1\,(\mathit{fst}(\mathit{divide}\ X)))\,(F2\,(\mathit{snd}(\mathit{divide}\ X)))$$

Here $F\,X$ is the original task—the function F operates on data X. This data is split into two parts using *divide*, and these parts accessed separately using *fst* and *snd* as above. These can then be operated on independently by the functions $F1$ and $F2$, which are related to F, and the results combined by G to give the same result as before.

This lemma form can be instantiated for the particular tasks in the program, taking care that the resultant lemma holds. For example, in the map-triangle case study in Chapter 5, the task is *foldl f a xs*, for *xs* a column of a matrix. This can be split up using the following instance of Lemma Form 1. The lemma is a combination of Lemmas 15 and 17.

Lemma 16 (Split *foldl* using *take* and *drop*)
For any $f :: \alpha \rightarrow \alpha \rightarrow \alpha$ an associative function, $a :: \alpha$ a unit of f, $xs :: [\beta]$, $m :: Int^+$.

$$\mathit{foldl}\ f\ a\ xs \;==\; f\ (\mathit{foldl}\ f\ a\ (\mathit{take}\ m\ xs))\ (\mathit{foldl}\ f\ a\ (\mathit{drop}\ m\ xs)).$$

This sets $G = f$, $F1 = F2 = \mathit{foldl}\ f\ a\ (= F)$, and $\mathit{divide}\ ys = (\mathit{take}\ m\ ys,\ \mathit{drop}\ m\ ys)$ in the Lemma Form 1. It can be proved using structural induction.

The lemma is now parametrised by a single variable, m, which determines the relative sizes of the new tasks.

Movement of the data

Once there are sufficient tasks that can be executed independently, some of them can be moved to other processors. In some cases, this can be done using a standard redistribution, but often less standard data distributions are needed.

As before, this process can be expressed using lemmas, with different lemmas for different situations. If the task is divided up by a division of the data, as above, the lemma for moving a set of tasks P_op on the set of processors has the following form:

Lemma Form 2 (Movement of a task)

$$P\_op\ XS \ = \ (move'.\ P\_op2.\ move)\ XS$$

Where • $XS :: ParFinSeq\ \alpha$ or uses a parallel distribution type, such as $Cyclic\ \alpha$.
• $move'_P \circ move_P = id$.

This rearranges the data XS on the set of processors using $move$, performs a task P_op2 (related to P_op) on it, and then moves the results back to the original processors using $move'$.

P_op and P_op2 are often $map_P\ F$, which executes F on each processor. Therefore an example instance of Lemma Form 2 is Lemma 18. This lemma works for map, map_S and map_P, but here we are only interested in map_P:

Lemma 18b (Move data in map_P)
For any function $F :: \alpha \rightarrow \beta$ (any α, β) and list permutations $move_P$ and $move'_P$ such that $move'_P \circ move_P \ = \ id$.

$$map_P\ F \ == \ move'_P \ \circ \ (map_P\ F) \ \circ \ move_P$$

This is parametrised by the related variables $move_P$ and $move'_P$, which describe the movement of the data.

4.4.3 Making the decision

Once the relevant lemmas for the program have been chosen, the decision involves finding the best values of their parameters. For the example lemmas above, these parameters are m and $move_P$—once $move_P$ is known, $move'_P$ can be calculated.

It is tempting to choose these parameters to give a perfect load balance, but this is not always a good idea because the communication time entailed by the load balancing must also be considered. If this is high, then the time taken to rearrange the data may be higher than the time saved by the improved load balance, so load balancing is not a good idea. On the other hand, if the tasks are expensive enough, then the load balanced program is also the cost optimised one. In general, there are also cases between these two extremes, when moving a few elements can produce a sufficiently good load redistribution to improve the program, but achieving a perfect load balance would require a prohibitively large amount of communication.

Parameter values can be calculated in each situation. This can be done by analysing the costs explicitly using a cost model. However, in some situations, this is not necessary. For example, if the task costs are known to be much higher than the communication costs, then the extra communication will be worth it and a perfect load balance will give good performance. In this case, we can use the load balance as a cost constraint as described in Section 4.2.2. We specify that the program must have a perfect load balance, and use no more communication than is absolutely needed to achieve this.

These properties can then be written algebraically, and used to manipulate the number of elements in each processor as shown in the map-triangle case study in Section 5.4.

4.5 Data distributions

As described in Section 2.7, data distributions specify the way in which the data is stored across the set of processors. Different distributions produce different amounts of parallelism and communication in the program, thus affecting its efficiency and complexity. Certain distributions may also balance the load on the system, as mentioned in Section 4.4. It is therefore important to make the choice of distribution correctly. This section looks at how this choice can be made and incorporated into a program. Examples of choosing data distributions are given in Sections 6.3.2 and 7.3.3.

4.5.1 Choosing data distributions

Difficulty in the choice of data distributions arises because this choice cannot be made in isolation for each part of the program. The efficiency of the whole program depends not only on the times taken by its individual parts but also on the connections between them. If consecutive parts use different data distributions, a redistribution of the data is needed between them. Depending on the amount of communication this requires and the communication speed, this can be significant.

Therefore, when choosing the data distributions for a program these redistributions need to be taken into consideration. The distributions that optimise each individual part may require an excessive amount of redistribution between them, and therefore not produce an efficient program.

The different possible combinations should be considered. In general, for a pair of program parts, these are:

1. Use the optimal distribution for each part and redistribute the data between the parts.
2. Pick one of the optimal distributions and use it for both parts.
3. Pick one sub-optimal distribution and use it for both parts. This may achieve decent, although suboptimal performance for each part and avoids a redistribution.
4. Use different sub-optimal distributions for each part. This is only useful if the redistribution required is cheaper than that necessary for the optimal distributions.

For a large program with many parts and many possible distributions, this produces a very large number of possibilities to consider.

As the aim of this thesis is not to produce novel methods for choosing data distributions, this section considers only how existing methods, some of which are mentioned below, can be incorporated into the methodology, and illustrates this in simple situations.

Related work

There has been a fair amount of research on choosing data distributions.

Some of this has been algorithm-specific. For a particular algorithm running on a particular machine or type of machine, certain data distributions are better than others. For example, [Rob90] gives a data partitioning technique for Gaussian Elimination in systolic computing. These data distributions can be worked out by examining the structure of the problem and its relationship to the architecture, or by using cost models or profiling different possibilities.

This has produced good implementations of functions and modules, but doesn't address the difficulties when these parts of the program are put together, as described above.

To cope with this difficulty, some work has used constraints on the distributions as mentioned in Section 8.1.4. However most work has focused on the use of cost models to model the costs of different parts of the program for the different possible data distributions. These can then be combined to determine which data distribution to use for each part. This could, of course, be done by a brute-force technique considering every possible combination, but this is rather inefficient. There are several more sophisticated algorithms. A few of these from systems related to this thesis are mentioned below.

To [To95] and Fradet and Mallet [FM00] work in the context of skeletons. Skeletons are fixed sets of higher-order functions or program "templates" for expressing the parallelism in a program, as described in Section 8.1.3. To gives a formalised algorithm for choosing from a small fixed number of possible distributions. Each skeleton has an implementation for each distribution. Fradet and Mallet use complexity analysis and symbolic cost analysis and comparison to choose from a wider range. However this involves fairly lengthy symbolic calculations, and they suggest the use of a symbolic maths package such as Maple [Kam99] to support them.

Determining data distributions is also a key step in TwoL (see Section 8.2.3). It uses an algorithm based on dynamic programming, as described in [RR00]. This is designed to work in a programming language in which computation is separated into modules. Runtime functions are again used to calculate the costs of each module.

A standard cost model such as BSP can also be used as shown in [SDPZ98], which gives an algorithm based on the shortest paths algorithm.

There are many methods for calculating data distributions, and any of them could be used with the methodology on this thesis. The case study given in Chapter 7 uses the costs of the parts of the program to determine the distributions. However it side-steps the issue of which algorithm to use to combine the costs by considering a simple case with only two main program parts, so that the brute-force method is effective. The aim of this study is to illustrate how such methods can be incorporated into the methodology and how the resultant data distributions can be expressed and reasoned about, and so a more complicated example is not necessary.

Modules

Because global and local optimality can be at odds with each other, it is usually important to consider the whole program at once. This makes it difficult to develop functions and subprograms in isolation, for example, in reusable modules.

One option is to develop these functions only up to a certain stage in the derivation. The data distributions are then incorporated when the context is known. This can be useful when the function is only to be used a few times, but this negates, to some extent, the advantage of using a module as the programmer must do an additional, fairly large amount of work each time the module is used.

An alternative is to produce a set of related modules, one for each data distribution. This requires more initial work, but aids reuse.

4.5.2 Specifying the decision

Introducing a data distribution changes the level of abstraction in a program. It makes the details of the parallelism more concrete. This is therefore a vertical transformation, and can be encapsulated in a change of APM.

The new APMs provide functions that manipulate data with specific data distributions, as described in Section 2.7. Section 3.4.4 shows how the program can be transformed to use these APMs, and provides the appropriate transformation rules. These rules are particularly simple because the APMs provide the same functions as the more general parallel APM.

The decision can therefore be specified by listing the possible target APMs. In some cases, these APMs and their corresponding functions and transformation lemmas can be parametrised. For example, the block-cyclic APM may be parametrised by its block size. Then values for these parameters must also be chosen.

4.5.3 Making the decision

As suggested in Section 4.2.2, it may be possible to choose the data distributions using desired properties or constraints on the final program such that satisfying these produces efficient code.

For example, placing data that is used in a single task on a single processor often reduces the communication required, and may be a suitable constraint. However there are many interacting factors and most existing methods use algorithms that compare the costs directly.

To do this, a cost model is needed. There are a variety of these available, several of which are discussed in Section 8.1.4. Different models are suitable depending on the target system and the current stage in the derivation. In early stages, not many target details are known, and general cost models should be used, while machine-specific models can be used late in the derivation. Alternatively, some methods for choosing distributions use their own specialised cost calculation methods (e.g., [RR00]). This thesis uses a version of the cost model proposed by Rauber and Rünger in [RR95], which is described in Section 6.3.1. It works for general message-passing systems.

Using such a model, the cost of each part of the program with each possible data distribution and the costs of redistributions can be estimated. These can then be combined using a method such as those described in Section 4.5.1 above. The result may depend on machine parameters, as in the example in Section 7.3, and so it may be necessary to introduce more details about the target machine at this stage. Specific values are not needed since ranges of values can be used in their place.

4.6 Summary

This chapter has described how decisions about a program can be made in the course of deriving that program using the methodology presented in this thesis. Decisions are made when the relevant transformations are applied to the program, and can be made using various methods, some of which were described in this chapter.

The chapters of this thesis so far have considered the methodology as a whole. The following chapters illustrate this methodology in action in some case studies.

Chapter 5

Case Study: Map-triangle

5.1 Introduction

This section presents a simple case study using the methodology given in the previous chapters. In this study the derivation is worked through from the initial mathematical specification to the final target program, giving the details of all the stages and transformations in between. This case study was used to motivate many of the points in the previous chapters, and it allowed them to be tried out in practice, a process which often led to changes and refinements in the methodology. It also elaborates, illustrates and clarifies the more general explanation given in the previous chapters.

This particular case study is a preliminary one and is therefore fairly simple in order to allow the basic mechanics of the methodology to be investigated. In particular, it does not use specific data distributions, but assumes that there are as many processors as is necessary. A case study involving data distributions can be found in Chapter 7.

The example used is the summation of the columns of a matrix, focusing on the case in which the matrix is triangular and the summation operation (which need not be addition) is associative and has a unit, so that the process can be speeded up by applying load balancing. The choice of an appropriate data distribution would have helped here, but an explicit load balancing was used instead to illustrate some decision-making methods. This example is chosen to be simple enough to keep the programs readable while still illustrating the key issues. The target language, as in the rest of this thesis, is C+MPI.

The main stages of the derivation are given in Sections 5.2 to 5.10, some timing results are discussed in Section 5.11, and the general layout of the case study and other general observations are given in Section 5.12.

Notation In this chapter, successive versions of the program are given together with notes about the rules or lemmas are used to derive them. These rules and lemmas can be found in Appendix A. Some of the versions of the program may contain the notation "...". This indicates that the code in that place is the same as or similar to that in the previous version.

5.2 Initial stages

5.2.1 Specification of the problem

The case study used in this chapter is the reduction of the columns of a matrix X, using an operator $\oplus$, which is assumed to be a relatively expensive computation. The matrix contains a sequence of n

columns $X_i = [x_{0,i}, \ldots, x_{n-1,i}]$ for $0 \leq i < n$, where column i contains $i+1$ non-zero elements. The aim is to compute the vector of column sums: $s_i = \bigoplus_{j=0}^{n-1} x_{j,i}$ for $0 \leq i < n$.

This mathematical specification can then be transformed into Haskell using the techniques in Section 3.3.2, by representing the matrix as a finite sequence of columns. Each column is a finite sequence of data values of type α. The reduction of the columns using $\bigoplus$ translates into *foldl*1 $\oplus$, and the *for*, which says that the sum is done for each column, translates into *map*. Writing f instead of $\oplus$, the function is:

Version 1

$$\begin{aligned} &\mathit{maptri} :: (\alpha \to \alpha \to \alpha) \to \mathit{FinSeq}(\mathit{FinSeq}\ \alpha) \to \mathit{FinSeq}\ \alpha \\ &\mathit{maptri}\ f \ = \ \mathit{map}\ (\mathit{foldl}1\ f) \end{aligned}$$

5.2.2 Specification of parallelism

This Haskell specification is abstract: it doesn't say what is done in parallel and what sequentially. In order to produce an implementation from it, and to enable the application of efficiency techniques, we specify what things are parallel and what have to be sequential in the program. This can also be done later in the study (see Section 2.6).

As described in that chapter, the annotation, P, indicates that the corresponding data is stored in several processors and the annotation, S, indicates that it is stored in a single processor. The lemmas discussed in Section 3.4.1 show that functions with these annotations on the appropriately typed sequences are equivalent to functions without them on the general *FinSeq* type.

The decision of what is parallel and what sequential can be made by examining various factors as explained in Section 3.4.1.

We are interested in exploring load balancing in a program. This can only be carried out on parallel data, not on sequential data. Therefore the columns of the matrix must be stored in different processors. This is also efficient since there are no data dependencies between the operations on the columns, and therefore each operation can be carried out completely independently. This can be expressed using the *ParFinSeq* APM and P annotations, as explained in Section 2.6. Using the transformation rules in Section 3.4.1, the program is now:

$$\begin{aligned} &\mathit{maptri} \ :: (\alpha \to \alpha \to \alpha) \to \mathit{ParFinSeq}(\mathit{FinSeq}\ \alpha) \to \mathit{ParFinSeq}\ \alpha \\ &\mathit{maptri}\ f \ = \mathit{map}_P(\mathit{foldl}1\ f) \end{aligned}$$

Within each column, the data elements can either be stored in a single processor or spread across several processors. Load balancing can be carried out in either situation. Therefore it is possible to remain abstract about the parallelism of this level, continuing to use the *FinSeq* data type. However, this makes the necessary cost calculations for load balancing much harder, and achieves little. Deciding on the parallelism of this level now does not significantly restrict the derivation.

Since there is only one level of parallelism involved and only a few functions, the choice of whether to make this level parallel or sequential can be made non-formally, using the suggested criteria in Section 3.4.1.

Some of these criteria require knowledge of the language or architecture model aimed for. This is not yet known, so we focus on the other criteria here.

If this level is parallel, then the *fold*s will have to execute in parallel. Due to the data dependencies involved, this requires a lot of communication, and is not as efficient as the parallel implementation of *map* in the outer level of parallelism. Therefore available processors should be used for the outer level *map* in preference to this level. This indicates that it may be a good idea to make the second level sequential.

$$\begin{array}{ccccc}
x_{00} & x_{01} & x_{02} & x_{03} & x_{04} \\
 & x_{11} & x_{12} & x_{13} & x_{14} \\
 & & x_{22} & x_{23} & x_{24} \\
 & & & x_{33} & x_{44} \\
 & & & & x_{44}
\end{array}
\qquad
\begin{array}{ccc|cc}
x_{00} & x_{01} & x_{02} & x_{03} & x_{04} \\
 & x_{11} & x_{12} & x_{13} & x_{14} \\
 & & x_{22} & x_{23} & x_{24} \\ \hline
 & & & x_{33} & x_{34} \\
 & & & & x_{44}
\end{array}
\qquad
\begin{array}{c|c|c|cc}
\underline{x_{00}} & x_{01} & x_{02} & x_{03} & x_{04} \\
x_{34} & \underline{x_{11}} & x_{12} & x_{13} & x_{14} \\
x_{44} & x_{33} & x_{22} & x_{23} & x_{24} \\ \hline
 & & & & \\
 & & & &
\end{array}$$

(a) **(b)** **(c)**

Figure 5.1: Map-triangle matrix **(a)** Initial triangular matrix; **(b)** partitioning; **(c)** load-balanced computations

Therefore, applying Lemmas 43 and 42 from Section 3.4, the program becomes:

Version 2

$$\begin{aligned}
&\textit{maptri} && :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \textit{ParFinSeq}(\textit{SeqFinSeq}\ \alpha) \rightarrow \\
& && \quad \textit{ParFinSeq}\ \alpha \\
&\textit{maptri}\ f && = \textit{map}_P\ (\textit{foldl1}_S\ f)
\end{aligned}$$

5.3 Preparation for load balancing: Special case

If the matrix, *xss*, is triangular, as in Figure 5.1(a), the program can be improved. In this case, Version 2 is inefficient because it has a bad load balance. This can be clearly seen using a simple cost analysis that takes account of both the computation and communication costs. Let T_f be the time needed for a processor to apply f to an argument stored in the local memory, assuming that T_f does not depend on the value of the argument. Let $T_{com} = T_0 + k \cdot T_c$ be the time required by a total exchange operation, where k is the size of the largest message.

In the *maptri* program, processor i requires time $T\ (\textit{foldl}_S\ f\ a\ [x_{0,i},\ \ldots,\ x_{i,i}]) = i \cdot T_f$. The poor load balance can be clearly seen, since the processors' computation times vary from 0 to $(n-1) \cdot T_f$, and the time for the whole program depends on the maximum time required by a processor. By spreading the work evenly, the computation time could be cut in half, although we must also consider the costs of the communications introduced to balance the load.

The first step in load balancing (see Section 4.4) is to divide the tasks up into smaller pieces, and a natural idea is to split the folds over a long list into separate folds over shorter pieces (Figure 5.1(b)). The partial folds can then be rearranged so that each processor has about the same amount of work (Figure 5.1(c)). This can be done using Lemma 15 in Section 5.4, but only if f is associative and has a unit.

It is not strictly necessary for f to have a unit, but if it doesn't then it is awkward to express the computation in SPMD-style. Some of the processors would have to send data while other processors do nothing, and some processors would sum two columns while others sum only one. Therefore this chapter assumes that f has a unit, a.

This is an example of a point at which the derivation may branch depending on the conditions under which we are working. If the matrix was not triangular, then this load redistribution could still be carried out and produce a correct program, but possibly at a reduced speed. If f was not associative, then the optimisation could not be used, and if it didn't have a unit, then extra complications would have to be introduced. However, in the following, we assume that f is associative and has unit, a.

However, before the load balancing is introduced, the unit, a, needs to be introduced to the program:

$$\begin{aligned} &maptri\ f\ xss = map_P(foldl1_S\ f)\ xss \\ &\quad\text{(Lemma 14)} \\ &\qquad = map_P(foldl_S\ f\ a)\ xss \\ &\qquad\quad \text{where } a \text{ is a unit of } f \\ &\quad\text{(Lemma 1)} \end{aligned}$$

Version 3

$$\begin{aligned} &maptri &&:: (\alpha \to \alpha \to \alpha) \to \alpha \to ParFinSeq(SeqFinSeq\ \alpha) \to \\ & && \quad ParFinSeq\ \alpha \\ &maptri\ f\ a\ xss &&= map_P(foldl_S\ f\ a)\ xss \end{aligned}$$

5.4 Load balancing

Now the program is in an appropriate form to introduce load balancing, as described in Section 4.4. Firstly the program is written with a general load balance. This introduces some parameters into the program, that represent details of the load balance, such as the number of elements to move and where to move them to. The program can then be analysed to determine good values for these parameters. This can be done using cost models, either by analysing the program directly or by picking out desired properties and determining how they can be satisfied.

5.4.1 A general load balance

As mentioned before, the first step in load balancing is to divide the tasks up into smaller pieces, which can then be rearranged. In many cases the tasks are already small enough, but in this case study, there is only one task per processor, so it has to be divided.

Lemma 15 permits the splitting of *foldl*, since f is associative. This lemma works for *foldl*, $foldl_S$ and $foldl_P$, but in this example we need the $foldl_S$ version, as follows:

Lemma 15b (Split $foldl_S$)
For any $f :: \alpha \to \alpha \to \alpha$ an associative function, $a :: \alpha$ a unit of f, and $xs,\ ys\ ::\ SeqFinSeq\ \alpha$.

$$foldl_S\ f\ a\ (xs\ {+\!\!+}_S\ ys) == f\ (foldl_S\ f\ a\ xs)\ (foldl_S\ f\ a\ ys).$$

To apply this, we have to divide up each of the columns. This can be done using the version of Lemma 17 for *SeqFinSeq*.

Lemma 17b (Divide up a list)
For all $xs\ ::\ SeqFinSeq\ \alpha$, $m :: Int^+$.

$$xs\ ==\ take_S\ m\ xs\ {+\!\!+}_S\ drop_S\ m\ xs$$

Each processor splits its data, and hence computation, into two parts, $work1\ =\ take_S\ m\ xs$ and $work2\ =\ drop_S\ m\ xs$. The parameter m can be calculated so as to minimise the total time; by leaving m as a variable, we are describing a family of related algorithms.

The program becomes:

$$\begin{aligned}
\mathit{maptri}\ f\ a\ \mathit{xss} &= \mathit{map}_P(\mathit{split\_fold}\ f\ a)\ \mathit{xss} \\
&\quad \textbf{where}\ \mathit{split\_fold}\ f\ a\ \mathit{xs} = f\ (\mathit{foldl}_S\ f\ a\ (\mathit{take}_S\ m\ \mathit{xs})) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{foldl}_S\ f\ a\ (\mathit{drop}_S\ m\ \mathit{xs})) \\
&= \quad \text{(Lemma 20)} \\
&\quad \textbf{let}\ \mathit{work}1 = \mathit{map}_P\ (\mathit{take}_S\ m)\ \mathit{xss} \\
&\qquad\ \ \mathit{work}2 = \mathit{map}_P\ (\mathit{drop}_S\ m)\ \mathit{xss} \\
&\qquad\ \ \mathit{res}1 = \mathit{map}_P\ (\mathit{foldl}_S\ f\ a)\ \mathit{work}1 \\
&\qquad\ \ \mathit{res}2 = \mathit{map}_P\ (\mathit{foldl}_S\ f\ a)\ \mathit{work}2 \\
&\quad \textbf{in} \\
&\quad \mathit{map}2_P\ f\ \mathit{res}1\ \mathit{res}2
\end{aligned}$$

The excess work *work*2 can be offloaded from processors with too much work. In other processors, $\mathit{work}2 = []$. This is done using a communication operation move_P with an inverse move'_P which can be used later to return the partial fold results to the right processor. This is expressed in the *ParFinSeq* version of Lemma 18:

Lemma 18b (Move data in map_P)
For any function $F :: \alpha \to \beta$ (any α, β) and any permutations move_P and move'_P such that $\mathit{move}'_P \circ \mathit{move}_P = \mathit{id}$.

$$\mathit{map}_P\ F == \mathit{move}'_P \circ (\mathit{map}_P\ F) \circ \mathit{move}_P$$

The following program moves the excess work, *work*2, to processors that have room for it. Each processor then computes two fold results, and sends the second, *res*2, back to its original processor, where it is combined with the first result, *res*1.

$$\begin{aligned}
\mathit{maptri}\ f\ a\ \mathit{xss} &= \textbf{let}\ \mathit{work}1 = \mathit{map}_P\ (\mathit{take}_S\ m)\ \mathit{xss} \\
&\qquad\ \ \mathit{work}2 = \mathit{map}_P\ (\mathit{drop}_S\ m)\ \mathit{xss} \\
&\qquad\ \ \mathit{res}1 = \mathit{map}_P\ (\mathit{foldl}_S\ f\ a)\ \mathit{work}1 \\
&\qquad\ \ \mathit{res}2 = \mathit{move}'_P(\mathit{map}_P\ (\mathit{foldl}_S\ f\ a)\ (\mathit{move}_P\ \mathit{work}2)) \\
&\quad\ \textbf{in} \\
&\quad\ \mathit{zipWith}_P\ f\ \mathit{res}1\ \mathit{res}2
\end{aligned}$$

This program contains parameters, m, move_P and move'_P (which is the inverse of move_P). Their values now need to be determined.

5.4.2 Analysis

The next step is to calculate values for m and move_P so as to to minimise the total time. As explained in Section 4.4.3, in general, the best performance may not result from an optimal load balance. It depends on the costs of calculation and communication.

Values for m and move_P can be calculated for each situation, but one calculation suffices to demonstrate the techniques. We assume that $T_f \gg T_{com}$ on the target architecture for all message sizes k, such that $0 \leq k \leq n+1$. Then the total cost is minimised by achieving a perfect load balance, i.e. a distribution satisfying the following two sub-goals:

1. Since a processor's work load is proportional to the number of elements it holds, and the total number of elements is $\frac{1}{2}n(n+1) \approx n\frac{n}{2}$, each processor should have about $\frac{n}{2}$ elements in a perfect load balance.

2. A perfect load balance avoids unnecessary communication. Therefore overloaded processors should only send data, under-loaded processors should only receive, and no processor should do both.

Together these form a set of desired properties, which can be written in equation form and manipulated to determine good values for the parameters as indicated in Section 4.4.3.

In the following, p is the index permutation function of $move_P$, i.e. $(move_P\ xs)!!i = xs\ !!\ p\ i$. It is used to calculated $move_P$. The other variable is m, the number of values kept in a processor. The symbol n is the number of columns in the matrix and also, since one column is stored in each processor, the number of processors. Processor and data element indices are counted from 0. In addition, the symbol # is used as a short-hand for the *length* function.

Approximate calculation

If a totally optimal solution is not needed, then one can be approximated as follows:

By Property 2, over-loaded processors don't receive any data. Therefore in order to satisfy Property 1, and have about $\frac{n}{2}$ elements, they must keep about $\frac{n}{2}$ elements. An over-loaded processor i keeps $\#(map\ (take\ m)\ xs) = min\ (m,\ \#xs) = m$ elements, where xs is the processor's contents. Therefore $m \approx \frac{n}{2}$.

An under-loaded processor, on the other hand, sends 0 values, by Property 2. The final number of elements in an under-loaded processor i is therefore approximately $i +$ the number of received values
$\approx i +$ the number of elements sent from processor $p\ i$ using *move*
$\approx i + (p\ i - m)$.
This is also about $\frac{n}{2}$ by Property 1.
$\Rightarrow p\ i \approx \frac{n}{2} - i + m \approx \frac{n}{2} - i + \frac{n}{2} = n - i$
So $move_P$ is roughly a *reverse*, and $move'_P = inverse\ reverse = reverse$.

Accurate calculation

For an accurate calculation, the number of elements remaining in processor i after load balancing is written in terms of m and p as follows:

$$\begin{aligned}
\text{task size } i &= \#((map_P(take_S\ m)xss)!!i) + \#((move_P(map_P(drop_S\ m)xss))!!i) \\
&= \text{(By the definition of } p) \\
&\quad \#((map_P(take_S\ m)xss)!!i) + \#((map_P(drop_S\ m)xss)!!(p\ i)) \\
&= \text{(By the definition of } map) \\
&\quad \#(take_S\ m(xss!!i)) + \#(drop_S\ m(xss!!p\ i)) \\
&= \text{(By Lemmas 21 and 22} \\
&\quad \text{standard size properties of } take \text{ and } drop) \\
&\quad \min(m, \#(xss!!i)) + \max(0, \#(xss!!p\ i) - m) \\
&= \text{(The matrix is triangular, so } \#(xss!!i) = i + 1) \\
&\quad \min(m, i+1) + \max(0, p\ i + 1 - m) \qquad (5.1)
\end{aligned}$$

The first term in this expression corresponds to data kept and the second to data received. The expression can be simplified further by considering the sending and receiving processors separately.

A processor i sends values iff

$$\begin{aligned}
&\#(drop\ m(xss!!i)) > 0 \Leftrightarrow max(0, i+1-m) > 0 \\
\Leftrightarrow\ & i+1 > m \Leftrightarrow i > m-1 \qquad (5.2)
\end{aligned}$$

By Sub-goal 2, a sending processor does not receive any values, so equation (5.1) becomes:

$$\text{task size } i = \min(m, i+1) + 0 = m, \ \text{ when } i > m-1 \tag{5.3}$$

Similarly, a receiving processor i doesn't send any values. Therefore by 5.2, $i \leq m-1$, and, if the processor receives values from processor, $p\ i$, then $p\ i > m-1$. Using these properties, equation 5.1 becomes:

$$\begin{aligned}\text{task size } i &= \min(m, i+1) + \max(0, p\ i + 1 - m),\\ &= i + 1 + p\ i + 1 - m\\ &= p\ i + i + 2 - m \end{aligned} \tag{5.4}$$

Property 1 can now be applied to the first expression (5.3), to give:
$\frac{n}{2} - 1 \leq m \leq \frac{n}{2} + 1$ where m is an integer.
Both n '*div*' 2 and n '*div*' 2 + 1 solve this. However, larger values of m cause less values to be communicated, so all else being equal, $m = n$ '*div*' 2 + 1 is chosen.

Property 1 can also be applied to expression 5.4 to give:

$$\begin{array}{rcl} \frac{n}{2} - 1 \leq & p\ i + i + 2 - m & \leq \frac{n}{2} + 1\\ \Rightarrow \frac{n}{2} + m - i - 3 \leq & p\ i & \leq \frac{n}{2} + m - i - 1\\ \Rightarrow \frac{n}{2} + (n\text{`}div\text{`}2) - i - 2 \leq & p\ i & \leq \frac{n}{2} + (n\text{`}div\text{`}2) - i \end{array}$$

<u>For n odd</u>, $n\text{`}div\text{`}2 = \frac{n}{2} - \frac{1}{2}$ (and $m = n\text{`}div\text{`}2 + 1 = \frac{n}{2} - \frac{1}{2} + 1 = \frac{n+1}{2}$), so

$$\begin{array}{rcll} n - i - \frac{5}{2} \leq & p\ i & \leq n - i - \frac{1}{2} & \\ \Rightarrow n - i - 2 \leq & p\ i & \leq n - i - 1 & \text{since } i, p\ i \text{ are integers.} \end{array}$$

This leaves two possibilities for p. These can be checked to see which of them are permutation functions. Checking boundary conditions, we have:

$$\begin{array}{lll} \underline{n-i-2} & i = n-1 & \Rightarrow n - i - 2 = -1 \quad X\\ \underline{n-i-1} & i = 0 & \Rightarrow n - i - 1 = n - 1 \quad \surd\\ & i = n-1 & \Rightarrow n - i - 1 = 0 \quad \surd \end{array}$$

A similar calculation for n even produces similar results.
Therefore $p\ i = n - i - 1$, and p is the index permutation function of $move_P$, so:
$(move_P\ xs)!!i \ = \ xs!!(p\ i) \ = \ xs!!(n - i - 1) \ = \ (reverse_P\ xs)!!i$
using Lemma 23.

So $move_P \ = \ reverse_P$
$\Rightarrow \ move'_P \ = \ inverse\ reverse_P \ = \ reverse_P$

5.4.3 Program transformation

The values of the parameters can now be substituted into the program to give the following version of the program.

Version 4

```
maptri f a xss =
    let m = (size_P xss)`div`2 + 1
        work1 = map_P (take_S m) xss
        work2 = map_P (drop_S m) xss)
        res1 = map_P (foldl_S f a) work1
        res2 = reverse_P (map_P (foldl_S f a) (reverse_P work2))
    in
    map2_P f res1 res2
```

5.5 Profiling

The previous calculations should produce a well-balanced program. However it is often useful to check if this has indeed happened by running or simulating the resulting program. This is not necessary and may be expensive, but it is reassuring and helps to find any errors which may have occurred in the calculations. These can then be corrected before time and effort are wasted in future development.

Profiles are often particularly useful because they give more information than just the function's run-time. For example, GpH's profiling tools display the number and state of tasks in the parallel system. This is useful for load balancing because one can see how much parallelism is being employed at each point in time.

In this case study, it was assumed that f is expensive compared to communication. To keep the example simple, this was achieved by forcing the function to do an expensive computation before doing an addition. In practice, an expensive summation function would be used in the first place.

In the methodology GpH is a good candidate for profiling because of its similarity to Haskell. The APM functions can simply be implemented using GpH instead of sequential Haskell, and the remainder of the code remains as before, as mentioned in Section 2.3.6.

Example For example, map_P can be implemented in GpH as:

```
map_P :: NFData β => (α -> β) -> ParFinSeq α -> ParFinSeq β
map_P f fs = toParFinSeq (parMap rnf f (fromParFinSeq fs))
```

where *parMap* is defined using the standard strategy *parList* as follows:

```
parMap strat f xs = map f xs `using` parList strat
```

parList executes each element of its list in parallel using the given strategy. In the above example, the given strategy is *rnf*, which reduces its argument to normal form. See [THLPJ98] for more details about GpH and strategies.

Profiles were then generated for the *maptri* function on 8*8 matrices both before and after load balancing, using GUM [THMJP96], a parallel implementation of GpH, on 8 processors of a 32-processor Beowulf cluster. These profiles are shown in Figures 5.2 and 5.3. It should be noted that GpH is not location aware, as previously mentioned in Section 2.3.6. So this is not a totally accurate representation. The threads are not allocated to specific processors as in SPMD, but to any available

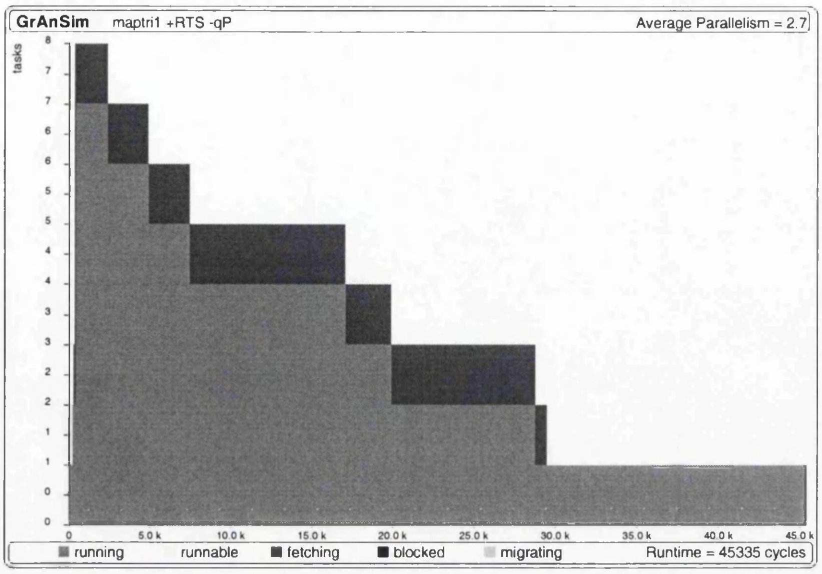


Figure 5.2: Activity profile of maptri before load balancing

processor. However, the idea of the load balance can be obtained by examining the total amount of parallelism in the system at each point in time. This is given by the mid-grey shade in the figures.

In Figure 5.2, the bad load balance can be clearly seen: the load on the processors rapidly decreases. Although 8 processors are initially used, several of the processors quickly fall idle as the computation progresses. In Figure 5.3, on the other hand, the load is balanced more evenly between the processors. It should be noted that the scales are different in the diagrams, so that 8 processors are used for a large part of the computation, and at least 4 for most of it. The average parallelism has risen from 2.7 to 6.2. This has the desired effect of decreasing the run-time. It has decreased from 45335 to 19753 cycles.

5.6 Tidying the code

The next stage is the introduction of monads, but before this can be done the code should be tidied up so that the monadic transformation rules can be applied. In this case these tidying up steps are very simple:

Firstly the variables are renamed, by alpha conversion, so that their names reflect their function. This is not a necessary step but it aids the user and leads to a better style in the final code:

```
maptri f a xss =
    let m = (size_P xss)`div`2 + 1
        kept_work = map_P (take_S m) xss
        sent_work = map_P (drop_S m) xss
        sum_kept = map_P (foldl_S f a) kept_work
        sum_rec = reverse_P (map_P (foldl_S f a) (reverse_P sent_work))
    in
    map2_P f sum_kept sum_rec
```

In preparation for the MPI APM step (Section 5.8), there should also be only one APM function per line. This can be done later, but now is a convenient time to do it since the program is being tidied up anyway. Lemma 8 can be used to introduce the extra **let** expressions, and Lemma 5 to rearrange the equations into a more sensible order.

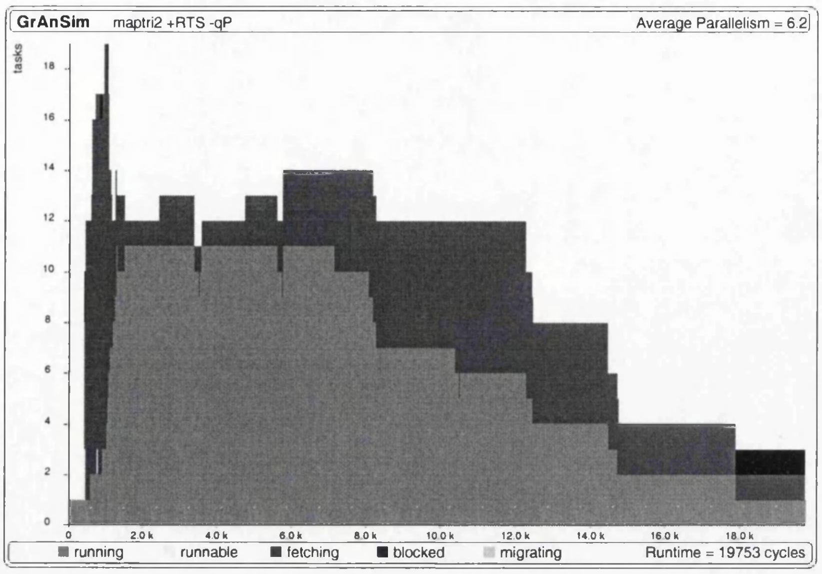


Figure 5.3: Activity profile of maptri after load balancing

Version 5

$$
\begin{array}{l}
maptri\ f\ a\ xss\ = \\
\quad \textbf{let}\ m\ =\ (size_P\ xss)\text{`}div\text{`}2\ +\ 1 \\
\qquad kept\_work\ =\ map_P\ (take_S\ m)\ xss \\
\qquad sent\_work\ =\ map_P\ (drop_S\ m)\ xss \\
\qquad rec\_work\ =\ reverse_P\ sent\_work \\
\qquad sum\_kept\ =\ map_P\ (foldl_S\ f\ a)\ kept\_work \\
\qquad sum\_offload\ =\ map_P\ (foldl_S\ f\ a)\ rec\_work \\
\qquad sum\_rec\ =\ reverse_P\ sum\_offload \\
\quad \textbf{in} \\
\quad map2_P\ f\ sum\_kept\ sum\_rec
\end{array}
$$

There may also be other changes to the program which are necessary at this stage in the derivation. For example, the left-hand sides of the **let** expressions should be single variables. Any tuples should be separated out at this point. Any **let** bindings which define functions should also be moved out of the **let** expression. Such changes are discussed in Section 3.5.1. However such changes are not needed in the current case study.

5.7 Monads

Because the target language is C+MPI, an imperative language, it can be useful to introduce monads into the program to model side-effects. Section 3.6 describes how to do this. The programs from now on become more complicated, and it may be helpful to refer to Section 2.8.4 which describes how such a program can be read.

5.7.1 Introducing monads to the program

First of all, the whole program is encapsulated in a monad, using *return* (Lemma 36):

$$
\begin{array}{ll}
maptri\ :: & (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow\ \alpha\ \rightarrow ParFinSeq(SeqFinSeq\ \alpha)\ \rightarrow \\
 & IOPST\ (ParFinSeq\ \alpha)
\end{array}
$$

```
maptri f a xss = return ( let m = ...
                              ...
                          in
                          ...)
```

Now we recursively apply Lemma 38 to move the **let** expressions out of the **let** and into the **do**. This lemma can be applied because of the nature of the data dependencies in the **let** expression—each equation depends only on previous equations and not on future ones. After this, Law 35 can be used to tidy up the program as shown below:

```
maptri f a xss =
    do m ← return ((size_P xss)`div`2 + 1)
       kept_work ← return (map_P (take_S m) xss)
       sent_work ← return (map_P (drop_S m) xss)
       rec_work ← return (reverse_P sent_work)
       sum_kept ← return (map_P (foldl_S f a) kept_work)
       sum_offload ← return (map_P (foldl_S f a) rec_work)
       sum_rec ← return (reverse_P sum_offload)
       return (map2_P f sum_kept sum_rec)
```

At the moment, *maptri* is a single function in isolation, but if it is to be run in a traditional language such as C, it needs to be contained within an entire program. Therefore a *main* function is introduced using Rule 44 from Section 3.6.4. This obtains the parameter values from the input, calls the *maptri* function with them, and prints the results, as described in Section 3.6.4.

This *main* function uses other functions to get data from the input. In this particular case study, *enter_int* and *enter_tmatrix* are used to obtain a and xss from the input. *enter_int* takes an integer from the input, and *enter_tmatrix* n takes a triangular matrix with n columns from the input. Input functions are discussed in greater detail in Section 3.6.4. The other parameter, f, is built into the program instead of being input into it because it is hard to input functions and because a program such as *maptri* is likely to be used extensively with a single fixed operator.

In this case, f is instantiated to $(+)$, which is an associative operation with unit 0. The parameter a should therefore be 0 in order for the program to be correct. This function is used because it is simple, and yet sufficient for the purposes of this case study. However it doesn't fulfill the assumption in Section 5.4 that communication costs are low compared to calculation costs. This can be altered for testing purposes by forcing the function to do an expensive computation before the addition as was done for the profiling in Section 5.5. See Section 5.11 for more details.

These changes modify the program as follows:

Version 6

```
main :: Int → IOPST()
main p =
    do let enter_int = ...
           enter_tmatrix n = ...
       let maptri f a xss =
             do ...
       a ← enter_int
       xss ← enter_tmatrix p
       result ← maptri (+) a xss
       pst_putStr(show result)
```

5.7.2 Variables

The target language, C+MPI, uses variables to store values. This can be simulated in Haskell as shown in Section 2.8.2. The variables need to be created, values for them need to be stored and their values also need to be retrieved. In addition the distribution of variables across the parallel system needs to be considered as some variables need to be replicated and others, which contain more than one piece of data, need to be distributed across the system. Other factors also need to be considered such as the effect on parameters and duplicated values. In order to keep track of all these things, the transformation is divided up into smaller steps using the suggestions and rules given in Section 3.7.

Instantiate the types

At the moment the program is polymorphic—it works for parameters of multiple types. These types now need to be restricted to particular types as described in Section 3.5.1. In this case study, we examine the case when the input values are integers, and so the types are instantiated to *Int*. This is done using Lemma 2 to give:

```
maptri :: (Int → Int → Int) → Int → ParFinSeq(SeqFinSeq Int) →
          IOPST (ParFinSeq Int)
maptri f a xss_v = ...
```

Global Variables

As indicated in Section 3.7.2, global variables need to be created at the beginning of the program. Commonly global variables are used for the number of processors in the system, p, the processor ids, ***pid***, and the number of data elements, n. There may also be global variables specific to the program. In this case study, the number of processors was assumed to be equal to the number of rows in the matrix, so only the variables ***pid*** and n are needed.

The global variables are created after the system is started (using ***start*** provided by *IOPST*), but before the local function definitions. Again, this follows the pattern in C. Their values are then set after the local definitions but before any of the other code, and these variables must be accessed before their use. In this case study, the value of n is used instead of $\boldsymbol{size_P}$ *xss* within ***maptri***, in the calculation of m, because they have the same value. Therefore it is retrieved from n_v at the start of ***maptri*** and the calculation of m modified accordingly.

Although n is scalar, it is required in all the processors and so must be replicated over the system, using $\boldsymbol{repeat_P}$.

These actions are encapsulated in Rules 45 and 48 in Section 3.7, the application of which results in the following version of the program.

```
main p =
    do start p
        — create global variables
      n_v ← create_var (repeat_P (0 :: Int))
      pid_v ← create_var (repeat_P (0 :: Int))
      let ...
      let maptri f a xss =
          do n ← retrieve n_v
             m ← return (n 'div' 2 + 1)
             ...
        — set the values of global vars
      n ← get_size
      store n_v (repeat_P n)
      pid ← get_pid
      store pid_v pid
      ...
```

As the reader may have noticed, there is a mistake in the above code in the calculation of m. n is a sequence of values, because it was replicated over the system to make it accessible to all processors. In contrast $size_P\ xss$, which was previously used to calculate m, returns a single value. Therefore the calculation of m produces a type error when the program is run.

In general, the expressions in which duplicated values (such as that stored in n) are used should be modified using Lemma 25 from Section 3.7.3, thus avoiding this problem. In *maptri*, n is only used in the call to *enter_tmatrix* and in the calculation of m. These pieces of code therefore become the following:

```
m ← return (map_P (λ i → i'div'2 + 1) n)
```

and

```
xss ← enter_tmatrix (head_P n)
```

Input values

Variables are also needed to store the values entered by the user. This can be achieved by changing the implementation of the input functions, and calling them with an additional variable parameter as described in Section 3.7. This has the additional effect of changing a from a single value to a sequence of values spread across the parallel system. The way a is accessed must also change in a similar way to the code involving n above.

The code calling the input functions becomes:

```
a_v ← create_var (repeat_P (0 :: Int))
enter_int a_v
a ← retrieve a_v
xss_v ← create_var (repeat_P (empty_S :: SeqFinSeq Int))
enter_tmatrix (head_P n) xss_v
xss ← retrieve xss_v
```

The input functions themselves also change in similar ways. The function *enter_tmatrix* is based on *enter_vector*, which is described in Section 3.6.4:

```
enter_tmatrix :: Int → VarFn (SeqFinSeq Int) → IOPST ()
enter_tmatrix n var_v =
    do xss ← enter n 0
       store var_v (list2parfs (map list2seqfs xss))
```

In this function, *enter* :: *Int* → *Int* → *IOPST*[[*Int*]] requests and returns the values in the triangular matrix. These values are then converted to the appropriate parallel and sequential types, using *list2parfs* and *list2seqfs*, and stored in *var_v*.

Parameters

Because of these changes, the parameters of *maptri* also have to change. For example, as mentioned above, the parameter a is no longer a single value, but rather a parallel sequence of values. We also have to consider which variables should be passed by reference and which by value, as discussed in Section 3.7.4.

In C, the target language for this case study, arrays are always passed by reference. This is modelled in the program by passing *xss*, a parallel array variable, by reference. In contrast, a is a parallel integer variable—it only has one value in each processor—and therefore does not have to be passed by reference. In addition, its value is fixed and *maptri* does not alter it. Therefore it is fine to pass it by value.

Therefore *xss_v* rather than *xss* itself, and a, which is now a sequence of values (one for each processor), are passed to *maptri*. The type of *maptri* changes accordingly, as does the way in which its parameters are accessed. *xss* is accessed from *xss_v* straight away, using *retrieve*, and a is referred to as a sequence instead of as a single value.

This is shown below:

```
maptri :: (Int → Int → Int) → (ParFinSeq Int) →
          VarFn(SeqFinSeq Int) → IOPST (ParFinSeq Int)
maptri f a xss_v =
     do xss ← retrieve xss_v
          ...
          sum_kept ← return (map2_P (foldl_S f) a kept_work)
          sum_offload ← return (map2_P (foldl_S f) a rec_work) ...
...
result ← (maptri (+) a xss_v)
```

Other variables

A few other variables need to be added. The *result* at the end of *main* can be stored in a variable, as can the intermediate values in *maptri*. This is done using the rules in Section 3.7.1.

5.7.3 The resultant program

After all these changes, the program is now as follows:

Version 7

```
...
maptri :: (Int → Int → Int) → (ParFinSeq Int) →
          VarFn(SeqFinSeq Int) → IOPST (ParFinSeq Int)
maptri f a xss_v =
        do n ← retrieve n_v
           xss ← retrieve xss_v
           m_v ← create_var (repeat_P (0 :: Int))
           m ← return (map_P (λ i → i`div`2 + 1) n)
           m ← retrieve m_v
           kept_work_v ← create_var (repeat_P(empty_S :: SeqFinSeq Int))
           store kept_work_v (map2_P take_S m xss)
           kept_work ← retrieve kept_work_v
           sent_work_v ← create_var (repeat_P(empty_S :: SeqFinSeq Int))
           store sent_work_v (map2_P drop_S m xss)
           sent_work ← retrieve sent_work_v
           rec_work_v ← create_var (repeat_P(empty_S :: SeqFinSeq Int))
           store rec_work_v (reverse_P sent_work)
           rec_work ← retrieve rec_work_v
           sum_kept_v ← create_var (repeat_P (0 :: Int))
           store sum_kept_v (map2_P (foldl_S f) a kept_work)
           sum_kept ← retrieve sum_kept_v
           sum_offload_v ← create_var (repeat_P (0 :: Int))
           store sum_offload_v (map2_P (foldl_S f) a rec_work)
           sum_offload ← retrieve sum_offload_v
           sum_rec_v ← create_var (repeat_P (0 :: Int))
           store sum_rec_v (reverse_P sum_offload)
           sum_rec ← retrieve sum_rec_v
           return (map2_P f sum_kept sum_rec)
  ...
  result_v ← create_var (repeat_P (0 :: Int))
  result ← (maptri (+) a xss_v)
  store result_v result
  ...
```

5.8 MPI APM

The next stage is to transform the program to use a different APM, the MPI APM. This provides Haskell simulations of MPI functions, as described in Section 2.10. When using this APM, all transfers of data from one processor to another should be expressed using functions from the APM.

Data transfers can usually be easily identified because they are encapsulated in functions from particular APMs, such as the *ParFinSeq* APM. Sometimes they occur implicitly, but not in this case study. See Chapter 7 for an example in which such communication occurs.

In the *maptri* case study, the data transfers occur within the *maptri* function and the input functions which broadcast or scatter their data. The changes to the input functions are described in Section 3.8.2, and we do not need to go into them here. The main function, *maptri*, changes in a similar way, using the rules described in Section 3.8.1 to transform the calls to standard APM functions into calls to MPI APM functions. In fact all the communication here is encapsulated in the *reverse* function which can be transformed to an MPI APM version, *requestrev*, described in Sections 2.10.5 and 3.8.1.

After the application of these rules, the program can be tidied up somewhat by removing redundant equations (e.g., instances of the *retrieve* functions), and by moving the *create_var*s to the top of the function. This can be done using monadic manipulation rules from Section 3.5.1.

This gives:

Version 8

```
maptri f a xss_v =
     do m_v ← create_var ($repeat_P$(0 :: Int))
        kept_work_v ← create_var ($repeat_P$ ($empty_S$ :: SeqFinSeq Int))
        sent_work_v ← create_var ($repeat_P$ ($empty_S$ :: SeqFinSeq Int))
        rec_work_v ← create_var ($repeat_P$ ($empty_S$ :: SeqFinSeq Int))
        sum_kept_v ← create_var ($repeat_P$ (0 :: Int))
        sum_offload_v ← create_var ($repeat_P$ (0 :: Int))
        sum_rec_v ← create_var ($repeat_P$ (0 :: Int))
        n ← retrieve n_v
        xss ← retrieve xss_v
        store m_v ($map_P$ (λ i → (i`div`2 + 1)) n)
        m ← retrieve m_v
        store kept_work_v ($map2_P$ $take_S$ m xss)
        kept_work ← retrieve kept_work_v
        store sent_work_v ($map2_P$ $drop_S$ m xss)
        requestrev sent_work_v rec_work_v (ARRAY INT)
        rec_work ← retrieve rec_work_v
        store sum_kept_v ($map2_P$ ($foldl_S$ f) a kept_work)
        sum_kept ← retrieve sum_kept_v
        store sum_offload_v ($map2_P$ ($foldl_S$ f) a rec_work)
        requestrev sum_offload_v sum_rec_v INT
        sum_rec ← retrieve sum_rec_v
        return ($map2_P$ f sum_kept sum_rec)
```

5.9 Individual level

C+MPI views the parallel world from an individual viewpoint: the program specifies what a single processor does. However, up to this point the Haskell programs have viewed the world from a global or collective viewpoint, specifying what the whole system does. These individual and collective levels are explained in more detail in Section 2.11.

To bring the program closer to the target language, an individual level viewpoint should therefore be used instead of a collective one. Such a program cannot be immediately run in Haskell, but can be with an appropriate wrapper function, as explained in Section 2.11.

The changes which need to be made to the code are outlined in Section 3.9. The calls to the MPI APM functions remain essentially the same. Variable manipulation functions such as *create_var* deal with single values rather than sequences of values. The input functions only take in values if the processor's id is 0, and then distribute them from that processor, as only one processor reads in the input in a real parallel system. Finally parallel *map*s change to direct function applications.

Making these changes, then tidying the code, we get the following code.

```
main :: IOPST()
main =
     do start
          — create global variables
        n_v ← create_var (0 :: Int)
        pid_v ← create_var (0 :: Int)
          — local input functions
        let enter_int var_v = ...
        let enter_tmatrix n var_v =
           do ...
              if (pid == 0) then
              do xss ← enter n 0
                 store tmp_v (list2seqfs (concat xss))
              else return ()
                — set up variables for scattering
              ...
                — scatter
              mpi_scatterv tmp_v sizes displs INT xs_v INT 0
          — other local function definitions
        let requestrev :: (Dyn α) ⇒ VarFn α → VarFn α → ItemType → IOPST ()
            requestrev sendbuf recvbuf datatype =
             do
               ...
                 — do reverse
               mpi_send sendbuf datatype (n − 1 − pid)
               mpi_recv recvbuf datatype (n − 1 − pid)

       — main program
     let maptri :: (Int → Int → Int) → Int →
                     VarFn(SeqFinSeq Int) → IOPST Int
         maptri f a xs_v =
        do
          m_v ← create_var (0 :: Int)
          kept_work_v ← create_var (empty_S :: SeqFinSeq Int)
          sent_work_v ← create_var (empty_S :: SeqFinSeq Int)
          rec_work_v ← create_var (empty_S :: SeqFinSeq Int)
          sum_kept_v ← create_var (0 :: Int)
          sum_offload_v ← create_var (0 :: Int)
          sum_rec_v ← create_var (0 :: Int)
          xs ← retrieve xs_v
          n ← retrieve n_v
          store m_v (n 'div' 2 + 1)
          m ← retrieve m_v
          store kept_work_v (take_S m xs)
          kept_work ← retrieve kept_work_v
          store sent_work_v (drop_S m xs)
          requestrev sent_work_v rec_work_v (ARRAY INT)
          rec_work ← retrieve rec_work_v
          store sum_kept_v (foldl_S f a kept_work)
          sum_kept ← retrieve sum_kept_v
          store sum_offload_v (foldl_S f a rec_work)
          requestrev sum_offload_v sum_rec_v INT
          sum_rec ← retrieve sum_rec_v
          return (f sum_kept sum_rec)
```

```
a_v ← create_var (0 :: Int)
xs_v ← create_var (empty_S :: SeqFinSeq Int)
  — set the values of the global variables
n ← get_size
store n_v n
n ← retrieve n_v
pid ← get_pid
store pid_v pid
pid ← retrieve pid_v
  — Input and broadcast a
enter_int a_v
  — Input and scatter xss, storing it in xs_v
enter_tmatrix (head_p n) xs_v
  — call maptri
result ← maptri(+) a xs_v
putStr(show result)
```

5.10 C+MPI

5.10.1 Transformation to C+MPI

The program can now be transformed into C+MPI using the rules given in Section 3.11. APM functions transform in fixed ways into MPI operations or combinations of C and MPI operations. An example of this is the *requestrev* function. The *main* function in Haskell is transformed into the `main` procedure in C+MPI. This initialises MPI, sets the values of global variables, `n`, `m` and `pid`, and calls the input functions and *maptri*. It also closes MPI at the end. The input functions are transformed into predefined C+MPI functions.

This transformation is accomplished using Rule 61 from Section 3.11.2 to change the structure of the program, together with a set of other rules, given in Table 3.2 to change individual function calls and variable manipulation functions. It also adds in necessary parts of the code such as function declarations, and a function, `output`, which prints out the results in the correct order.

Note that reverse is transformed into predefined functions `requestrevarr` and `requestrevint`, which implement *requestrev* for arrays of integers and single integers respectively, depending on the type of the values used, and that *main* is written according to a set pattern.

```
int n,pid,m;

main(int argc, char *argv[])
{
  int a = 0;
  int xss[SIZE*SIZE] = [];
  int result[SIZE] = [];
  int xs[SIZE] = [];
  int sizes[SIZE] = [];
  int displs[SIZE] = [];

    /* mpi_init */
  errcode = MPI_Init (&argc, &argv);

    /* set the values of global variables */
  MPI_Comm_size (MPI_COMM_WORLD, &n);
  MPI_Comm_rank (MPI_COMM_WORLD, &pid);
```

```
    /* enter and broadcast/scatter input values */
  enterint(&a);
  entertmatrix(n, xs);

    /* call maptri */
  result = maptri(&add, a, xs);

  output(result);

    /*finish MPI*/
  errcode = MPI_Finalize ();
}

int maptri (int (*f)(int,int), int a, int xs[])
{
    /* The convention about variables is that x corresponds to
    /* x and x_v */
  int kept_work[SIZE], sent_work[SIZE], rec_work[SIZE] = [];
  int sum_kept, sum_offload, sum_rec = 0;
  int kept_work_size, sent_work_size, rec_work_size = 0;

  m = n/2 + 1;

  take(xs,size xs,m,kept_work,&kept_work_size);
  drop(xs,size xs,m,sent_work,&sent_work_size);

  requestrevarr(sent_work, sent_work_size, rec_work, &rec_work_size);
  /* We pick the array version of reverse because of the
  /* (ARRAY INT) parameter */

  sum_kept = foldl(f, a, kept_work, kept_work_size);
  sum_offload = foldl(f, a, rec_work, rec_work_size);

  sum_rec = requestrevint(sum_offload);

  return ((*f)(sum_kept, sum_rec));
}

void requestrevarr(int xs[], int size, int *res, int *res_size)
   ~~~

int requestrevint(int x)
   ~~~

void enterint(int *var)
   ~~~

void enterdmatrix(int n, int xs[])
   ~~~

void output(int result)
   ~~~
```

Unnecessary initialisations can now be removed, using Rule 62 in Section 3.11.4, the size variables renamed, and *size xs* replaced by *pid* + 1, where *pid* is the current processor's id, since these are equal. This produces the following code:

```
int n,pid;

main(int argc, char *argv[])
{
  int errcode;
  int result,a,i;
  int sizes[SIZE];
  int displs[SIZE];
  int xss[SIZE*SIZE];
  int xs[SIZE];

  ...
}

int maptri (int (*f)(int,int), int a, int xs[])
{
  int kept_work[SIZE], sent_work[SIZE], rec_work[SIZE];
  int sum_kept, sum_offload, sum_rec;
  int m, kept_size, sent_size, rec_size;

  ...

  take(xs,pid+1,m,kept_work,&kept_size);
  drop(xs,pid+1,m,sent_work,&sent_size);

  requestrevarr(sent_work, sent_size, rec_work, &rec_size);

  sum_kept = foldl(f, a, kept_work, kept_size);
  sum_offload = foldl(f, a, rec_work, rec_size);

  ...
}
```

5.10.2 Optimisations

After the basic transformation, it is still possible to apply extra rules to tidy up the code and perform some optimisations, but it isn't as easy and correctness can no longer be proven. As much reasoning as possible should be done in the Haskell versions higher up in the derivation.

In this case study, the above version is executable but isn't very efficient because it creates unnecessary arrays during the takes and drops. Instead of calculating and storing kept_work and sent_work separately, C can manipulate them as slices of the main array, xs. This optimisation could be included in the Haskell stages, possibly by replacing *take*s and *drop*s at those stages with the function *slice* which transforms more naturally into C slices. However the rest of this section shows how the optimisation can be carried out at this point for demonstration purposes.

The replacement of takes and drops by slices of arrays in C can be expressed more formally in Rules as follows. Note however that these cannot be proved correct due to a lack of formal semantics in C.

Rule 63 (take rule)
For any arrays xs, res, positive integers m, res_size and C statements Fs such that

- size is the number of elements in xs and
- Fs never accesses res[i], i $\geq$ res_size.

```
take(xs,size,m,res,& res_size);     =>    res_size = min(m,size);
Fs;                                       Fs[xs/res];
```

Rule 64 (drop rule)
For any arrays xs, res, positive integers size, m, res_size and C statements Fs such that

- size is the number of elements in xs and
- Fs never accesses res[i], i $\geq$ res_size.

```
drop(xs,size,m,res,& res_size);     =>    res_size = size - min(m,size);
Fs;                                       Fs[xs+min(m,size)/res];
```

In order for these rules to be applied, the sizes of each of the arrays need to be known. They can be calculated using the size properties of *take* and *drop* (Lemmas 21 and 22).

Size of kept_work = kept_size = min(m,xs_size) = min(m,pid+1).

Size of sent_work = sent_size = xs_size - min(m,xs_size) = (pid+1)-kept_size.

where pid is the processor number.

The application of these rules then leads to the final version of the program, as follows:

Version 9

```
int m,n,pid;

main(int argc, char *argv[])
{
  ...
}

int maptri (int (*f)(int,int), int a, int xs[])
{
  int rec_work[SIZE], kept_size, sent_size, rec_size;
  int m, sum_kept, sum_offload, sum_rec;

  m = n/2 + 1;

  kept_size = min(m,pid+1);
  sent_size = (pid+1)-kept_size;

  requestrevarr(xs+kept_size, sent_size, rec_work, &rec_size);

  sum_kept = foldl(f, a, xs, kept_size);
  sum_offload = foldl(f, a, rec_work, rec_size);

  sum_rec = requestrevint(sum_offload);

  return ((*f)(sum_kept, sum_rec));
 }
```

5.10.3 Discussion of the C version

The MPI APM functions are implemented by predefined MPI operations or combinations of C and MPI operations. See Section 3.11.2 for a general look at this. This section examines a specific example, *requestrev*, which occurred in the case study. This function expresses a pattern of communication in which elements from processor i are swapped with those from processor $n-i-1$, when the processors are numbered 0 to $n-1$. This is the communication pattern encapsulated in the function, $reverse_P$ from the *ParFinSeq* APM.

This function has two forms, depending on the type with which it is used. The user of the methodology can work out which version to transform to by looking at the type parameter in the *requestrev* function.

The code for one form of the *requestrev* function is given below. There is no MPI function which carries out this pattern of communication, and so it must be implemented using MPI functions which are available, in this case, individual sends and receives.

```
void requestrevarr(int xs[], int size, int res[], int *res_size)
/* Implements reverse for arrays of integers.
   The 2nd parameter gives the number of elements of the array to send;
   res holds the received array */
{
  MPI_Request rev_handle;
  MPI_Status status,status2;

  /* send and receive data in a reverse pattern*/
  /* data should be sent to and received from processor (n-1-pid)*/
  MPI_Issend(xs,size,MPI_INT,n-1-pid,0,MPI_COMM_WORLD,&rev_handle);
  MPI_Recv(res,SIZE,MPI_INT,n-1-pid,0,MPI_COMM_WORLD,&status);
  MPI_Wait(&rev_handle,&status2);
  MPI_Get_count(&status, MPI_INT, res_size);
}
```

C+MPI versions of such functions can be created in advance and stored in a library in a similar way to the APMs themselves. They can then be used without having to be re-implemented each time.

5.11 Timings

This program was compared to C+MPI code written by hand without using load balancing, on a 32-processor Beowulf cluster. The timings use standard input matrices for xss, and $a = 0$, with a computationally intensive function for f. The timings are given in Table 5.1. Figure 5.4 compares them with each other graphically. Notice how the load balanced version performs consistently better than the one without load balancing.

However the peculiarities of the communication system and varying processors can skew the results. In this case, the times jump between 28 and 32 processors. This is probably caused by slowness in one of the intermediate processors or connections. If the processors are allocated to matrix rows in a different way, the timings also come out differently.

The speedup graph (Figure 5.5) also shows the superior performance of the load-balanced version, but it shows some odd behaviour. The load-balanced version exhibits super-linear speedup. This can probably be explained by a sub-optimal sequential version. Note the dip at 32 processors, as before.

5.12 Discussion

After looking at the case study in detail, examining each transformation and stage of the derivation in detail, it is time to step back and look at it as a whole, observing its structure and comparing it

n	Sequential	Naive Parallel	Load balanced Parallel
4	33	10	10
8	117	19	14
12	253	28	19
16	442	38	24
20	683	47	29
24	975	56	33
28	1319	66	38
32	1716	141	85

Table 5.1: Maptri program timings in seconds

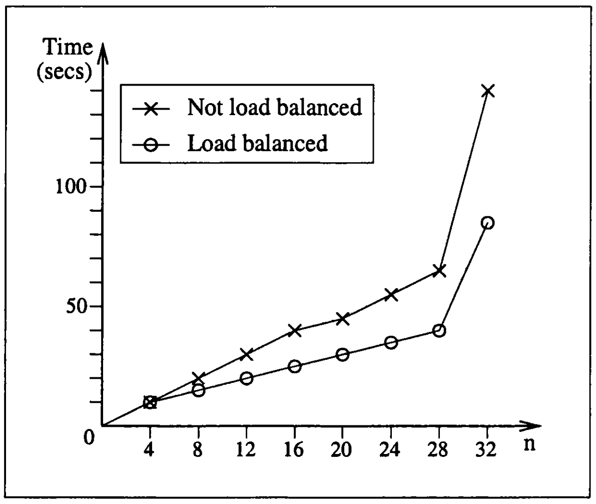


Figure 5.4: Comparison of maptri timings

to the general information which was given in Chapter 2.

First of all, let us look at the layout of the derivation. This is given in Figure 5.6. The reader may find it interesting to compare it with the diagrams of more general derivation layouts in Figure 2.2 of Chapter 2 and Figure 3.1 of Chapter 3.

There are several interesting points to note. First of all, the case study is sufficiently simple that only a single maths specification is needed. The abstract specification is detailed enough to also act as the method specification. The latter is only really needed in more complicated situations in which the abstract maths specification does not give sufficient details to allow one to calculate a result.

After the specification stage, the case study proceeds through the sequence of steps mentioned in the general case (Chapter 2). First of all, the maths specification is transformed into Haskell and then parallelism is identified. The program then proceeds through a sequence of intermediate stages before being transformed to an individual viewpoint style program, and then to the target language, C+MPI. Note that the intermediate stages form the body of the derivation, and that, despite the relative simplicity of the case study, there are a significant number of them.

In the description of the general derivation (Section 2.2), it is not specified which intermediate transformations are actually carried out, how many times or in what order. This is left up to the programmer and the particular case study. In the case study just carried out, however, we can see a particular sequence of these stages. The reader may also have noted that each of the stages is only carried out once. This is due to the simplicity of the case study, which does not require the full power

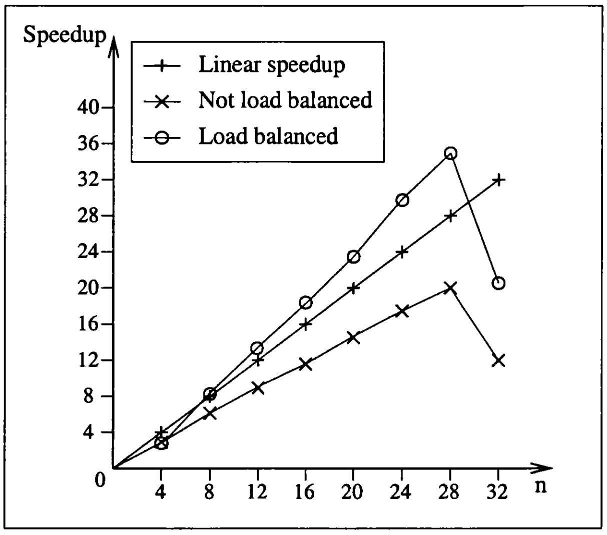


Figure 5.5: Speedups of maptri programs

of the methodology. See Chapter 7 for a more complicated case study.

Since C+MPI is targeted, monads are used to model the imperative features of IO and mutable state. C+MPI is also SPMD (see Section 2.10.1), and so uses an individual level semantics. Since the Haskell model is collective, a transformation to an individual level viewpoint is required. Note that all the transformations are carried out formally using equational reasoning in Haskell. Only at the last step is the program written in another language - the target language, C+MPI.

However, this is only one possible derivation of many. Figure 5.7 shows the branching structure of the possible derivations, with the one carried out in this chapter shown in detail. Only a few of the possible choices at each stage and a few of the target programs are shown for the sake of simplicity. However it can be seen that programs in different languages and using different efficiency techniques can be targeted. Some of these lie more closely together than others in the tree structure and share many common derivation steps while others share only the first few steps.

5.13 Summary

This chapter has presented a simple case study which was carried out using the methodology described in the previous chapters. This has illustrated and clarified many of the points in those chapters, and demonstrated the methodology in action. However this study was fairly simple, and did not cover many areas. The following chapters present a further, more complicated case study. First of all, a specific part of this further study is examined in some detail.

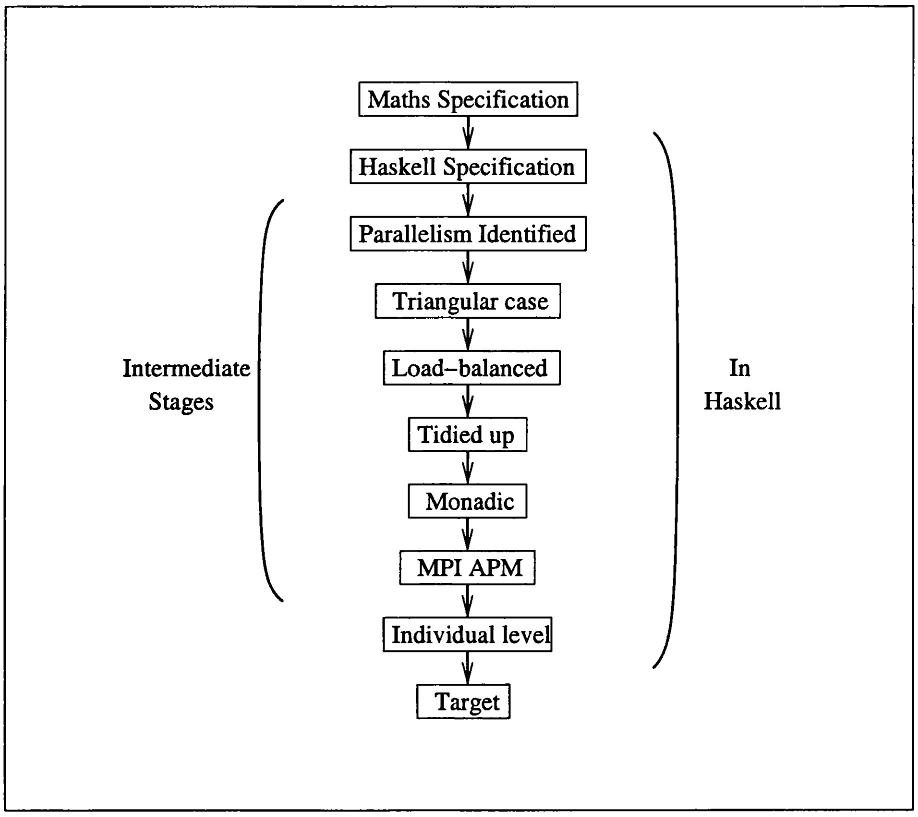


Figure 5.6: The layout of the maptri derivation

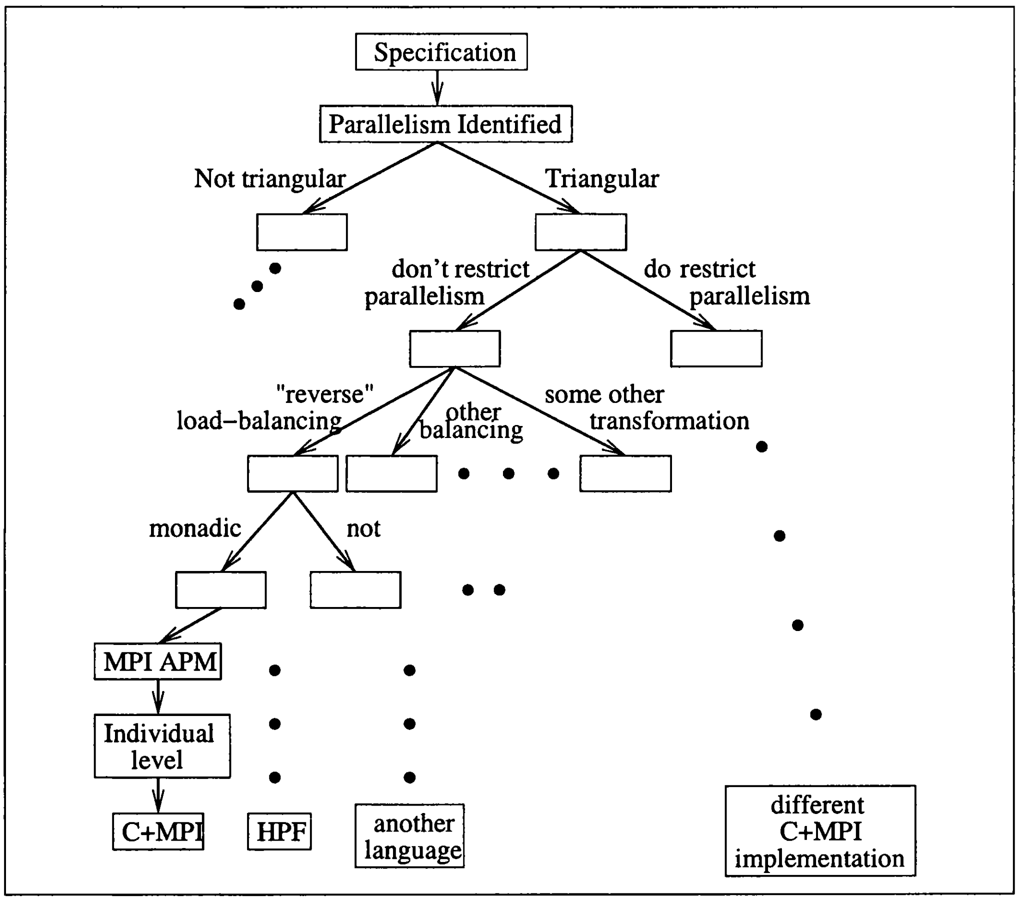


Figure 5.7: The tree of alternative possible derivations in this case study

Chapter 6

Pipelining in Accumulating Scan

6.1 Introduction

This chapter concerns the development of an APM function from start to finish. It follows this function through a series of APMs and observes how it develops, focusing especially on detailed communication-specific optimisations.

The function used is an accumulating scan which is a version of the standard higher-order function *scan*. It can use all of the previously calculated results in calculating the next one, and can be written as an APM function in its own right.

The development of accumulating scan gives us insight into the relationship between APMs, and into the development of an APM function. Accumulating scan is particularly interesting because it can be implemented using a pipelining technique, thus providing a chance to see how a common parallel technique can be used within the context of the methodology. The pipelining technique can be optimised in several communication-specific ways, some of which are dealt with in this chapter. The chapter therefore demonstrates how detailed, low-level optimisations which manipulate communication operations can be introduced within the APM framework.

Due to the communication-specific optimisations involved, many of the programs in this chapter use the monadic and MPI APMs described in Sections 2.8 and 2.10. Section 2.8.4 provides a brief description of the form that such programs take, and information on how to interpret them. This work was developed as part of the Gaussian elimination case study (Chapter 7), and the accumulating scan functions are used there to implement the back substitution phase of the algorithm.

6.1.1 The layout of this chapter

Figure 6.1 describes the sequence of transformations which the accumulating scan function goes through. The rectangles represent the APMs, and the arrows show how they are related. Functions from one APM can be transformed into those from the next. The labels in each rectangle describe that APM. In particular they describe the way in which the accumulating scan function is implemented in that APM. The dots and other arrows represent other possible transformations and APMs, and the dashed line shows the path taken by this chapter through this family of APMs.

First of all, this chapter describes the general *scan* function and shows how it can be transformed through the family of APMs. Successive sections focus on the accumulating scan function. They follow it through the sequence of APMs, describing and explaining each APM and the function's transformation from its previous version.

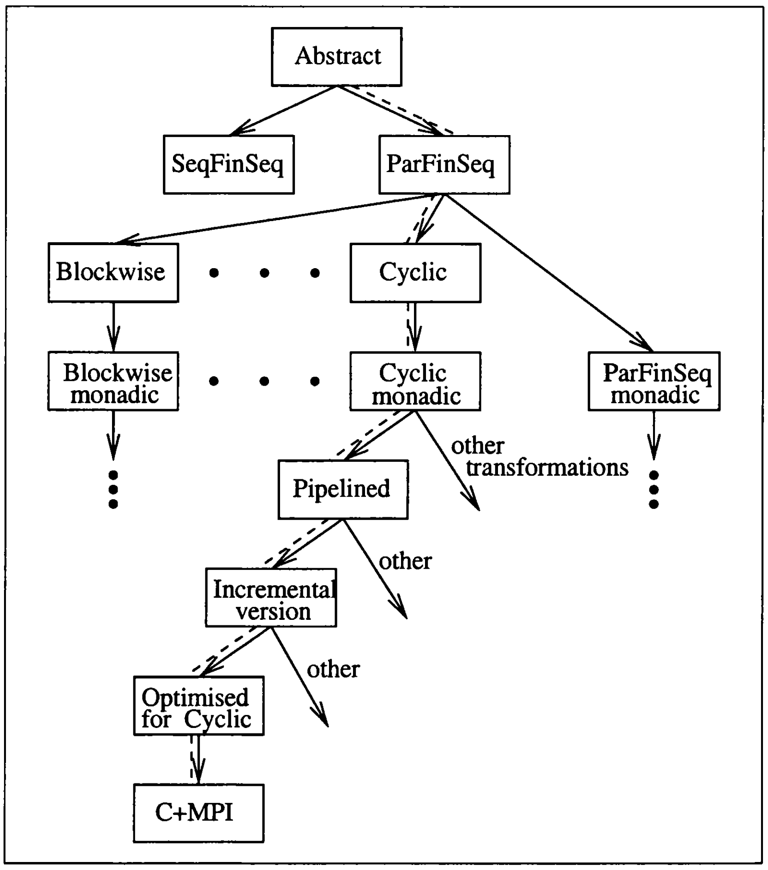


Figure 6.1: The accumulating scan APM hierarchy

6.2 The functions scan and accum_scan

6.2.1 The general scan function

The *scan* functions are standard higher-order functions (h.o.f.'s). They apply a function argument to a list or sequence, and produce a list of its partial sums. For example, *scanr*, a version of *scan* which sums the elements starting at the right-hand end of the list, can be specified as follows:

$$\begin{aligned}&scanr :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to [\beta]\\ &scanr\ f\ a\ [x_1,\ \ldots,\ x_n] = [f\ x_1\ (f\ x_2\ \ldots),\ \ldots,\ f\ x_{n-1}\ (f\ x_n\ a),\ f\ x_n\ a,\ a]\end{aligned}$$

Or the following, if the results of the *scan* are named $res_1, \ldots, res_{n+1}$:

$$\begin{aligned}&scanr\ f\ a\ [x_1,\ \ldots,\ x_n]\\ &\quad = [f\ x_1\ res_2,\ \ldots,\ f\ x_{n-1}\ res_n,\ f\ x_n\ res_{n+1},\ a]\\ &\quad = [res_1,\ \ldots,\ res_{n+1}]\end{aligned}$$

Other variants of *scan* sum from the left, or don't use an initial value, a. For parallel programming it's often better to use a variant which discards the leftmost or rightmost value so that the returned list is the same length as the initial one. These *scan* functions can be easily modified to operate over finite sequences of various types, as are used in the methodology, instead of over lists.

scan functions commonly occur in APMs as they express a pattern of communication when the elements of the sequence are in different processors. Sometimes, for example, when f is associative, they can be implemented in special ways, such as with tree structures (see [O'D94] for a description of how this is done). But sometimes, even when the sequence elements are in different processors, they have to be implemented sequentially, as shown in Figure 6.2. Each processor calculates the partial sum so far and then passes it on to the next processor.

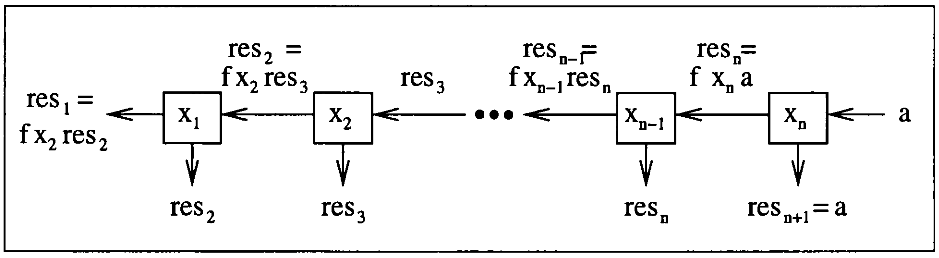


Figure 6.2: Sequential implementation of scanr

This is a form of *pipelining*. Every processor receives a value, does some standard thing to it, and then passes on the result to the next processor. In this example, processor i receives value res_{i+1} from processor $i + 1$, then it applies f to it and the value of x in the processor.

As this stands, it is performed for only one sequence of values, and this is not very efficient. However it can be made more efficient by using the pipeline for several sequences at once. The earlier processors calculate values for new sequences while the later processors are still calculating values for older sequences. It is also possible, in certain situations, to optimise the pipeline so that it works more efficiently even for only one sequence. This situation is what this chapter is interested in. In particular, it focuses on the special case of accumulating scan and how it works when the data is distributed using a standard distribution such as a blockwise or cyclic distribution.

The function *scanr* can be transformed from its specification to a pipelined version in C+MPI with a cyclic distribution using standard transformation techniques as described in Chapter 3.

A non-cyclic version of *scanr*, which discards the rightmost value, is given below in its MPI APM form. It can be used as the basis for an optimised pipelined implementation of special cases such as accumulating scan, as is shown later in this chapter. The cyclic version could also be used as this basis, but that is left until later in this chapter so that the data distributions can be discussed separately (see Section 6.3).

In the following code, *n_v* is a global variable which contains the length of the sequence. *init* is the starting value for the summation, and f is the function used for the summation. *local_val_v* holds the original values of the sequence, and the result is distributed across the processors and stored in the variable *new_val_v*.

```
pipelineP :: (Float → IOPST Float → IOPST Float) → IOPST Float
        → VarFn Float → VarFn Float → IOPST()
pipelineP f init local_val_v new_val_v =
do $n'$ ← $retrieve\ n\_v$
   $n$ ← $return\ (head_P\ n')$
     — set up processor (n-1)
   $local\_val$ ← $retrieve_{indiv}\ (n-1)\ local\_val\_v$
   $new\_val$ ← $f\ local\_val\ init$
   $store_{indiv}\ (n-1)\ new\_val\_v\ new\_val$
     — pipeline
   $for\ [n-2, n-3..0]$
     — deal with each processor in turn starting from the right
   $(\lambda\ i \rightarrow$ do $mpi\_spt2pt\ new\_val\_v\ FLOAT\ (i+1)\ new\_val\_v\ FLOAT\ i$
                  — do calculation for relevant processor
              $new\_val$ ← $retrieve_{indiv}\ i\ new\_val\_v$
              $local\_val$ ← $retrieve_{indiv}\ i\ local\_val\_v$
              $new\_val$ ← $f\ local\_val\ (return\ new\_val)$
              $store_{indiv}\ i\ new\_val\_v\ new\_val)$
```

6.2.2 The accum_scanr function

Accumulating scans are a sub-family of *scan* functions which make use of all the previously calculated values in calculating the next one. This can be best illustrated by looking at one particular example of an accumulating scan, *accum_scanr*1, which will be used extensively throughout this chapter.

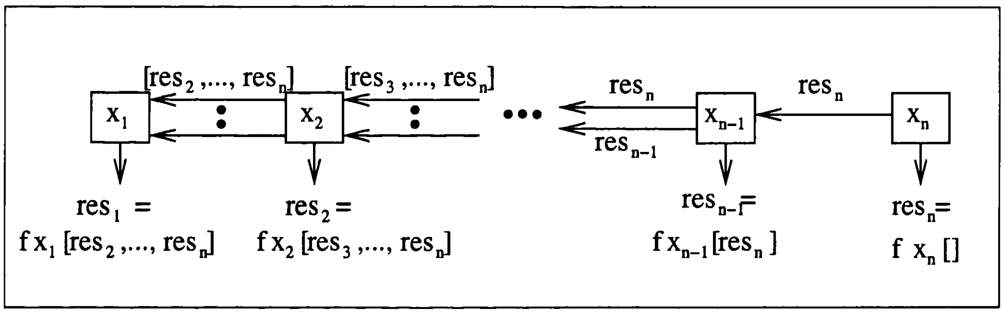


Figure 6.3: Accumulating scanrl

As with the standard higher-order function, *scanr*1, this sums its elements from the right-hand side of its list and doesn't use a specific starting value, a. It is illustrated in Figure 6.3, and can be specified as follows.

$$
\begin{aligned}
&accum\_scanr1 :: (\alpha \rightarrow [\beta] \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\
&accum\_scanr1\ f\ [x_1, \ldots, x_n] \\
&\qquad = [f\ x_1\ [res_2, .., res_n], \ldots, f\ x_{n-1}\ [res_n], f\ x_n\ []] \\
&\qquad = [res_1, \ldots, res_n]
\end{aligned}
$$

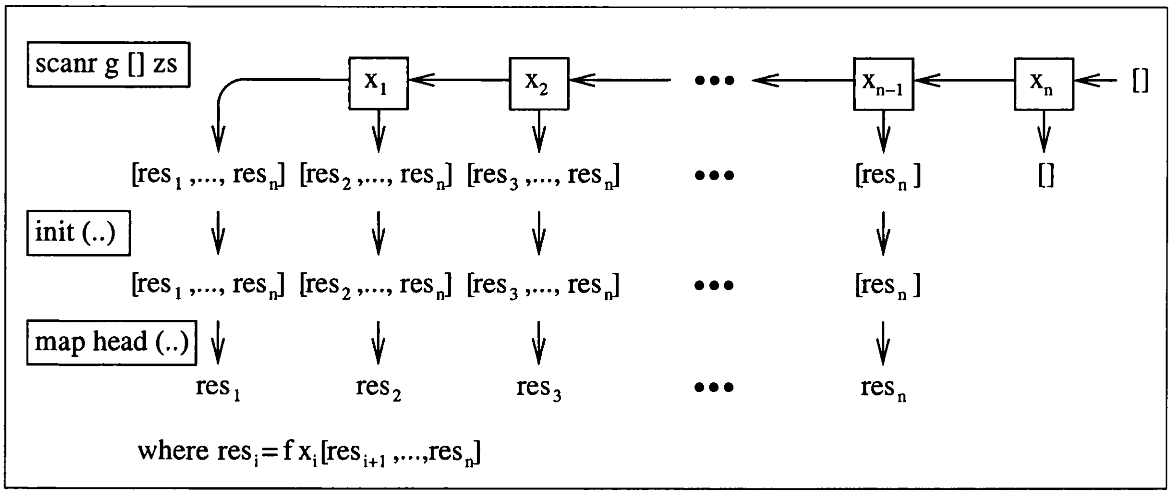
At each point, this function knows $[res_{i+1}, \ldots, res_n]$, the list of results produced so far, and uses this to generate the next result, res_i, using f.

This can be implemented using recursion, in the normal way for higher-order functions on lists:

```
$accum\_scanr1\ f\ [] = []$
$accum\_scanr1\ f\ (z : zs) = (f\ z\ ys) : ys$
  where
    $ys = accum\_scanr1\ f\ zs$
```

This can also be written using a *scanr*, thus showing its connection to the *scan* family more clearly. A new function, g, is written which does the same as f, except that it returns the whole set of results produced so far and not just the last one. This function can then be used in an ordinary *scanr* to produce a sequence of all the results produced at each point in the calculation. As not all of these values are needed, this list is then pruned using *head* and *init*, as shown in Figure 6.4.

```
accum_scanr1 f zs = map head (init (scanr g [] zs))
  where
    g x ys = (f x ys) : ys
```



scanr g [] zs
x1
x2
xn-1
xn
[]
[res1,..., resn] [res2,..., resn] [res3,..., resn]
[resn]
[]
init (..)
[res1,..., resn] [res2,..., resn] [res3,..., resn]
[resn]
map head (..)
res1
res2
res3
resn
where resi = f xi [resi+1,...,resn]

Figure 6.4: An implementation of accum_scanr1 in terms of scanr

However this is not efficient because each processor must wait for the one before it to pass all of the results before it starts its calculation. The following sections show how this function can be optimised as it's transformed into C+MPI code.

6.3 Different data distributions

A data distribution is a method of distributing the data across the processors of the parallel system. Different data distributions are described and discussed in Section 2.7. They can be introduced into a program as described in Section 3.4.4 using special APMs that model specific data distributions, as in Section 2.7.

But before the data distribution can be incorporated, a key decision has to be made. Which data distribution should be used? As shown previously in Figure 6.1, this decision contributes to the branching structure of the derivation. Different data distributions lead to different resultant programs and different optimisations which can be applied.

The data distributions need not be chosen this early in the derivation. Actually there are some advantages to doing it later. For example, more of the derivation can be shared when deriving the same program for different data distributions. However the decision is considered at this point in this chapter because it's simpler at this stage - both to derive and to explain to the reader. At this stage the code isn't yet complicated with monads and MPI functions.

This section restricts the choice to blockwise and cyclic distributions because they are common and sufficient to illustrate the principles. These distributions are described in Section 2.7. The naive parallel and sequential distributions are also considered for comparison purposes. The choice is made by comparing the costs of the function with each distribution.

Operation	Cost
point-to-point communication	$\tau + m \cdot t_c$
point-to-point communication between neighbouring processors	$\tau + m \cdot t_{cn}$
single-node broadcast	$(1 + \log p)(mt_c + \tau)$
single-node scatter	$(p-1)mt_c + (p + \log p)\tau$
total exchange	$(p-1)\log pmt_c + p \log p\tau$

Table 6.1: Cost formulae for some communication primitives. m is the maximum number of elements in the messages transmitted

6.3.1 The cost model

A cost model is needed if cost calculations are to be carried out. The cost model used is a slightly modified version of that used by Rauber and Rünger when deriving structured parallel programs [RR95]. The machine is described by the set of parameters,

$$\begin{aligned} p &= \text{the number of processors} \\ t_c &= \text{element transfer time for point-to-point messages} \\ t_{cn} &= \text{element transfer time for point-to-point messages} \\ &\quad\ \text{between neighbouring processors} \\ \tau &= \text{startup time for point-to-point messages} \end{aligned}$$

In order to simplify the calculations and since we only use the costs for comparison purposes, an arithmetic operation is assumed to take unit time. Transfer times, t_c and t_{cn}, are expressed as multiples of this. The latter, t_{cn}, is introduced because pipelining makes heavy use of communication between neighbouring processors. Each communication involves an element, typically a floating-point number, that takes up a fixed number of bytes. Since all of these are the same size, t_c and t_{cn} are the times for the communication of one of these elements rather than of a single byte.

Rauber and Rünger also give the costs of various communication operations in terms of these parameters. Table 6.1 summarises the relevant ones for the cost calculations in this chapter. Total exchange is sometimes called all-to-all scatter.

As described by Rauber and Rünger [RR95], this is a major simplification of real machines and ignores many practical issues such as overlapping communication and computation, network contention and cache effects. However they have shown that the resulting predictions are close enough to timings on real machines to give sufficiently accurate results for their purposes. For us, they are accurate enough to highlight large differences between efficiencies, and hence accurate enough for making choices about such things as data distributions. There is work on improving the cost models used with APMs [ORR01].

6.3.2 Cost calculations

In order to calculate the cost of the function, we need to look at how it can be implemented with each data distribution. Some implementation details have to be known. However these details are not introduced until later in the derivation, although they can still be referred to now. This section therefore describes some possible implementations, including optimisations which can be made, focusing on pipelining implementations, as described in Section 6.4.

This section also assumes that the pipelined function can be done incrementally, as explained in

Section 6.6, so that it can start before all of the needed results are available. It is assumed that part of the calculation can be done for each value which *is* available, and that these parts are of approximately the same size.

These cost calculations are actually done for *accum_scanl*, which starts from the left-hand end of the list, rather than for *accum_scanr* which is the version used in the rest of this chapter and in Chapter 7. This simplifies the presentation and calculation, and the costs apply equally to *accum_scanr*.

The cost analyses use diagrams (Figures 6.5 to 6.10) to help the reader to understand what is going on in the algorithms. These diagrams represent the time used within a processor as a rectangle. Each processor calculates several values, and therefore these large rectangles are divided into smaller rectangles, one for each value calculated. These may again be subdivided, and either shaded in or annotated with a number. Shaded areas indicate that the processor is idle and not engaged in any computation - perhaps the computation is blocked, waiting for some needed value. An annotation, on the other hand, indicates that the processor is busy calculating a value. The number indicates which value it is computing. In *accum_scan* these values can be numbered from 1 to n, where n is the total number of values in the pipeline, and the annotations indicate which of these values is being calculated at the moment.

There are also arrows in these diagrams. These represent communication. Sometimes they are annotated with a number to indicate which value is being communicated, but sometimes these annotations are removed to simplify the diagram. The communication arrows are shown vertically, even though in reality communication takes up time. This is done to simplify the diagrams. Arrows going from the top processor in a diagram wrap round to the bottom processor, as in Figure 6.8. All numbers in the diagrams refer to values calculated by the processors, not input values.

The cost of calculating element i is dependent on the particular pipelining functions used. However this calculation needs i values and therefore is commonly $O(i)$. For example, in back substitution, i additions/subtractions and i multiplications/divisions are needed. For the sake of simplicity, this cost will be written as i in the following. The loss of a (usually small) constant factor makes little difference in the approximated calculations which follow.

Sequential and naive parallel distributions

The sequential distribution is perhaps the simplest. In it, all of the elements are stored in a single processor, and therefore the computation is the same as the equivalent sequential computation. This has the advantage that no communication is needed, but, of course, the disadvantage that no part of the computation can be proceed in parallel.

The cost of the function in this case is:

$$\sum_{j=1}^{n}(\text{cost of computing element } j) \approx \sum_{j=1}^{n} j = n(n+1)/2 \approx n^2/2.$$

The naive parallel distribution is a simple parallel distribution, which allocates one element to each processor. This is not usually realistic since large numbers of processors are needed. Nevertheless, calculations with this distribution can be found in textbooks such as [WA99], which calculates the optimised cost for back substitution, a common example of an incremental pipelined calculation (see Chapter 7), to be $O(n)$.

Blockwise

This cost analysis can be easily modified for a general pipeline with the blockwise data distribution as follows.

The behaviour of the processors is described in Figure 6.5. In the blockwise distribution, consecutive values are stored and calculated in the same processor, and so each processor can be seen to be working on consecutively numbered computations. For example, Processor P_0 works on elements 1, 2, 3 and 4.

These numbers represent values that are being calculated, using *accum_scanl*1, the left-hand version of *accum_scanr*1, specified in Section 6.2.2. For *accum_scanl*1 f $[x_1, \ldots, x_{12}]$, these values are:

$$\begin{array}{rcccl} \text{Result } 1 & = & res_1 & = & f\ x_1\ [] \\ 2 & = & res_2 & = & f\ x_2\ [res_1] \\ 3 & = & res_3 & = & f\ x_3\ [res_1, res_2] \\ \ldots & & & & \\ 12 & = & res_{12} & = & f\ x_{12}\ [res_1, \ldots, res_{11}] \end{array}$$

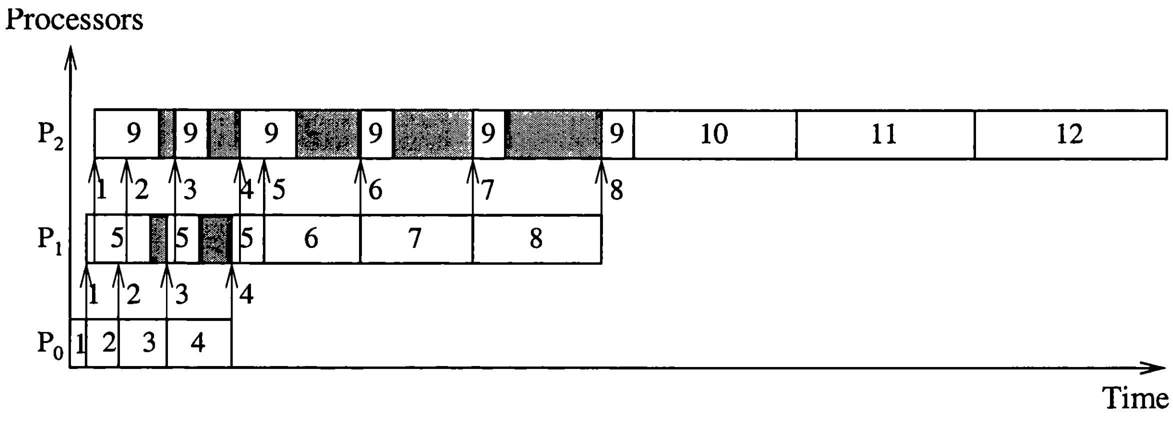


Figure 6.5: An unoptimised blockwise pipeline

Consider, for example, calculation 5. This requires result values 1, 2, 3 and 4 from processor 0. As we're assuming that the calculation can be done incrementally, it can start its calculation when it receives result 1. While it is doing this it receives result 2, and so can continue with the next part of the calculation. However, when it has finished dealing with result 2, 3 has not yet arrived. The processor lies idle for a little while, indicated in the diagram by a shaded area, until result 3 arrives. The same happens later as it waits for result 4.

This leads to some overlap between the calculations in different processors, but also to wasted time, waiting for values to arrive. The next subsection considers an optimisation to improve this.

This mixture of idle time and calculation only occurs for the first calculation in each processor as it waits for the results from the previous processor. Later calculations already know all necessary values: some were communicated from the previous processor for the previous calculation and some were calculated by the current processor.

The cost of this blockwise method can be calculated as the sum of all the calculation times and communication times minus the overlap when processor 1 and processor 2 start calculating their first results early. This overlap can be estimated generously as the total time for these values.

Communications are all done separately and each contains only one element, which is assumed to have unit size. Therefore the costs, in general, are (approximately):

$$\begin{array}{cc} \displaystyle\sum_{j=1}^{n} & (\text{cost of computing element } j) - \sum_{i=1}^{p-1}(\text{cost of the first element in } P_i) \\ & +\text{communication cost} \\ \approx & \sum_{j=1}^{n} j - \sum_{i=1}^{p-1}(n/p * i + 1) + \sum_{i=0}^{p-2}(n/p * (i+1)) * t_{cn} \end{array}$$

$$
\begin{aligned}
&= \quad n(n+1)/2 - n(p-1)/2 - (p-1) + n(p-1)/2 * t_{cn} \\
&\approx \quad n^2/2 - np/2 - p + np/2 * t_{cn}
\end{aligned}
$$

Optimised blockwise

The first version of the blockwise pipeline has quite a lot of wasted processor time, as shown by the shaded rectangles in Figure 6.5. The majority of the computation in each processor is done after all the computation in the previous processor. For example, the calculation of results 10 to 12 in processor 2 waits until the last value, 8, from processor 1 is received, even though processor 2 lies idle for much of the time before this.

However, as we have assumed that results can be calculated incrementally (see Section 6.6), this behaviour can be improved. The wasted time can be spent doing calculations for several elements in advance, not just for the first element. This is illustrated in Figure 6.6. Areas which were previously shaded are now annotated with calculations for further elements in the processor. This speeds up the calculation, but at the expense of space and the complexity of the program.

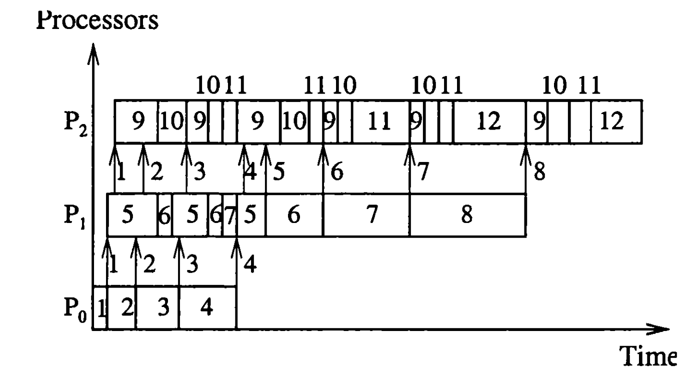


Figure 6.6: An optimised blockwise pipeline

This reduces the calculation time considerably, especially for processors later in the pipeline, so that now much less time is wasted. In most instances, a negligible amount of time is still wasted although it does depend on the number of elements in a block and on the particular calculations being done. Some calculations can be made incremental, but not very well, so that the non-incremental part still dominates.

For a well-behaved incremental function, such as that used in Gaussian elimination, the calculation cost is now approximately the same as that in an ideal case in which there is no idle time. Each calculation i takes the time its sequential implementation would take, given all necessary values, i.e., $O(i)$. This ideal case is illustrated in Figure 6.7.

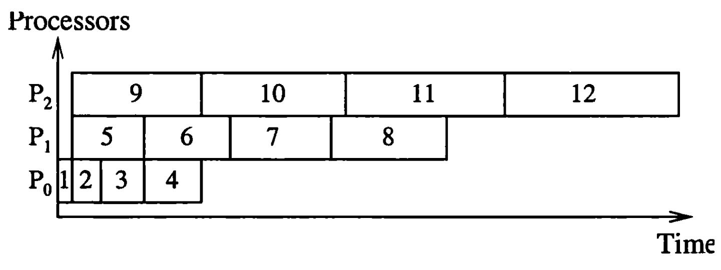


Figure 6.7: A blockwise pipeline with no idle time

Since the last processor in the pipeline has the largest amount of computation, the calculation cost is the time this processor spends in calculation, plus the initial cost of the first element. This latter is negligible, and the last processor in the pipeline can be assumed to be processor $p-1$ with n/p elements. Therefore the computation cost is approximately

$$\begin{array}{rl} & \sum_{i=0}^{n/p-1}(\text{cost of element } (n-i)) \\ = & n^2/p - n^2/2p^2 + n/2p \end{array}$$

Therefore the total cost is now approximately $n^2/p - n^2/2p^2 + n/2p + np/2 * t_{cn}$.

Cyclic

A cost analysis can also be done for the cyclic distribution. The behaviour of this pipeline is described in Figure 6.8, in which the communication arrows from P_2 wrap around to P_0.

As before, each stage in the pipeline receives the previously calculated results and passes them on to the next stage. However, this time consecutive stages aren't stored in the same processor. For example, stage 5 is done in processor 1. It receives results 1, 2, 3 and 4 from processor 0 and passes them onto processor 2, before calculating result 5 and passing that on too. But stage 2 has already passed results 1 and 2 to processor 2. There is much more communication than is necessary. The next subsection considers an optimisation which improves this.

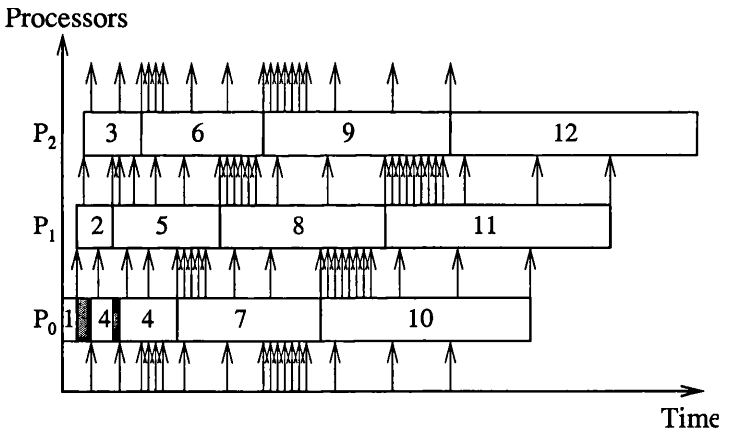


Figure 6.8: An unoptimised cyclic pipeline

The time spent in the communication is

$$\begin{array}{rl} & \text{volume of communication} * t_{cn} \\ \approx & \sum_{i=1}^{n-1} i * t_{cn} \\ \approx & n(n-1)/2 * t_{cn} \end{array}$$

There are very few shaded rectangles in the figure, and then only at the start, therefore most of the processor computation time is being used for calculation, and the calculation time can be estimated by assuming zero idle time as was done for the optimised blockwise version in Section 6.3.2. This is illustrated in Figure 6.9.

The cost calculation is fairly similar to that done in the previous section. The computation cost is approximately the cost of computation in the last processor in the pipeline which can be assumed

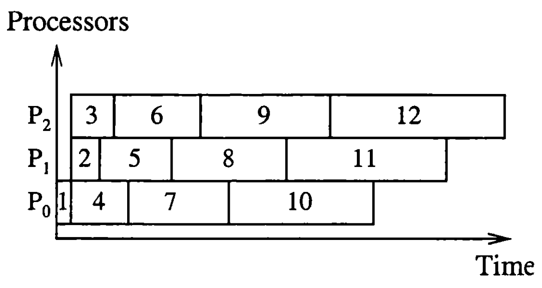


Figure 6.9: A cyclic pipeline with no idle time

to be processor $p-1$. This is

$$\begin{aligned} & \textstyle\sum_{i=1}^{n/p}(\text{cost of element } i * p) \\ \approx\ & p * \textstyle\sum_{i=1}^{n/p} i \\ =\ & p * (1 + n/p) * (n/p)/2 \\ =\ & n^2/2p + n/2 \end{aligned}$$

Therefore the total cost is (approximately) as follows:

$$\begin{aligned} & \text{Calculation cost} + \text{Communication cost} \\ \approx\ & n^2/2p + n/2 + n(n-1)/2 * t_{cn} \\ \approx\ & n^2/2p + n/2 + n^2/2 * t_{cn} \end{aligned}$$

Optimised cyclic

The first version of the cyclic pipeline has an excessive amount of communication, as can be seen by the large number of arrows in Figure 6.8. This is because *all* of the previously calculated values are passed on to the next processor at each step, even though the next processor has often seen several of these values before. This communication can be reduced by only sending values which have not already been seen as described in Section 6.8, and illustrated in Figure 6.10.

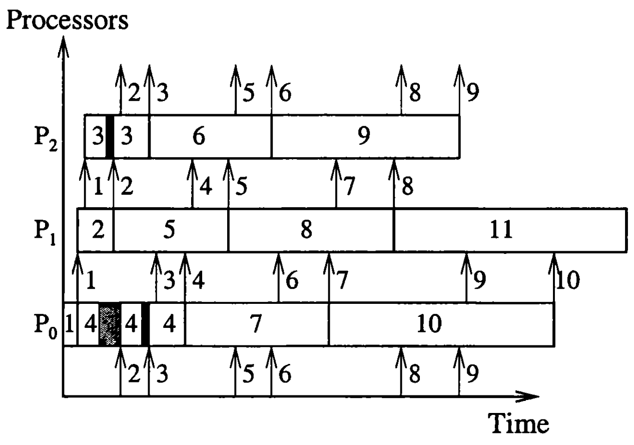


Figure 6.10: An optimised cyclic pipeline

At stage 5, processor 1 now receives results 3 and 4 from processor 0, but only passes on 4 and the value it calculates itself—result 5. Processor 2 has already seen results 1 to 3, and so 3 doesn't

Type of Pipeline	Approximate Cost
Sequential	$n^2/2$
One element per processor	$O(n)$
Basic Blockwise	$n^2/2 - np/2 - p + np/2 * t_{cn}$
Basic Cyclic	$n^2/2p + n/2 + n^2/2 * t_{cn}$
Optimised Blockwise	$n^2/p - n^2/2p^2 + n/2p + np/2 * t_{cn}$
Optimised Cyclic	$n/2 + n^2/2p + (np - n) * t_{cn}$

Table 6.2: Approximate costs for some different types of pipelines

have to be passed on.

This reduces the quantity of communication to about $(n - n/p) * p$, and hence the approximate total cost to $n^2/2p + n/2 + (np - n) * t_{cn}$.

6.3.3 Cost comparison

The costs of the different distributions can now be compared in order to determine which data distribution should be used. These costs, which were calculated in the previous section, are summarised in Table 6.2. The variable n is the number of data elements, and the other variables were described in Section 6.3.1.

In order to see which distribution leads to the most efficient program, we need to compare the best versions available. Therefore the optimised times are used. With both data distributions, the optimised program includes a communication cost, which involves a factor of t_{cn}, and a calculation cost, which is the rest of the cost.

The table shows that the blockwise version usually has a smaller communication cost, as $np/2$ is usually smaller than $np - n$. However its calculation cost is higher by a factor depending on n/p.

When the communication cost, t_{cn}, is high compared with n/p, it can outweigh the calculation cost, so that the blockwise implementation is more efficient. However, in other cases, the cyclic distribution is better.

The rest of this chapter focuses on the cyclic implementation. This is because when pipelining is put in combination with other parts of a program, it may be better to use the cyclic distribution not only when communication is cheap but when it's more expensive. This turns out to often be the case for Gaussian elimination (see Chapter 7).

6.3.4 A cyclic version

Based on the cost comparison, the cyclic distribution is used in the outer level of parallelism. The parallelism in the inner level is restricted to be sequential for the sake of simplicity and in order to fit with the Gaussian elimination study. All this requires in the program is a change to the annotations used on the functions, giving the following version of *accum_scanr*1:

```
$accum\_scanr1_{CycS}$ :: $(\alpha \to SeqFinSeq\,\beta \to \beta) \to Cyclic\,\alpha \to$
                      $Cyclic\,\beta$
$accum\_scanr1_{CycS}\ f\ zs$ =
        $map_{Cyclic}\ heads_S\ (init_{Cyclic}\ (scanr_{Cyclic}\ g\ []_S\ zs))$
    where
    $g\ x\ ys\ =\ (f\ x\ ys)\ :_S\ ys$
```

6.4 Pipelining

As mentioned in Section 6.2, pipelining is a method which can be used to speed up the parallel implementation of functions such as *scans* and *folds*. Each processor receives a value, does some calculation to it, usually involving its local memory, and then passes the resultant value on to the next processor in the pipeline, as illustrated in Figure 6.11. This speeds up calculations when results for several sets of data need to be calculated, as values can be fed into the pipeline one after the other. In special cases this technique can also be used to improve the performance when only one set of results is needed.

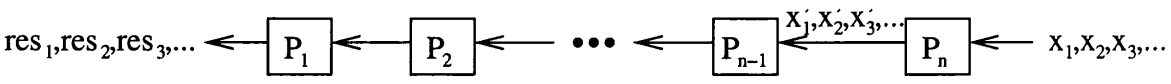


Figure 6.11: A general pipeline

There are special pipelining architectures, but pipelines can also be implemented efficiently on many general-purpose architectures. In particular, Cole shows how a pipeline can be implemented on grids of processors using only communications between neighbouring processors [Col89].

Section 6.2 discussed how a pipeline can be implemented for the general form of *scanr*. Since *accum_scanr*1 can be implemented using *scanr*, it can also use this form of a pipeline. It simply passes whole sequences of values between the processors instead of single values. However, it *is* worth-while writing *accum_scanr*1 as a separate function because its pipeline can be optimised in ways specific to it. These will be discussed later in Sections 6.5, 6.6 and 6.8.

6.4.1 Data distributions and pipelines

When we use a data distribution, there are several values in each processor, and therefore several calculations to be done in each processor. It is also often necessary for values to be communicated between the processors.

With a blockwise distribution, the calculations in a processor can be done one directly after the other since the values are in their original order. This is illustrated in Figure 6.12. The long arrows represent communication between processors and the short ones represent data dependencies within a processor. A more accurate description can be found in Section 6.3.2 and Figure 6.5.

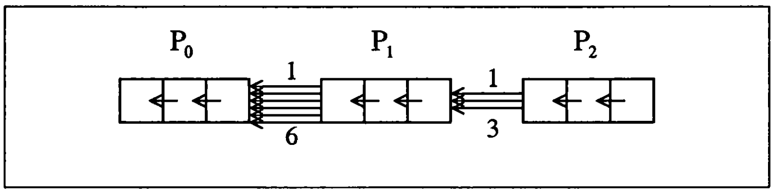


Figure 6.12: A blockwise pipeline

In cyclic pipelining, each processor also has to do several calculations. However, this time, the values in one processor aren't consecutive. A processor may have to wait for another processor to pass it a result before it can perform its next calculation. The pattern of communication in this case is shown in Figure 6.13.

As can be seen from these diagrams, the cyclic pipeline requires much more communication than the blockwise one. However, *accum_scanr*1 is a special case, and optimisations can be applied to

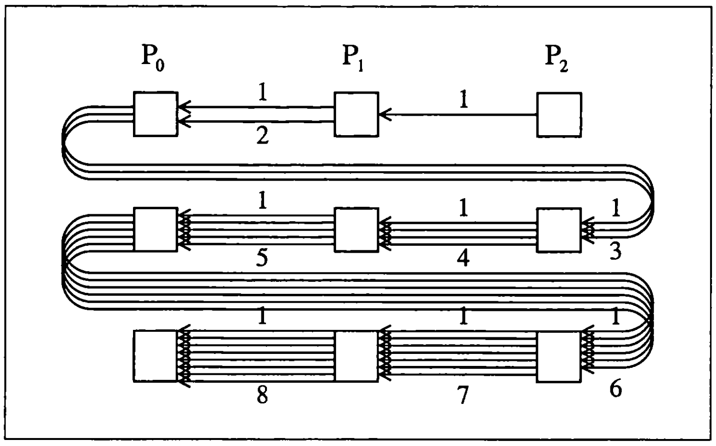


Figure 6.13: A cyclic pipeline

both pipelines which reduce the difference between them. This was discussed in the previous section (Section 6.3), when the decision about the data distributions was made.

6.4.2 A cyclic pipeline

A cyclic pipelined version of *accum_scanrl* uses MPI APM functions to make the communications in the pipeline explicit. The code for this can be based on the code for *scanr* since *accum_scanrl* can be written in terms of *scanr*. Standard transformation techniques (such as those in Chapter 3) can also be applied to aid the transformation from the fairly abstract version of the function in the previous section to this new version.

First of all, let's look at *scanr*. The code for this with no data distributions is given in Section 6.2. This can be modified to work with a cyclic distribution by simply replacing the MPI APM and variable manipulation functions with cyclic equivalents. For example, $retrieve_{indiv}$ becomes $retrieve_{Cyclicindiv}$ and *mpi_spt2pt* becomes $mpi\_spt2pt_{Cyclicindiv}$. These functions are specially designed to calculate the correct processor addresses and variable locations given the index of the value which we're using. They are described in greater detail in Chapter 2 in Sections 2.12.1 and 2.12.2. They can be easily modified to communicate sequences of values instead of single values.

Once this code is produced, it can be inlined into the code for *accum_scanrl*. After also introducing variables to hold intermediate values, the program is:

```
accum_scanrl_CycS :: (SeqFinSeq Float → IOPST(SeqFinSeq Float) → IOPST Float) →
          VarFn_Cyclic(SeqFinSeq Float) → VarFn_Cyclic Float → IOPST()
    — local_vals_v holds the initial sequence.
    — The result is returned in res_v.
accum_scanrl_CycS f local_vals_v res_v =
       do p' ← retrieve p_v
          p ← return (head_P p')   — allows p to be used as an integer
          n' ← retrieve matrix_size_v
          n ← return (head_P n')
          new_vals_v ← create_var_Cyclic (repeat_Cyclic p (empty_S :: SeqFinSeq Float))
```

```
let $g\ x\ ys$ = do $old\_vals \leftarrow ys$
                    $new\_val \leftarrow f\ x\ ys$
                    $return\ (new\_val :_S old\_vals)$
  — set up start of pipeline
$local\_vals \leftarrow retrieve_{Cyclicindiv}\ (n-1)\ local\_vals\_v$
$new\_vals \leftarrow g\ local\_vals\ (return\ empty_S)$
$store_{Cyclicindiv}\ (n-1)\ new\_vals\_v\ new\_vals$
  — pipeline
$for\ [n-2, n-3..0]$
  — for each value i, do its calculation in processor (i'mod'p)
$(\lambda\, i \rightarrow$
    do $mpi\_spt2pt_{Cyclicindiv}\ new\_vals\_v\ (ARRAY\ FLOAT)$
                         $(i+1)\ new\_vals\_v\ (ARRAY\ FLOAT)\ i$
       $new\_vals \leftarrow retrieve_{Cyclicindiv}\ i\ new\_vals\_v$
       $local\_vals \leftarrow retrieve_{Cyclicindiv}\ i\ local\_vals\_v$
       $new\_vals \leftarrow g\ local\_vals\ (return\ new\_vals)$
       $store_{Cyclicindiv}\ i\ new\_vals\_v\ new\_vals)$
  — the results are now stored cyclicly in new_vals_v but
  — with extra values.
$new\_vals \leftarrow retrieve_{Cyclic}\ new\_vals\_v$
$store_{Cyclic}\ res\_v\ (map_{Cyclic}\ head_S\ new\_vals)$
```

6.5 Some accum_scan optimisations

Some *scans* have special features which can be made use of to improve their implementations. The branching structure of the methodology supports this. It allows the general *scan* implementations to be used for the early levels of the derivation, and then later it allows the derivation to split, taking different paths for functions with different features.

In particular, *accum_scanr*1 can be improved. It sends whole sequences between processors, but only one of these values is new. The rest can be forwarded as soon as they are received.

This has the advantage that some of the communication is overlapped as shown in Figure 6.14, in which the rectangles represent the time spent in communication between the specified processors. This optimisation also means that early results propagate more quickly through the pipeline than before. This will be useful later when the calculations are done incrementally in Section 6.6.

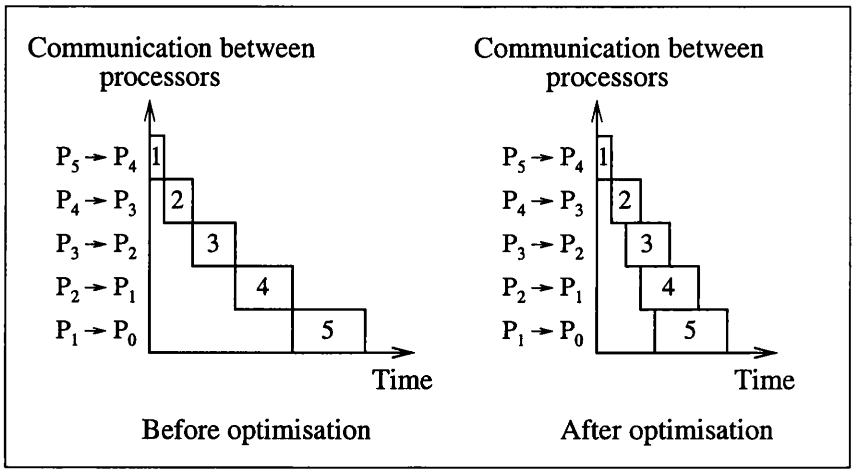


Figure 6.14: Optimisation of accumulating scan by overlapping communication

6.5.1 Individual level

The program is, first of all, transformed to use an individual level semantics as described in Section 2.11 of Chapter 2. This is done using the guidelines given in Section 3.9 of Chapter 3. This makes the introduction of the optimisation and further transformations easier because it simplifies the program and makes the communications easier to see.

At the individual level, the program contains many individual variable manipulations and send and receive pairings which only take place in one or two processors. It can be manipulated to place code for each processor together, since this makes it easier to see what's going on.

The main part of the code now deals with the start, end and main parts of the pipeline separately. In the body of the pipeline, it uses *startelt* and *endelt* to identify the items which the current processor has to process. It looks like this:

```
    — set up start of pipeline (in the last processor = (n-1)‘mod‘p)
  if $(pid == (n-1)`mod`p)$ then
      do $local\_vals \leftarrow retrieve_{indiv}\ ((n-1)`div`p)\ local\_vals\_v$
         $new\_vals \leftarrow g\ local\_vals\ (return\ empty_S)$
         $store_{indiv}\ ((n-1)`div`p)\ new\_vals\_v\ new\_vals$
         $mpi\_send'\ new\_vals\_v\ (ARRAY\ FLOAT)\ ((pid-1)`mod`p)$
                    $((n-1)`div`p)$
  else $return$ ()


    — pipeline (elements [n-2,n-3..0])
    — set up which elements to deal with in the current processor
  let $startelt$ = if $(pid > (n`mod`p) - 2)$ $then$ $n`div`p - 1$
                   else $(n`div`p)$
     $endelt$ = if $(pid > 0)$ then 0 $else$ 1
    — Run through those elements
  $for\ [startelt, startelt - 1..endelt]$
  $(\lambda\, x \rightarrow$
      do   — receive elements from the previous processor in the pipeline
         $mpi\_recv'\ new\_vals\_v\ (ARRAY\ FLOAT)\ ((pid+1)`mod`p)\ x$
           — access values; calculate and store new value
         $new\_vals \leftarrow retrieve_{indiv}\ x\ new\_vals\_v$
         $local\_vals \leftarrow retrieve_{indiv}\ x\ local\_vals\_v$
         $new\_vals \leftarrow g\ local\_vals\ (return\ new\_vals)$
         $store_{indiv}\ x\ new\_vals\_v\ new\_vals$
           — send on received values and new value to the next processor
         $mpi\_send'\ new\_vals\_v\ (ARRAY\ FLOAT)\ ((pid-1)`mod`p)\ x)$

    — end of pipeline
  if $(pid == 0)$ then
      do $mpi\_recv'\ new\_vals\_v\ (ARRAY\ FLOAT)$ 1 0
         $new\_vals \leftarrow retrieve_{indiv}\ 0\ new\_vals\_v$
         $local\_vals \leftarrow retrieve_{indiv}\ 0\ local\_vals\_v$
         $new\_vals \leftarrow g\ local\_vals\ (return\ new\_vals)$
         $store_{indiv}\ 0\ new\_vals\_v\ new\_vals$
  else $return$ ()
```

mpi_send' and *mpi_recv'* are versions of the individual level MPI functions *mpi_send* and *mpi_recv*. They take an extra parameter describing the given variable's location in the sequence stored in the processor.

```
Initial version
Each processor does the following:
     Receive previous values
     Calculate new value
     Attach new value to previous values
     Send all values
```

```
Using split communications
Each processor does the following:
     For each previous value
         Receive it
     Calculate new value
     For each previous value
         Send it
     Send new value
```

Figure 6.15: Pseudo-code for one stage in a pipeline, without and with split communications

6.5.2 Optimisation

The program can now be improved by overlapping the communication as mentioned at the start of this section. This can be done in two main stages.

First of all, the communicated sequences are split into their individual values which are sent and received one at a time instead of all at once as described in the pseudo-code in Figure 6.15. This transformation does not actually improve the program. It simply slows it down by introducing more communication messages. However, it paves the way for a second transformation that rearranges the communication so that values are sent on as soon as they are received, as described in the pseudo-code in Figure 6.16.

```
Each processor does the following:
     Initialise accumulating variable, new_vals_v
     For each previous value
         Receive it
         Send it
         Add it to new_vals_v
     Calculate new value using new_vals_v
     Send new value
```

Figure 6.16: Pseudo-code for one stage in a pipeline after communications have been moved

The previous version of the program received all of the values, calculated the new one and then sent on all of the old values plus the new one. Instead the new version receives each of the old values and sends it on immediately. It also accumulates these values in a new variable, *new_vals_v*, which can be used later to calculate the new value. Lastly the newly calculated value is sent on to the next processor in the pipeline.

This overlaps communication, and therefore often speeds up the program. However, it increases the number of messages sent (although not the volume of data). If the communication start-up cost is sufficiently high, this may still slow the program down. Nevertheless, even in this situation, the transformation is still useful as it prepares the way for later transformations.

6.5.3 Resultant code

The code for the main part of the pipeline after these transformations have been carried out is given below. In it the current processor only deals with one value, indicated by x. This code is contained in a loop which iterates through several values of x as indicated in Section 6.5.1. Note that to ensure correctness at this stage g must be altered to attach new values to the end of the sequence instead of to the start.

```
  — receive and send values one at a time
let $no\_to\_receive = (n-1) - (pid + p * x)$
$for$ $[0..no\_to\_receive - 1]$
$(\lambda j \rightarrow$
     do $mpi\_recv\ tmp\_v\ FLOAT\ ((pid+1)`mod`p)$
        $mpi\_send\ tmp\_v\ FLOAT\ ((pid-1)`mod`p)$
        $tmp \leftarrow retrieve\ tmp\_v$
        $new\_vals \leftarrow retrieve_{indiv}\ x\ new\_vals\_v$
        $store_{indiv}\ x\ new\_vals\_v\ (new\_vals\ +\!\!+_S\ (tmp\ :_S\ empty_S)))$
  — do calculations
$new\_vals \leftarrow retrieve_{indiv}\ x\ new\_vals\_v$
$local\_vals \leftarrow retrieve_{indiv}\ x\ local\_vals\_v$
$new\_vals \leftarrow g\ local\_vals\ (return\ new\_vals)$
$store_{indiv}\ x\ new\_vals\_v\ new\_vals$
$store\ tmp\_v\ (last_S\ new\_vals)$
  — send final value
$mpi\_send\ tmp\_v\ FLOAT\ ((pid-1)`mod`p)$
```

6.6 Incremental calculations

Sometimes there are optimisations which can only be applied to functions in certain circumstances, and this is true for *accum_scanr*. Sometimes its function parameter, f, has properties which allow some optimisations to be applied.

In other circumstances, the part of the derivation to do with those optimisations can be skipped, and the other transformations applied as usual. There might also be other transformations which could be applied. In pipelining, if f doesn't have the properties we're interested in, we can skip this section, and go straight to the transformations described in Sections 6.7 to 6.9.

Both versions of the function could actually be produced, resulting in 2 different sets of APMs. For a particular use of *accum_scanr*1, f would be tested to see whether it satisfies the properties, and thence the appropriate implementation chosen.

What is interesting here is this branching structure and the ability to perform detailed low-level optimisations, rather than the actual transformation. However the following details give a flavour of what is going on.

6.6.1 What is an incremental calculation?

There is an optimisation which can be applied if f can be split into several smaller parts, each of which needs only the current value in the sequence, i.e., if f can be calculated *incrementally.* This is useful because it allows the calculation and communication to be overlapped to a greater extent. In the blockwise distribution, it also allows the optimisation described in Section 6.3.2. This can be expressed formally if we modify f slightly.

We introduce a summation variable, *sum_v*, to store the current running value. This gives the incremental calculation somewhere to store the intermediate values. f should also expect a sequence with the newest values at the end instead of at the beginning, as this is the order in which values are received from the previous processor in the pipeline.

Given this, f can be split up into 3 parts, *init* which initialises *sum_v*, g which performs the incremental part of the calculation, and h which finishes the calculation off. Each of these can use some local values, *row*, in the processor, and g also has access to the current value, *current*, of the sequence of previously calculated values, and the index, j, of this value. There may also be other local values which the functions need to know. For example, h may need to know the index of the current data value.

```
init :: VarFn Float → SeqFinSeq Float → IOPST()
init sum_v row = ...

g :: Int → VarFn Float → SeqFinSeq Float → Float → IOPST ()
g j sum_v row current = ...
  — current is the value in the previously calculated values which
  — is currently being worked on
  — j is its index, starting from 1

h :: Int → VarFn Float → SeqFinSeq Float → IOPST ()
h rowno sum_v row = ...
```

These functions can be combined to form f as follows:

```
f :: SeqFinSeq Float → IOPST(SeqFinSeq Float) → IOPST Float
f row old_vals_m =
    do sum_v ← create_var (repeat_P (0 :: Float))
       old_vals ← old_vals_m
       n ← return((size_S row) − 1)
       i ← return(n − size_S old_vals)
       init sum_v row
       for [1..size_S old_vals]
         — These indices are increasing because
         — values are appended onto old_vals from the right.
       (λ j → g j sum_v row (old_vals !!_S (j − 1)))
       h (i − 1) sum_v row
       sum ← retrieve sum_v
       return sum
```

All this may seem rather contrived, but it arose out of considering a practical and common example, the Gaussian elimination algorithm in Chapter 7. The second part of this algorithm, back substitution, uses *accum_scanr*1 with a function which can be divided up as described above. This is shown in Section 7.6.2.

6.6.2 Interleaving calculation and communication

If the pipelined function is divided up into different parts like this, then each stage of the pipeline can proceed slightly differently. Previously the calculation waited until all the values had been received (and sent on). Now the calculation can start before any of these values are received and some of the calculation can take place while the processor is waiting to receive the next value. This can be seen in the pseudo-code in Figure 6.17.

6.6.3 Code

The code for each stage of the pipeline needs to change as indicated in the pseudo-code in Figure 6.17. This can be done in three main steps. First of all, the code for f changes to use *init*, g and

```
Before interleaving calculations and communications
        For   each value
              Receive it
              Send it
              Add it to new_vals_v
        Calculate new value using new_vals_v
        Send new value
```

```
After interleaving calculations and communications
  Initialise sum_v
  For   each value
        Receive it
        Send it
        Update sum_v using value and function g
  Finish calculating sum_v using function h
  Send its value
```

Figure 6.17: Pseudo-code for one stage in a pipeline that uses incremental calculations, before and after the calculations and communications are interleaved

h. This transformation can be proved correct for each particular instance of *f*. This code is then inlined into the program. Lastly, the different parts of *f* are moved around. According to various monadic laws, a selection of which are given in Appendix A.4, they can be moved as long as data dependencies are maintained. Therefore *init* can move before the communication as it doesn't use any of the communicated values. *g*'s `for` loop matches the communication `for` loop, and its data dependencies match the values received in this loop. Therefore it can be moved inside that loop. *h* is the only part which has to remain after the loop.

Therefore the code for a stage in the pipeline now looks like this:

```
...
init sum_v local_vals
for [0..no_to_receive − 1]
(λ j → do
            mpi_recv tmp_v FLOAT ((pid + 1)‘mod‘p)
            mpi_send tmp_v FLOAT ((pid − 1)‘mod‘p)
            tmp ← retrieve tmp_v
            g (j + 1) sum_v local_vals tmp)
h (pid + p * x) sum_v local_vals
sum ← retrieve sum_v
store_indiv x res_v sum
store tmp_v sum
  — send final value
mpi_send tmp_v FLOAT ((pid − 1)‘mod‘p))
```

6.7 Removal of explicit cyclic functions

At this point, the program can be transformed to use ordinary MPI and variable manipulation functions instead of cyclic ones. Doing this achieves two main things. It brings the program closer to the target language which, in this case, doesn't contain any functions for manipulating data distributions directly. It also makes it easier to get inside the distributions and perform optimisations which involve individual elements. It is possible to perform such optimisations with the cyclic structures in place, as the previous transformation showed. However, removing them often makes it easier.

The transformation itself is fairly straight-forward, following the standard rules given in Section 3.10.4. Many of the explicit cyclic references have already been removed, in the transformation to the individual level, and when the calculations were grouped together by stage in Section 6.5. The remainder are variable manipulation functions.

Example For example, in the main part of the pipeline, $store_{indiv}$ *x res_v new_val* is used to deal with just one of the many values stored in *res_v*. This can be changed to a sequence of operations, in which the old value of *res_v* is accessed, updated (using the sequence function, *replace*) and then re-stored.

```
res ← retrieve res_v
store res_v (replace_S res x new_val)
```

In C this could be implemented using standard array manipulation code.

```
res[x] := new_val;
```

This is also a good point at which to make sure that all sequence parameters are passed by reference, and to tidy up the code as in Section 3.5.1 in Chapter 3.

6.8 Optimisation of the cyclic pipeline

So far, both the cyclic and blockwise pipelines work in similar ways. However, their efficiencies are very different as shown in Section 6.3.2. At the moment, neither of the pipelines are optimised and the calculations show that the blockwise version wastes quite a bit of calculation time while it waits for values, but that the cyclic version is even worse. It communicates an enormous number of values - far more than is actually needed. Since communication is often much more expensive than calculation, this is serious.

Both of these pipelines can be improved. Since we've already chosen to use a cyclic distribution, the blockwise improvement isn't relevant at the moment. It is outlined in Section 6.3.2.

The first step in improving the cyclic version is to examine the expensive part of the program in more detail. As already observed, the program uses a lot of communication, far more than the blockwise version. But why?

Figures 6.12 and 6.13 from earlier in the chapter reveal part of the answer. In these diagrams each stage and its communications are shown separately and labelled to indicate which ones are which. In the cyclic case, in Figure 6.13, each stage receives all of the values from previous stages. A processor deals with several stages, and so it receives all of the required values for each of these stages separately, even though many of these values are repeated. In contrast to this, the blockwise distribution (shown in Figure 6.12), receives all of the required values together so that none are repeated.

Therefore one way to improve the performance of the program is to remove the unnecessary communications. This can be done by keeping hold of values which a processor has already seen, in previous stages, and only sending it new values.

6.8.1 Putting the idea into practice

The first step towards putting this idea into practice is the introduction of variables, *accum_v*, which keeps track of all the values seen so far, and *no_seen*, which keeps track of how many values are in *accum_v*. They can be introduced and modified without affecting the correctness of the program by Lemma 39.

```
accum_v ← create_var empty_S
no_seen_v ← create_var (0 :: Int)
```

The communications need to change to take this into account. Figure 6.18 shows which communications are now necessary. The numbers in the boxes indicate values which the processor knows because it has calculated them itself. A processor knows these values plus those which it has been sent.

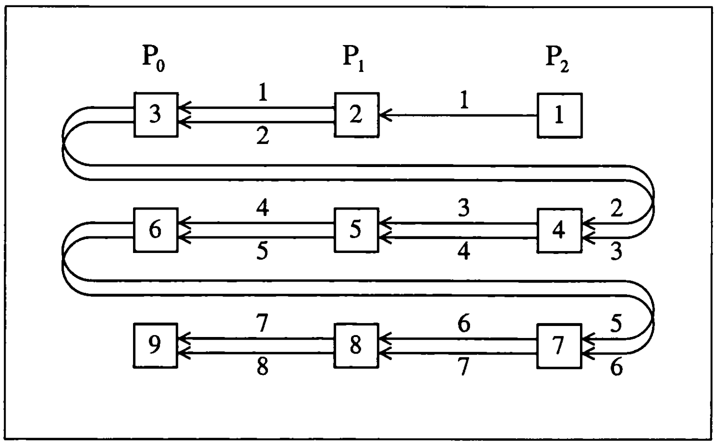


Figure 6.18: Communications in the optimised cyclic pipeline

Therefore the first pass through the processors remains the same as in the previous version, but on future passes things change. A processor sends on the values which it receives, except for the first of these values, plus the value it's just calculated. The first value is not needed because the next processor has already seen it. It's important to make sure that this first value received is not removed on the first pass through the processors, only on subsequent passes.

In order to maintain the correctness of the program once these communications are removed, the *g* calculations with the removed values must still be performed. They can be done before the first values are received using the values from *accum_v*. In order to prove that this transformation is correct, it's necessary to show that the values in *accum_v* are indeed the values which were received in the previous pipeline. This is relatively easy as each value of accum_v is set only once and not changed.

A stage in the pipeline now looks like the following. ... represents code which is either straightforward or the same as before. The initial pass through the pipeline is indicated by *no_seen* $= 0$.

```
...
  — initialise and do g for values already seen
init sum_v tmp_local_vals_v
accum ← retrieve accum_v
for [0..no_seen − 1]
(λ i → g (i + 1) sum_v tmp_local_vals_v (accum !!_S i))

  — receive and send values one at a time, doing the rest of g
  — first the number of values to receive is calculated
if (no_seen == 0) then store no_to_receive_v ((n − pid − 1)`mod`p)
else store no_to_receive_v (p − 1)
no_to_receive ← retrieve no_to_receive_v
for [0..no_to_receive − 1]
(λ j →
        do $mpi\_recv\ tmp\_v\ FLOAT\ ((pid + 1)`mod`p)$
            — only send on the first val on the first pass
          if $(j == 0)\&\&((no\_seen > 0)||$
            $((no\_seen == 0)\&\&(pid == n`mod`p)))$ then $return\ ()$
          else $mpi\_send\ tmp\_v\ FLOAT\ ((pid - 1)`mod`p)$
            — set accum and no_seen
          ...
          $tmp \leftarrow retrieve\ tmp\_v$
          $store\ accum\_v\ (accum +\!\!+_S (tmp :_S []_S))$
            — g
          $g\ no\_seen\ sum\_v\ tmp\_local\_vals\_v\ tmp)$
    — final calculation
    ...
    — send final value
    $mpi\_send\ sum\_v\ FLOAT\ ((pid - 1)`mod`p)$
    — store value in accum
    ...
    $store\ accum\_v\ (cat_S\ accum\ (cons_S\ new\_val\ empty_S))$
```

6.9 C+MPI implementation

All that now remains is to convert this version of the program into C+MPI. This can be done using the transformation rules given in Section 3.11.2.

Many of the explicit *stores* and *retrieves* can be removed or tidied up, especially when using index arithmetic on arrays. For example,
store accum_v $(accum +\!\!+_S (new\_val :_S []_S))$ becomes
`accum[no_seen-1] = sum;`

We also need a user-defined version of *mod*. C does provide % but it doesn't work properly for negative numbers.

The C+MPI program can be found at http://www.dcs.gla.ac.uk/~joy/research/thesis where it is used as part of the Gaussian elimination program.

6.10 Summary

One of the *scan* family of functions, *accum_scanr*1, can be implemented using a pipeline. However, as it stands, that pipeline is often not very efficient. Using the methodology it is possible to introduce a series of optimisations to the function, including optimisations which are detailed, low-level and involve the manipulation of communication operations.

Different versions of the function can be incorporated into different APMs and used in different situations. Some optimisations can also only be applied when certain conditions hold. This leads to a branching structure of versions of the function, matching and refining the branching structure of the APMs.

This function, *accum_scanr*1, is an APM function, and demonstrates part of the usefulness of APMs. They separate out computation, and can later be used without worrying about the functions' parallel details, or about how to transform or optimise them. The version in one APM simply transforms into its version in the next. This is demonstrated in the case study in the following chapter.

Chapter 7

Case Study: Gaussian Elimination

7.1 Introduction

The previous case study (Chapter 5) looked at the basics of the methodology, and a little at the issue of load-balancing. The case study presented in this chapter, Gaussian elimination, investigates some further aspects of the methodology. In particular, this case study:

- places the methodology under a bit more stress with a larger and more complicated example, and demonstrates that it can work on practical examples.
- investigates data distributions in practice. In particular, it looks at how they can be used to help load-balancing, and at techniques for dealing with the situation when the optimal data distributions in different parts of the program vary. The use of explicit data distributions also affects the way in which variables and MPI APM functions are handled, and this case study enabled the development of methods for dealing with this, as well as of ways to transform programs with explicit data distributions into programs which use ordinary operations to manipulate them. These methods are described generally in Chapter 3.
- examines parallel *folds* and *scans* in more detail. In particular, it looks at pipelining for a particular form of *scan*. This work is described in Chapter 6, and used in this chapter.
- deals with an example in which subsidiary functions are used extensively. When such functions are used as parameters to higher-order functions, the situation becomes more complicated. The use of a specific example helped to work out the details. (See Section 7.4.)
- tries out some optimisations that deal with imperative features such as the efficient use of variables.

7.1.1 The study

Gaussian elimination is a numerical mathematical method for solving a system of n linear equations in n variables. Such a system of equations looks like:

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \dotfill \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array}$$

where the a_is and b_is are constants, and the x_is are variables. The values of these variables are unknown, and the purpose of the algorithm is to discover them.

The system of equations can also be written as a matrix equation, $\mathbf{A}\underline{x} = \underline{b}$ where $\mathbf{A} = (a_{ij})$, $\underline{x} = (x_i)$, and $\underline{b} = (b_i)$.

In most cases[1], there is a unique value of $\underline{x}$ which satisfies these n equations. Gaussian elimination is one method of discovering this value. It converts the set of equations using linear transformations, multiplying equations and adding them together. This results in another set of equations which is in upper triangular form, i.e., it looks like:

$$\begin{array}{rrrr} a'_{11}x_1 + & a'_{12}x_2 + & \cdots + & a'_{1n}x_n = b'_1 \\ & a'_{22}x_2 + & \cdots + & a'_{2n}x_n = b'_2 \\ & & \ldots\ldots\ldots & \ldots\ldots\ldots \\ & & & a'_{nn}x_n = b'_n \end{array}$$

This can be easily solved by starting with equation n and using it to calculate the value of x_n, and then using this value in equation $n-1$ to calculate the value of x_{n-1} and so on for all the equations.

The first part of this process is often called *forward elimination* and the second part *back substitution*. This method is described formally and in more detail in the following section. More information about it can be found in many algorithms textbooks such as [CLR90]. Some of these describe variants of it, such as "LU Decomposition". These vary slightly from the method used in this chapter, but they are very closely related. There are also techniques, such as partial pivoting, which can be used to improve the algorithm. However, these are not considered here, as they add little to the main aims of this chapter. The parallelisation of Gaussian elimination is discussed in books on parallel algorithms such as [WA99] or [Rob90].

7.1.2 Layout of this chapter

This chapter presents the case study by examining each of the key transformation steps in turn. It does not go into the same level of detail as the map-triangle case study in Chapter 5, as to do so would be to add little that the previous study has not already shown. Instead, at each stage, the interesting parts are highlighted. These are parts which have not been previously demonstrated, and which illustrate key or intellectually interesting or challenging things. The code for each of the main stages can be found at http://www.dcs.gla.ac.uk/~joy/research/thesis.

7.2 Creating a Haskell specification

The first part of the derivation is the specification of the problem and its transformation into Haskell. This chapter aims to give some insight into this process. First of all, the maths specification is given and then we examine various facets of the conversion process. The process is described in general in Section 3.3.

7.2.1 Maths specification

The mathematics specification for Gaussian elimination takes as input a matrix $\mathbf{A} = (a_{ij})$ and a vector $\underline{b} = (b_i)$, as described above, and returns the vector $\underline{x}$ which solves the system of equations.

It is first described fairly abstractly in terms of row operations, and then refined to deal with operations on individual elements of the matrix and vector.

[1]i.e., when the equations are linearly independent

Row operations version

```
1. Forward elimination:
   For  $j = 1$ to $n - 1$
        Place 0 in columns $k \leq j$ of rows below row $j$:
        For  $i = j + 1$ to $n$
             Put 0 in column $j$ in row $i$:
             row $i \rightarrow -\frac{a_{ij}}{a_{jj}} *$ row $j$ + row $i$
             (row of both $A$ and $b$)

2. Back substitution:
   For  $i = n$ to 1
        Calculate the value of $x_i$ using the values of $x_{i+1}, \ldots, x_n$:
        $x_i = (b_i - (a_{i(i+1)} * x_{i+1} + \ldots + a_{in} * x_n))/a_{ii}$
```

Individual elements version

```
1. Forward elimination:
   For  $j = 1$ to $n - 1$
        For  $i = j + 1$ to $n$
             For  $k = 1$ to $n$
                  Calculate the new value of element $k$ of row $i$:
                  $a_{ik} = -\frac{a_{ij}}{a_{jj}} * a_{jk} + a_{ik}$
             $b_i = -\frac{a_{ij}}{a_{jj}} * b_j + b_i$

2. Back substitution:
   For  $i = n$ to 1
        $x_i = (b_i - (a_{i(i+1)} * x_{i+1} + \ldots + a_{in} * x_n))/a_{ii}$
```

This could be further modified if we note that a_{lm} is set to zero for $l > j - 1$, $m \leq j - 1$ by the $j-1$th iteration of the outermost loop. Therefore on the jth iteration, a_{jk} and a_{ik} are already zero for $i \geq j$, $k < j$. This means that for these values of i and k the calculation is not necessary—it merely sets a_{ik} to a value it already has. We can therefore restrict the loops so that they don't calculate these values. This does not affect the i loop, but the innermost loop becomes, "For $k = j$ to n".

7.2.2 Practicalities

When converting the specification into Haskell, there are some practicalities to be taken care of. First of all, types must be chosen for the data. As in Section 2.4, finite sequences are used to represent the vector and matrix data. In the following, however, we use lists for simplicity of presentation. These lists should be assumed to work in the same way as finite sequences.

```
type Vector α = [α]
type Matrix α = [[α]]    — list of rows
```

It is also useful to deal with **A** and $\underline{b}$ as a single unit because the same operations are carried out on the ith element of $\underline{b}$ as on the ith row of **A**. This element is therefore attached to the end of this row as follows:

```
matrix = zipWith join a b = map2 join a b :: Matrix α
  where

join xs y = xs ++ [y]
```

7.2.3 Standard conversions to Haskell

Haskell specifications can be produced from maths specifications using the techniques given in Section 3.3 of Chapter 3, especially by referring to Table 3.1. However, this table is not complete. This section focuses on one missing aspect of this table, namely incremental `for` loops.

This is done in order to extend the transformations provided so that the basic building blocks for numerical algorithms are in place. In particular, this allows the Gaussian elimination case study in this chapter to be carried through.

It also demonstrates the techniques used when working out such transformations. Sometimes it is necessary to convert a non-standard maths specification, and it may be necessary to work out the transformations for oneself.

Incremental `for` loops were chosen because such loops are commonly used in maths specifications, when one calculation depends on previous ones.

Producing a new conversion - Example

A very simple and common example of a loop involves the generation of a single value, which depends at each iteration on its previous value. This is often expressed as follows:

```
$res = a$
For  $i = 1$ to $n$
        $res = f\ res\ other\_parameters$
```

where *res* is a variable holding the result, *a* and *n* are constants, *other_parameters* is commonly the *i*th value from a list, and $xs = [xs_1, \ldots, xs_n]$:

```
$res = a$
For  $i = 1$ to $n$
        $res = f\ res\ xs_i$
```

Unwinding this, we get:

$$\begin{aligned} res &= f\ (previous\ res)\ xs_n \\ &= f\ (f\ (previous\ res)\ xs_{n-1})\ xs_n \\ &\ldots \\ &= f\ (f\ (\ldots\ (f\ a\ xs_1))\ xs_{n-1})\ xs_n \end{aligned}$$

We can then look through the set of standard Haskell functions to see if there are any which produce this output given this input. *foldl* looks promising; on *xs* it gives:

$$foldl\ f\ a\ xs\ =\ f\ (f\ (\ldots\ (f\ a\ xs_1))\ xs_{n-1})\ xs_n$$

Therefore this particular form of a `for` loop can be converted into an application of *foldl*.

New functions

However, sometimes no standard Haskell function captures the required pattern of computation. Then one needs to define a new function which does.

At other times it is possible to express the pattern of computation using existing functions, but only by manipulating the parameters and obscuring what's actually going on. This also often leads to the same Haskell function being used for multiple patterns of computation, but it is often preferable to keep to one function per pattern of computation. This makes the program clearer and more understandable. It also makes it easier to develop specialised implementations of functions if they are focussed on a single communication pattern.

An example of this is given in row 4 of Table 7.1. The loop produces a list of results as in *scanr*1 (row 3, Table 7.1), but this time each result depends not just on the latest result, but on all the previously generated results.

No.	Mathematics	Haskell
For loops in which iterations depend on previous iterations These hold for any $xs = [xs_1, \ldots, xs_n]$, constant a, variables res, res_i $(i = 1, \ldots, n)$		
1	ith iteration depends on $(i-1)$th $res = a$ for $i = 1..n.\ res = f\ res\ xs_i$	$res = foldl\ f\ a\ xs$
2	ith value depends on $(i-1)$th $res_1 = xs_1$ for $i = 2..n.\ res_i = f\ xs_i\ res_{i-1}$	$res = scanl1\ f\ xs$
3	ith value depends on $(i+1)$th $res_n = xs_n$ for $i = (n-1)..1.\ res_i = f\ xs_i\ res_{i+1}$	$res = scanr1\ f\ xs$
4	ith value depends on jth value $(j > i)$ $res_n = xs_n$ for $i = (n-1)..1.$ $res_i = f\ xs_i\ [res_{i+1}, \ldots, res_n]$	$res = accum\_scanr1\ f\ xs$

Table 7.1: Transformation guidelines for specifications with incremental loops

The similarity between these 2 specifications is obvious simply by examining the table. If they can be made to match, then *scanr* can be used to implement row 4 as it is with row 3. However, it is still better to encapsulate the implementation for this pattern of computation in a separate function. A specific pattern of communication can be used later in the derivation because this function passes several result values between processing sites.

Let us call this new function *accum_scanr*1 to indicate its accumulative nature. It can be defined in 2 main ways, either directly, for example by recursion, or by using the similarity of row 3 and row 4 to define it in terms of *scanr*.

A recursive implementation notes that the result of the function, the list of all calculated values, is the value of the most recent result (calculated using f) attached to the list of all previously calculated values. Therefore it can be written using pattern-matching as follows:

```
accum_scanr1 :: (α → [β] → β) → [α] → [β]
accum_scanr1 f [] = []
accum_scanr1 f (z : zs) = (f z ys) : ys
  where
    ys = accum_scanr1 f zs
```

A version of this function written using *scanr* is given in Section 6.2. These versions can be proved equivalent. This then allows standard properties of *scan* to be used in the derivation.

The rest of the for loop conversions

Transformations for other kinds of `for` loops can be produced in a similar way. Some of these are summarised in Table 7.1.

7.2.4 Applying the standard conversions

Once such standard transformations are available, they need to be applied. However specifications may use nested loops and combinations of functions which make these applications non-obvious. It helps to separate out disjoint parts of the specification and to separate out the insides of loops into new functions as shown for the case study below.

In this study, at the algorithm divides naturally into two main parts, forward elimination and back substitution, as described in Section 7.1. It simplifies the conversion to consider these parts separately.

Forward elimination

The conversion of the matrix to upper triangular form consists of several nested loops. Each loop can be examined in turn, and encapsulated in a Haskell function as follows.

Innermost loop :
The innermost loop normalises one row of A and b (i.e., of the combined matrix), by placing 0 in its jth position as shown in Figure 7.1.

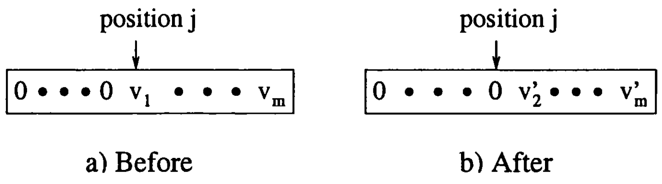


Figure 7.1: The effect of Gaussian elimination's innermost loop on one row of a matrix

The Haskell function for this loop must, therefore, take the relevant row as a parameter, and can be called *put_0_in_jth_pos*. It is specified as follows:

```
put_0_in_jth_pos row
 =
For  $k = 1$ to $n$
     Calculate new value of $a_{ik}$:
     $a_{ik} = -\frac{a_{ij}}{a_{jj}} * a_{jk} + a_{ik}$
$b_i = -\frac{a_{ij}}{a_{jj}} * b_j + b_i$
 =
For  $k = 1$ to $n + 1$
     $m_{ik} = -\frac{m_{ij}}{m_{jj}} * m_{jk} + m_{ik}$
where $\mathbf{M} = (m_{ij})$ is the combined matrix of $\mathbf{A}$ and $\underline{b}$.
 =
For  $(x, y) \leftarrow zip\ row\ (row\ j)$
     $x = pivot * y + x$
where $pivot = -\frac{m_{ij}}{m_{jj}}$ and $row = row\ i, row\ j$ are rows of $M$.
```

Each iteration is independent of the others, so we can now apply transformation 5 in Table 3.1. Notes that index j becomes $j - 1$, because Haskell list indices start from 0.

```
 =
map2 join row (m!!(j - 1))
where join x y = pivot * y + x
      pivot = -(row!!(j - 1)/m!!(j - 1)!!(j - 1))
```

Inner loop : *put_0_in_jth_pos* is nested inside another `for` loop which places 0 in columns $k \leq j$ of rows below row j. This is done by changing values in the jth column to 0, as shown in Figure 7.2.

This loop needs to know at least the values of the rows it changes, i.e., rows $(j + 1)$ to n of M. This can be achieved by passing it the whole of M which it then modifies and returns. It also needs to know j, the index of the current row.

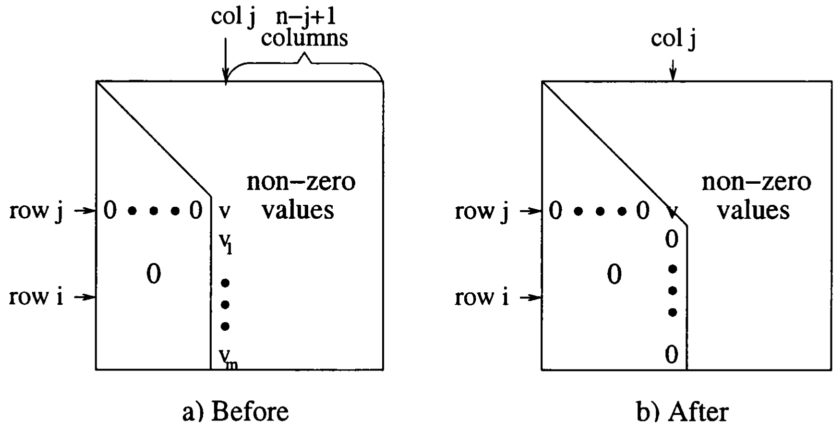


Figure 7.2: The effect of Gaussian elimination's inner loop on a matrix

```
deal_with_jth_row m j
=
For    i = j + 1 to n
           put_0_in_jth_pos (row i of M)
```

As before, each iteration is independent of the others, so this can be converted into a combination of *map* and *drop*, using transformation 9 in Table 3.1.

```
=
(take j m) ++ (map put_0_in_jth_pos (drop j m))
```

Outer loop : *deal_with_jth_row* is nested inside yet another `for` loop. This loop is passed the combined matrix, and converts it to upper triangular form. It forms the whole of the first part of the algorithm, and has the following form:

```
For    j = 1 to n − 1
           deal_with_jth_row m j
```

This loop is incremental, each iteration depending on the result of the last. Therefore transformation 1 in Table 7.1 applies to it, converting it into a *foldl* loop with f = *deal_with_jth_row*, $a = m$ and $xs = [1..n-1]$, the list of row indices.

```
=
foldl deal_with_jth_row m [1..n − 1]
```

Back Substitution

The second part of the Gaussian elimination algorithm uses back substitution on the upper triangular matrix to calculate the solution to the system of equations. It also consists of a `for` loop:

```
For    i = n to 1
           $x_i = (b_i - (a_{i(i+1)} * x_{i+1} + \ldots + a_{in} * x_n))/a_{ii}$
               $= (m_{i(n+1)} - (m_{i(i+1)} * x_{i+1} + \ldots + m_{in} * x_n))/m_{ii}$
```

In this loop, the ith value depends on the jth values for $j > i$. Therefore, transformation 4 in Table 7.1 can be applied to it, to produce a program using *accum_scanr*.

The function parameter of *accum_scanr* uses the previously calculated values, $x_{i+1}, \ldots, x_n$, and the current row (row i) of the matrix to calculate the next value of $\underline{x}$. This function has type,
row of M → sequence of previous values of x → a new x value,
i.e., *solve_ith_eqn* :: *Vector* $\alpha \rightarrow [\alpha] \rightarrow \alpha$.

It can be written in Haskell as follows:

```
solve_ith_eqn row previous
= $x_i$
= $(m_{i(n+1)} - (m_{i(i+1)} * x_{i+1} + \ldots + m_{in} * x_n))/m_{ii}$
where row      = $[m_{i1}, \ldots, m_{i(n+1)}]$
      previous = $[x_{i+1}, \ldots, x_n]$
= $(row_{n+1} - (row_{i+1} * previous_1 + \ldots + row_n * previous_{n-i}))/\ row_i$
= $(row!!n + (-row!!i * previous!!0) + \ldots +$
  $(-row!!(n-1) * previous!!(n-i-1)))/\ row!!(i-1)$
= $(foldl\ (+)\ (row!!n)\ (map2\ f\ (drop\ i\ row)\ previous))/row!!(i-1)$
where $f\ x\ y = -x * y$
      $i = n - (n - i) = n - length\ previous$
```

7.2.5 Haskell specification

Putting all this together, the Gaussian elimination specification looks like this:

```
gauss :: Num α ⇒ [[α]] → [α] → [α]
gauss a b =
    let m = join a b
        where join xs y = xs ++ [y]
        n = length b
        m2 = foldl deal_with_jth_row m [1..n − 1]
        x = accum_scanr1 solve_ith_eqn m2
    in
    x

deal_with_jth_row :: Num α ⇒ [[α]] → Int → [[α]]
deal_with_jth_row m j =first_j_rows ++
                         (map put_0_in_jth_pos other_rows)
  — makes rows below row j have 0 in columns i ≤ j
where first_j_rows = take j m
      other_rows = drop j m
      put_0_in_jth_pos row = map2 join row (m!!(j − 1))
      where join x y = multiplier * y + x
            multiplier = −((row !! (j − 1))/((m !!(j − 1))!!(j − 1)))

solve_ith_eqn :: Num α ⇒ [α] → [α] → α
solve_ith_eqn row previous =
 (foldl (+) (row!!n)(map2 f (drop i row) previous))/(row!!(i − 1))
where n = length row − 1
      i = n − length previous
      f x y = −(x * y)
```

7.2.6 Some notes about parallelism

Although all of the data *can* be manipulated in parallel, whether this is actually done or not depends on how good a parallel implementation can be obtained for the functions which operate on it. This is fairly straightforward for functions like *map*2, but more complicated for those like *foldl*. As mentioned in Section 3.4.1, *foldl* is usually only parallelisable when its function is associative. There are various transformation techniques for converting *fold*s to use associative functions (see [O'D94]), but these do not belong within the scope of this thesis, and *foldl* will be implemented sequentially in this case study.

Therefore, in forward elimination, the outer loop is sequential, but the inner and innermost loops can be parallel, because they use *map* and *map*2. The first operates over the rows of the matrix, and therefore these rows are stored in different processors. However the innermost loop operates on individual rows. These can still be stored in different processors, but this does not achieve much compared with the benefit gained in the outer level of parallelism, and increases the complexity of the program and the cost of communication.

In back substitution, at first glance, the data dependencies involved in *accum_scanr*1 mean that a parallel implementation is overly expensive. However, this function can be optimised, as shown in Chapter 6, so that parallel versions are faster provided communication isn't too expensive. Therefore the rows of the matrix are also stored in different processors for back substitution.

This choice of parallelism can be reflected in the data types used:

$gauss :: Num\ \alpha \Rightarrow ParFinSeq(SeqFinSeq\ \alpha) \rightarrow ParFinSeq\ \alpha \rightarrow ParFinSeq\ \alpha$

The next question concerns the particular way in which the rows are distributed across the processors.

7.3 Choosing the data distributions

Data distributions are used for a variety of purposes (see Section 2.7). Principally, they enable larger quantities of data to be dealt with than would otherwise be the case, and they can help to balance the load of data across the system. A good load balance is very handy because it helps to speed up the program (see Section 4.4), often by a significant amount.

7.3.1 Load balancing

In Gaussian elimination, there is a bad load balance in the first part of the algorithm, the forward elimination phase. This is because *deal_with_row_j* only involves row j and the rows below it. Nothing happens to the rows above it, and so any processors which operate solely on these rows have nothing to do. This row number j also increases during the program, and so the number of processors with nothing to do also increases.

However *deal_with_row_j* is a simple function which only uses simple arithmetic operations, addition and multiplication. Therefore forward elimination is not computationally intensive unless n is high. The load balance is not very bad for low values of n, but in general parallel programming is used when n is high, and therefore when the load balance is indeed bad.

Load balancing using a data distribution

There are two main reasons for using data distributions for this algorithm. Firstly, Gaussian elimination is usually performed on large systems of equations and therefore it is likely that there are less processors than data, and so multiple pieces of data in the same processor. This means that some kind of data distribution will be needed. Secondly, the load imbalance is very regular. Data distributions are often used in situations like this to balance the load (See Section 4.4).

In order to decide which particular data distributions will be best, it is necessary to examine the pattern of the load imbalance in more detail. Figure 7.3 shows the usage on each row. There is a triangular load imbalance. This is because row i is used once for each $j < i$. This suggests that a cyclic or block-cyclic distribution would work best (see Section 4.4).

Aside on direct load balancing

If we assume that the parallel machine is sufficiently large, then only one row will be dealt with in each processor. In this unusual case, it is necessary to divide tasks up in order to balance the load (see Section 4.4).

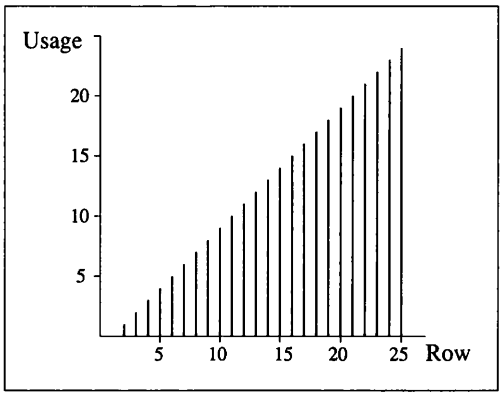


Figure 7.3: The load imbalance in forward elimination

A task deals with a single row:

$$put\_0\_in\_jth\_pos_S\, row = map2_S\ join\ row\ (m!!(j-1))$$

This can be split up since the parts of *map*2 are independent and can be evaluated in parallel. However there are also other factors to take into consideration, as the value of the pivotal element, row_{j-1} needs to be known by all parts of the row, necessitating extra communication. Also the load balance changes in each iteration of the loop, so using the same load balance each time through the loop is likely to make the load balance worse in some iterations, especially the early ones. We need to be careful to outweigh this imbalance by either creating a large enough improvement in later iterations, or by moving data between processors in each iteration.

This can be calculated using similar methods to the map-triangle example (Chapter 5), but it is more complicated. As the case is not common, it is not considered here.

7.3.2 Interaction of distributions

There are two main parts to the Gaussian elimination algorithm. As mentioned previously, forward elimination can be load balanced using a cyclic data distribution. Back substitution is usually optimised using pipelining techniques as shown in Section 7.5.3. These tend to work more efficiently with blockwise distributions, as shown in Section 6.3.

This produces a conflict between global and local optimality. Such conflicts are common when choosing data distributions and ways of solving them are presented and discussed in Section 4.5. They can be applied to this problem.

To make the situation simpler, let us restrict our attention to blockwise and cyclic distributions. Block-cyclic distributions, in which blocks of values are distributed cyclicly, are also common. Checkerboard distributions have also been used for Gaussian Elimination (see, for example, [WA99]). In these the matrix is partitioned into rectangular sections, with parts of several rows and columns but no complete rows or columns. However, two distributions is enough to illustrate the principles behind the decision-making.

There are three main possibilities. A distribution which is best for one of the parts of the algorithm can be used for all of it. Alternatively, cyclic can be used for the first part, and blockwise for the second with a data redistribution between the parts. In general, it would also be possible to use a different distribution for all of it, avoiding any redistributions and large inefficiencies, although not achieving optimal performance in either part of the algorithm.

The next step is to examine the efficiencies of these distributions for the two parts and the cost of the redistribution.

7.3.3 Cost calculations

A cost model is needed if we are going to calculate costs. This section uses a simple cost model which is described in Section 6.3.1. n is the number of rows, and p is the number of processors. So in both data distributions there are about n/p rows and hence about n^2/p elements in each processor. Other variables are given in Section 6.3.1.

Forward Elimination

The forward elimination process consists of multiple calls to *deal_with_jth_row*, which transforms the matrix as shown in Figure 7.4. It is used repeatedly, for $j = 1, \ldots, n-1$, until an upper-triangular matrix is produced.

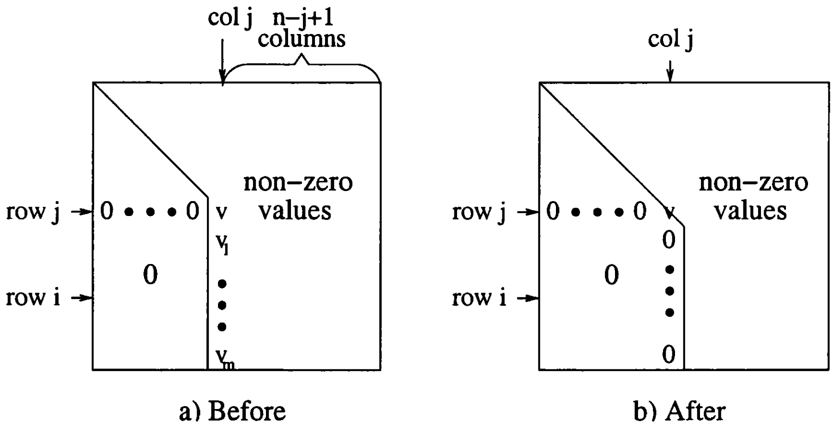


Figure 7.4: The effect of deal_with_jth_row on a matrix.

deal_with_jth_row works as follows:

• It performs row operations on each of the rows $i = j \ldots n$. As shown in Figure 7.4, each of these rows has $n - j + 1$ non-zero elements. Therefore a row operation on each of them takes $n - j + 1$ time.

• Each of these rows must know the value of row j in order to perform this row operation. This will involve communication. It can be done with point-to-point communications, but a single-node broadcast is usually cheaper. This would cost $(1 + \log p)((n + 1)t_c + \tau)$.

So far this is the same for all data distributions, but now let's look at the different possibilities.

Blockwise

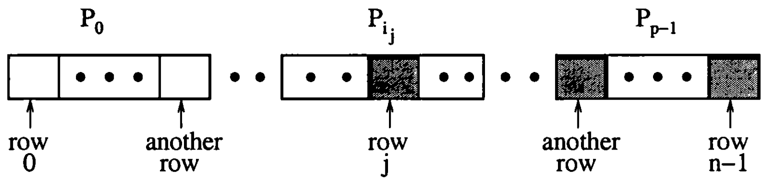


Figure 7.5: The blockwise distribution of rows.

• *deal_with_jth_row*: The active rows $(j \ldots n)$ are stored in the last few processors as shown in Figure 7.5, where the active rows are shaded. They aren't spread throughout all the processors.

Therefore only a few rows can be processed at once, and the problem with the load balance described earlier remains.

There are $n-j+1$ active rows and about n/p rows per processor, so $(n-j+1)/(n/p) = (n-j+1)p/n$ processors are used, and only this number of elements can be processed at once.

Therefore the total length of time for *deal_with_jth_row* is approximately

$$\begin{array}{rc} & \text{(number of rows)/(no. which can be processed at once)} * \\ & \text{cost to process one row + broadcast} \\ = & (n-j+1)/((n-j+1)p/n) * (n-j+1) \\ & +(1+\log p)((n+1)t_c+\tau) \\ \approx & n(n-j)/p + (\log p)(nt_c+\tau) \end{array}$$

• This process is repeated $n-1$ times until an upper-triangular matrix is produced. Therefore the total time is

$$\begin{array}{rc} & \sum_{j=1}^{n-1}(n(n-j)/p + (\log p)(nt_c+\tau)) \\ = & n/p * n(n-1)/2 + (n-1)(\log p)(nt_c+\tau) \\ \approx & n^3/2p + n\log p(nt_c+\tau) \end{array}$$

Cyclic

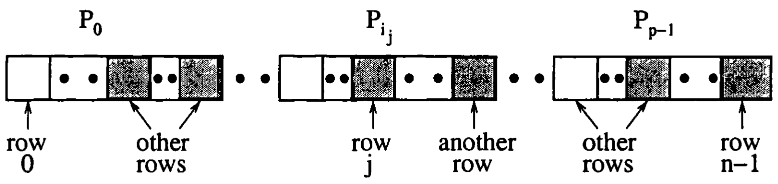


Figure 7.6: The cyclic distribution of rows

• *deal_with_jth_row*: The active rows $(j \dots n)$ are stored evenly throughout p processors as illustrated in Figure 7.6. Again, the active rows are indicated by shaded boxes. This allows p rows to be processed at once.

Therefore the total length of time for *deal_with_jth_row* is approximately

$$(n-j+1)/p * (n-j+1) + (1+\log p)((n+1)t_c+\tau)$$

• This is repeated for $j = 1 \dots n-1$. So the total time is approximately

$$\begin{array}{rc} & \sum_{j=1}^{n-1}((n-j)/p * (n-j) + (1+\log p)((n+1)t_c+\tau)) \\ \approx & \frac{1}{p}\sum_{j=1}^{n-1}(n-j)^2 + (n-1)\log p(nt_c+\tau) \\ = & \frac{1}{p} * \frac{1}{6}n(n-1)(2n-1) + (n-1)\log p(nt_c+\tau) \\ \approx & n^3/3p + n\log p(nt_c+\tau) \end{array}$$

For t_c small or n sufficiently large, the first term dominates. Therefore we can see that in these situations, the cyclic distribution is faster than the blockwise one.

Back Substitution

Due to the data dependencies involved in back substitution, it is most commonly implemented in parallel as a pipeline (see Section 6.4). Section 6.3.2 gives a cost analysis for both data distributions:

Blockwise: $n^2/p - n^2/2p^2 + n/2p + np/2 * c_1$

Cyclic: $n^2/2p + n/2 + (np - n) * c_1$

The blockwise uses less communication but doesn't overlap the calculation so well.

Redistribution

Cyclic to blockwise

A data redistribution from a cyclic to a blockwise distribution involves a lot of communication. Every processor has to send multiple values to other processors, possibly even sending every value it has. An example case is shown in Figure 7.7, in which the communication is shown as arrows, and the data elements are labelled with numbers to identify them.

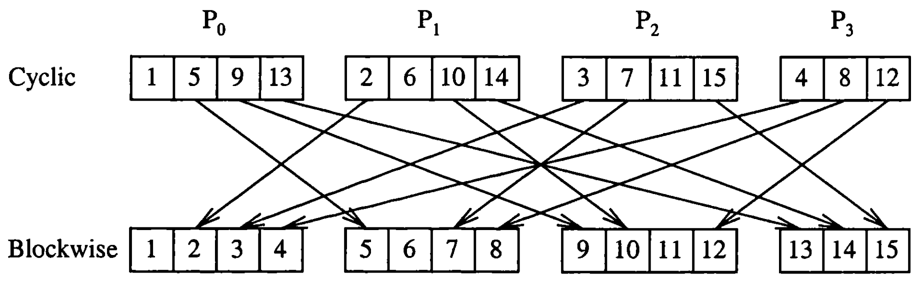


Figure 7.7: A cyclic to blockwise data redistribution

The number of elements which must be sent varies with the number of elements in each processor, n/p, and the number of processors, p. In general, each processor sends about n/p elements in total, or about $(n/p)/p = n/p^2$ elements to a specific other processor.

In a total exchange, each processor sends 1 element to each other processor. Therefore we can implement this redistribution with about n/p^2 total exchanges, each of which involves the communication of rows, which have size $\leq n + 1$.

Therefore the cost of a total exchange from Table 6.1 can be used to estimate the cost of the redistribution to be

$$\begin{array}{cc} & (p-1)\log p(n+1)t_c + p\log p\tau \\ \approx & p\log pnt_c/2 + p\log p\tau \end{array}$$

7.3.4 Cost comparison

These costs are summarised in Table 7.2. Based on them, some observations can be made about the relative merits of the different distributions. These observations are summarised in Table 7.3.

For forward elimination the cyclic implementation is faster then the blockwise. The benefit of using cyclic increases as the ratio n/p increases.

On the other hand, the blockwise implementation is often more efficient for back substitution. However this varies depending on the communication cost, t_{cn}, and on n and p. It is more efficient for the communication part but not for the computation part, and hence is better when t_{cn} is expensive relative to n, but the cyclic distribution is often more efficient when communication is cheap.

Stage	Data Distribution	Approximate Cost
Forward Elimination	Blockwise	$n^3/2p + n \log p(nt_c + \tau)$
	Cyclic	$n^3/3p + n \log p(nt_c + \tau)$
Back Substitution	Blockwise	$n^2/p - n^2/2p^2 + n/2p$ $+np/2 * t_{cn}$
	Cyclic	$n/2 + n^2/2p + (np - n) * t_{cn}$
Redistribution	Cyclic to Blockwise	$p \log pnt_c/2 + p \log p\tau$

Table 7.2: The costs of the stages of Gaussian elimination with different data distributions

Stage	Data Distribution	Advantage	Applies when
Forward Elimination	Cyclic	$n^3/6p$	Always
Back Substitution	Blockwise	$np/2 * t_{cn}$	t_{cn} high compared to n/p
	Cyclic	$\frac{n}{2p}(n - \frac{n}{p} + 1 - p)$	t_{cn} low compared to n/p

Table 7.3: A comparison of the benefits of different data distribution in Gaussian elimination

The cost of the redistribution must also be taken into consideration. This is proportional to $p \log pnt_c$ and hence tends to outweigh the advantage gained by using the blockwise distribution in the back substitution. It also tends to outweigh the advantage of using the Cyclic distribution in forward elimination unless t_c is small compared to n/p. However, in these situations, the cyclic distribution is also better for back substitution and no redistribution is needed. Therefore the redistribution is only cheap enough to be used when it's not needed and we can ignore it.

This leaves us with basically two possibilities - using the blockwise distribution throughout or the cyclic distribution throughout. The first tends to be better when communication costs, expressed in t_c and t_{cn}, are high compared with n, and the latter when they are low. Therefore, if we want to produce an optimal program, we have to know some things about the target architecture and target problem size. The derivation may branch depending on the target.

For this presentation, we choose to use the cyclic distribution throughout, as n is likely to be high.

7.3.5 Incorporating the distribution

The cyclic data distribution can be incorporated as shown in Section 3.4.4. Using these techniques, and strangling the 2nd level of parallelism, we get the following. The extra parameter, p, is the number of processors, and *scanr'* used in the implementation of *accum_scanr*1 is a form of *scanr* which returns the same number of results as the size of the initial list. It misses out the initial value. This is useful for parallelism since there is a fixed number of processors. It is implemented as $scanr'_{Cyclic}\ f\ a\ xs\ =\ init_{Cyclic}(scanr_{Cyclic}\ f\ a\ xs)$.

```
gauss :: Fractional α ⇒ Int → Cyclic(SeqFinSeq α) → Cyclic α
    → Cyclic α
```

```
gauss p a b =
    let m = map2_{Cyclic} join a b
          where join xs y = xs ++_S (list2seqfs [y])
        n = size_{Cyclic} b
        m2 = foldl_S deal_with_jth_row_{CyclicS} m (list2seqfs [1..n − 1])
        x = accum_scanr1_{CycS} solve_ith_eqn_S m2
    in
    x

deal_with_jth_row_{CyclicS} :: Fractional α ⇒ Cyclic(SeqFinSeq α) →
                                Int → Cyclic(SeqFinSeq α)
  — makes rows below row j have 0 in columns i ≤ j
deal_with_jth_row_{CyclicS} m j =
    first_j_rows ++_{Cyclic} (map_{Cyclic} put_0_in_jth_pos_S other_rows)
  where
    first_j_rows = take_{Cyclic} j m
    other_rows = drop_{Cyclic} j m
    put_0_in_jth_pos_S row = map2_S join row (m !!_{Cyclic} (j − 1))
              where
                join x y = multiplier * y + x
                multiplier = −((row !!_S (j − 1))/
                                ((m !!_{Cyclic} (j − 1)) !!_S (j − 1)))


solve_ith_eqn_S :: Fractional α ⇒ SeqFinSeq α → SeqFinSeq α → α
  — calculates the value of x_i, using the ith equation.
  — row is the ith eqn, and zs are previously calculated x values
solve_ith_eqn_S row zs =
     (foldl_S (+) (row !!_S n) (map2_S f (drop_S i row)zs))/
     (row !!_S (i − 1))
  where
    n = (size_S row) − 1
    i = n − size_S zs
    f x y = −(x * y)


accum_scanr1_{CycS} :: (α → SeqFinSeq β → β) → Cyclic α
                        → Cyclic β
accum_scanr1_{CycS} f zs = map_{Cyclic} head_S (scanr'_{Cyclic} g empty_S zs)

  where
     — g :: α → SeqFinSeq β → SeqFinSeq β
    g x ys = (f x ys) :_S ys
```

7.4 More about monads

The next step is the conversion of the program to use monads. In other programs there may be other steps which can or should be done at the non-monadic level, but not here. This conversion is quite a complicated step and is discussed in general in Sections 3.6 and 3.7 in Chapter 3. The transformation is split into several parts to make it easier, with versions of the program written after each part. There is no point in giving all the intermediate versions here, but it is interesting to note the things which needed to be done.

Introducing monads makes the program quite a lot more complicated, and there is little additional benefit from giving the code for this stage here. It can be found, if required, at the following URL: http://www.dcs.gla.ac.uk/~joy/research/thesis.

7.4.1 Basic transformation

The transformations involved a lot of uninteresting steps, which are described for a general program in Sections 3.6 and 3.7. This section lists some of these steps to show the form that the transformation took, and the modifications that are necessary for a real program. There were a few places where more interesting points had to be considered. These are given in the next section (7.4.2).

The first step involved the preliminary introduction of monads and input functions as described in Section 3.6:

- The IOPST monad was introduced to *gauss* by changing the **let** into a **do** as in Lemma 38.
- Appropriate input functions allowing the user to enter the size of the matrix and the matrix and vector were written. These are called *enter_int*, $enter\_matrix_{Cyclic}$ and $enter\_vector_{Cyclic}$ respectively.
- A *main* function was added which called these input functions and then *gauss* with the appropriate parameters.

Next the parallel system was set up and variables were introduced, as described in Section 3.7:

- The model of the parallel system was set up using *start p*.
- The types were instantiated to *Float*s.
- Global variables (p, *matrix_size* and *pid*) were added. p is the number of processors, *pid* the processors' ids, and n was renamed to *matrix_size* to avoid confusion.
- Even though p and *matrix_size* are now global, they're still passed to the input functions as parameters so that they can be accessed directly instead of through *retrieve* with the appropriate variables. This also gives the functions local to the input functions easier access to them.
- The values of these global variables were set using *get_size* and *get_pid*.
- The input data was stored in variables.
- The parameters to *gauss* were passed by reference instead of by value, by making the variables themselves and not their values the parameters (see Section 3.7.4 for more details). This was done to mimic the situation in C, where compound types such as matrices and vectors are passed by reference.
- *gauss* was also given an extra parameter, *x_v*, to return the result in, instead of just returning a value.
- *retrieve* and $retrieve_{cyclic}$ must now be used to access the values of the parameters and global variables.
- Intermediate results in *gauss* were stored in variables.

At this point monads have been introduced to the main *gauss* function but not to its subsidiary functions, so that is the next thing to do. This process is similar to the introduction of monads to the main function, but there are a few slightly more interesting things here. These are discussed in the next section (7.4.2). Some of the routine changes were:

- A new global variable *row_nos* was introduced to keep track of which rows were stored in each processor.

- The subsidiary functions (*deal_with_jth_row* and *solve_ith_eqn*) were made local and monadic.
- This meant that the places where they were used also had to change. They were used in *folds* and *scans* which expected non-monadic results, and thus needed to be modified.
- They were then written with only one APM function per line to help with the conversion to the MPI APM later (see Section 7.5).
- Operations which only took place on certain rows were previously written using *takes* and *drops* over the parallel system. They were changed to use *maps* with a function parameter whose behaviour depends on *row_nos*. More on this can be found in the next section.

At this point the subsidiary functions were monadic, but did not contain variables, so the next step was to introduce variables to them. This created a bit of difficulty because these functions were being used in *foldl* and *accum_scanr* and their parameters had to match up properly. This is discussed more in the next Section. Basic changes were:

- Their matrix and vector parameters were passed by reference.
- They had to return variables because they were being used in *foldl* and *accum_scanr*.

7.4.2 Some interesting points

The previous section demonstrates that most of the steps in converting a program to monadic form are fairly straight-forward and boring, even if detailed and fiddly. They can potentially be automated as discussed in 9.5, so that the user doesn't have to deal with them.

However there are a couple of interesting points which haven't been previously discussed. They are, however, still fairly straight-forward to apply to the program or incorporate into the program, once the basics of them have been set up.

Cyclic Variables In the previous case study all of the variables used the straightforward naive data distribution, with one value in each processor (see Section 5.7). This is the default type of variable. However this case study involves data with a particular data distribution, the cyclic one. Therefore cyclic variables are needed. These are discussed in Section 2.12.1.

It was nice to see that such variables could be introduced into a program easily, in a very similar way to the data with the naive distribution, as indicated in Section 3.7. These variables encapsulate the data distributions so that the program remains fairly clear. For example, here is a fragment of code from the program. The only parts specific to the cyclic distribution are hidden inside the relevant functions.

```
$row\_nos \leftarrow retrieve_{Cyclic}\ row\_nos\_v$
$matrix \leftarrow retrieve_{Cyclic}\ matrix\_v$
$result \leftarrow return\ (map2_{Cyclic}\ put\_0\_in\_jth\_pos_S\ matrix\ row\_nos)$
$store_{Cyclic}\ matrix\_v\ result$
```

Monadic subsidiary functions It might seem fairly straight-forward to make all the subsidiary functions monadic, but introducing variables to them changes their parameters and their types. Some of them are called by higher-order functions such as *fold*, and therefore it is important that their types should match up properly.

For example, in this case study, the matrix variable had to be threaded through the *fold*. It was passed to the subsidiary function and returned from this function in a modified form every time the function was used by the *fold*.

However, once this was done once, it could be easily generalised to a simple form which can be applied every time. Figure 7.8 shows the transformation for the *fold* example in Gaussian elimination, together with a general version for *foldl*. Note that introducing monads changes the type of the result of the fold as well as the types of its parameters. This is reflected in its use and in the type of the result of the *fold*. This is explained in Section 3.6.3.

General version

For any $f :: \alpha \to \beta \to \alpha, a :: \alpha, xs :: [\beta]$.

There exists $f' :: \mathit{IOPST}(\mathit{VarFn}\ \alpha) \to \beta \to \mathit{IOPST}(\mathit{VarFn}\ \alpha)$ related to f and a variable a_v such that.

$$\begin{array}{c} \mathit{foldl}\ f\ a\ xs :: \alpha \\ \downarrow \\ \mathit{foldl}\ f'\ (\mathit{return}\ a\_v)\ xs :: \mathit{IOPST}(\mathit{VarFn}\ \alpha) \end{array}$$

Example

$$\begin{array}{c} \mathit{foldl}_S\ \mathit{deal\_with\_jth\_row}_{\mathit{Cyclic}}\ m\ [1..n-1] :: \mathit{Cyclic}(\mathit{SFSFloat}) \\ \text{where} \\ \mathit{deal\_with\_jth\_row}_{\mathit{Cyclic}} :: \mathit{Cyclic}(\mathit{SFS\ Float}) \to \mathit{Int} \to \mathit{Cyclic}(\mathit{SFS\ Float}) \\ \downarrow \\ \mathit{foldl}_S\ \mathit{deal\_with\_jth\_row}'_{\mathit{Cyclic}}\ (\mathit{return}\ m\_v)\ [1..n-1] :: \mathit{IOPST}(\mathit{VarFn\_Cyclic}(\mathit{SFS\ Float})) \\ \text{where} \\ \mathit{deal\_with\_jth\_row}'_{\mathit{Cyclic}} :: \mathit{IOPST}(\mathit{VarFn\_Cyclic}(\mathit{SFS\ Float})) \to \mathit{Int} \\ \to \mathit{IOPST}(\mathit{VarFn\_Cyclic}(\mathit{SFS\ Float})) \end{array}$$

Figure 7.8: Transformation of a subsidiary function to be monadic. Some annotations are removed for simplicity, and *SeqFinSeq* is abbreviated to *SFS*.

takes and drops In forward elimination row operations are only performed on the last few rows. This was previously achieved in the program using *take* and *drop* to select the needed rows.

This is fine for early abstract stages in the derivation, but the monadic stages are more concrete, and it would be better to reflect the parallel nature of the operation. The same thing is happening in several processors, even if only in a few of them, and therefore it would be best if this could be expressed using a *map* over the whole matrix. The function to this *map* depends, however, on the row number, so that it can tell whether it is working on a row which should be changed or not.

This can be done using a global parameter which gives the row numbers of the rows stored in each processor, counting from 0. These *could* be calculated from the processor id and the position of the row within the processor whenever they are needed, but it greatly simplifies things to store them separately.

The transformation can then be written in a standard way as shown in Figure 7.9. Basically, this is a modification of the code for a loop that only affects elements satisfying a given property (see rules 6 and 7 in Table 3.1 in Section 3.3).

Similar rules can be written for functions which only operate on other sets of processors.

For any sequence xs and function f of appropriate types, $i :: Int^+$.

$$(take\ i\ xs) +\!\!+ (map\ f\ (drop\ i\ xs))$$
$$\downarrow$$
$$map2\ f'\ xs\ row\_nos$$
where $f'\ x\ rowno = if\ x > rowno\ then\ f\ x\ else\ x$

Figure 7.9: Transformation of take

7.5 More about MPI

The introduction of variables brings the program closer to C. However it is not much closer to MPI. This can be achieved by using the MPI APM, to be more specific, the *Cyclic* MPI APM, because the program uses a cyclic data distribution. This APM provides MPI functions which operate on cyclic distributions, but it hides the details of the cyclic implementation, keeping things as high-level as possible for just now. This also makes it easier to add in or change the data distributions at this late stage rather than earlier in the methodology, if so desired.

The Cyclic MPI APM is described in Section 2.12.2, and the necessary program transformations are described in Section 3.8, which explains how the ordinary MPI APM functions can be introduced, and Section 3.10, which describes the changes necessary for cyclic functions.

7.5.1 Communication in Gaussian elimination

As MPI is concerned with communication within a program, the first stage in introducing the MPI APM is the identification of this communication.

In Gaussian elimination communication occurs, as with most parallel programs, in the initial distribution of input values. It also occurs in forward elimination when processors must know the value of the pivotal row in order to do the row operations, and in back substitution when each processor must know the values previously calculated by other processors.

7.5.2 Input values

There are standard ways of distributing input values, as described in Section 3.10. Single values such as the size of the matrix are broadcast to all the processors. Compound values such as matrices and vectors, on the other hand, may be scattered, since each processor only needs to know a few values.

Each input function can therefore be implemented in a standard way for its type, with only slight modifications for the message which is to appear on the screen, if one is necessary.

Example For example, $enter\_matrix_{Cyclic}\ n\ p\ var\_v$ reads in the elements of an $n * n$ matrix and distributes its rows in a cyclic fashion across p processors. These rows are stored in each processor in var_v. Its implementation is as follows. It uses a subsidiary function, *enter*, which is not shown. *enter* prompts for and accepts the values of an $n*n$ matrix, and returns them in a 2-dimensional list.

$$\begin{aligned}enter\_matrix_{Cyclic} ::\ & Int \rightarrow Int \rightarrow VarFn_{Cyclic}\ (SeqFinSeq\ Float) \rightarrow \\ & IOPST\ ()\end{aligned}$$

```
enter_matrix_Cyclic n p var_v =
      do pst_putStr "Enter the elements of the matrix."
         xss ← enter n
         tmp_v ← create_var (repeat_P (list2seqfs[empty_S]))
         store_indiv 0 tmp_v (list2seqfs (map list2seqfs xss))
         mpi_scatter_Cyclic tmp_v (ARRAY FLOAT) var_v
                               (ARRAY FLOAT) 0
```

Note that the input value is stored in a temporary variable, *tmp_v* within a *single* processor, the root processor. From here it is then scattered to the whole system in such a way as to produce a Cyclic distribution.

7.5.3 Back substitution and incremental pipelining

Although back substitution comes after forward elimination in the algorithm, here it is considered first, because the communication in it is expressed in a clearer manner. It is encapsulated in the APM function *accum_scanr*1, whose implementation is considered in Chapter 6.

As discussed in that chapter, it can be implemented using pipelining. The implementation in that chapter was tailored to work for the special case when the function parameter to *accum_scanr*1 can be calculated incrementally (see Section 6.6 for details). Therefore, in order to use that implementation here, we need to show that the function *solve_ith_eqn* can be calculated incrementally. We also need to work out the appropriate incremental functions for it before we can use all the code from that chapter.

Firstly, *solve_ith_eqn* has to be modified to expect a sequence with the newest values on the end, not on the front. The only parts that need to be changed are those which identify the elements to be operated on.

```
solve_ith_eqn'_S row zs_m =
      do ...
         rev_row ← return (take_S i (reverse_S row))
         inter_vals ← return (map2_S f rev_row zs)
         ...
```

We can prove that

$$solve\_ith\_eqn'_S\ x\ ys = \mathbf{do}\ vals \leftarrow ys \quad solve\_ith\_eqn_S\ x\ (return\ (reverse\ vals))$$

as required, using the following properties (which can be proved by induction if necessary):

$$reverse(drop\ i\ xs) = take\ i\ (reverse\ xs)$$
$$map2\ f\ xs\ (reverse\ ys) = reverse\ (map2\ f\ (reverse\ xs)\ ys)$$
$$foldl\ (+)\ a\ xs = foldl\ (+)\ a\ (reverse\ xs),$$
since (+) is associative and commutative

The function then needs to be written using an intermediate summation variable, *sum_v*, as follows, so that it's in a similar form to that given in Section 6.6.

```
solve_ith_eqn'_S row zs_m =
    do sum_v ← create_var (repeat_P (0 :: Float))
       zs ← zs_m
       n ← return((size_S row) − 1)
       i ← return(n − size_S zs)
       store sum_v (row !!_S n)    — init
       for [1..size_S zs]   — the interior of this loop is g
         (λ j → do sum ← retrieve sum_v
                   store sum_v (sum + (−(zs !!_s (j − 1)) *
                      (row !!_s (n − j))))
                   — the n-j index is due to the reversal of the seq)
       sum ← retrieve sum_v
       store sum_v (sum/(row !!_S (i − 1)))    — final division - h
       sum ← retrieve sum_v
       return sum
```

The various parts of *solve_ith_eqn* can be identified as shown above: *init* which initialises *sum_v*, g which updates it incrementally, and h which finishes off its calculation. These parts can then be separated out as described in Section 6.6.

Now the APM function, *accum_scanr*1, from Chapter 6, with all its optimisations can be used. We can simply include the function from the appropriate APM at each stage. Here the collective level MPI APM code can be used.

It's also important to note that the incremental property isn't used yet in the derivation. The code from Chapter 6 could be used even if the property didn't hold until the property is used— part way through the transformations at the individual level. However, by establishing that we can use it at this stage, we can use all of the code from Chapter 6 without worrying about whether it will apply.

7.5.4 Implicit communication in forward elimination

Although much of the communication in a program is encapsulated in APM functions, as in back substitution, there may be other sources of communication. For example, calculations done in one processor may use values from another processor without explicit communication. Such instances of implicit communication need to be identified now, because when using an MPI APM, all communications should be made explicit through the use of MPI functions.

Implicit communication takes place in the Gaussian elimination study in *deal_with_jth_row* in the forward elimination phase. Here several rows have to do row operations combining their values with values from row j. Row j may not actually be in these processors, so it must be communicated to them. However in the current program the values are simply referred to without being explicitly communicated, as can be seen in the following code for the row operation. Here row j is simply referred to as *matrix* !! $(j-1)$.

```
put_0_in_jth_pos_S row rowno | rowno > j =
           map2_S join row (matrix !!_Cyclic (j − 1))
```

The communication must now be made explicit. Row j must be communicated to the processors which have rows $j+1, \ldots, n$. Since the rows are distributed cyclicly, this is usually all the processors in the system (see Figure 7.6). The easiest and quickest way to communicate a single piece of data like this to all the processors is via a broadcast, as shown in Figure 7.10.

Row j is originally stored in processor $(j-1) \bmod p$, and so a variable is set up in this processor with row j in it. This processor is also used as the root of the broadcast.

```
rowj_v ← create_var (repeat_P empty_S)
  — Store row j in processor (j − 1) mod p
rowj ← retrieve_Cyclicindiv (j − 1) matrix_v
store_indiv ((j − 1)`mod`p) rowj_v rowj
  — Now do the broadcast
mpi_bcast_simple rowj_v (ARRAY FLOAT) ((j − 1)`mod`p)
```

Figure 7.10: The code for broadcasting row j

Now that each processor has the value of row j, stored in a variable, the row operation function can be passed an extra parameter giving the value of row j:

```
put_0_in_jth_pos_S rowj row rowno | rowno > j =
        map2_S join row rowj
```

Making implicit communication explicit is discussed in general in Section 3.8.

7.6 Individual Level

Now the program looks a lot more like C+MPI, but there is a major hurdle still to be overcome. C+MPI operates on the individual level— it says what a single processor does—whereas the Haskell program works on the collective level—it says what the whole system does. We need to change the Haskell program to work on the individual level if we're going to make the transition to C as easy as possible. More information about this can be found in Sections 2.11 and 3.9.

Individual level code cannot at the moment be run and the functions are not implemented. This is because an individual level semantics has not yet clearly been developed. This is work in progress (see [O'D01]).

7.6.1 Transformation to the individual level

Changing the Gaussian elimination program to individual level is fairly straight-forward. The basic transformation can be done by applying the set of guidelines given in Section 3.9. However, the use of the cyclic data distribution causes a few changes as indicated in Section 3.10. Some fragments of the code help to demonstrate what happens.

One of the first things the program does is combine the matrix and vector. This is done by mapping a *join* function across the matrix and vector as shown in Figure 7.11.

For any matrix a, vector b, matrix variable m_v.
For p = number of processors.

```
m_v ← create_var_Cyclic(repeat_Cyclic p empty_S)
store_Cyclic m_v (map2_Cyclic join a b)
                    ↓
m_v ← create_var_Cyclic(repeat_S empty_S)
store_Cyclic m_v (map2_S join a b)
```

Figure 7.11: Transformation of the matrix and vector join code to the individual level

At the individual level, it is important to remember that there are multiple rows in each processor because of the data distribution. Therefore the map_{Cyclic} can't translate directly into a function application as it did in the *maptri* case study (Section 5.9) which didn't use data distributions. Instead a sequential *map* over the values stored in the processor is used.

The transformation is made simpler by individual level cyclic functions such as $create\_var_{Cyclic}$ (see Section 2.12.3 for a full set of these functions). These manipulate the multiple values for the user, so that he doesn't have to do so explicitly himself.

This might seem a bit tricky and rather detailed. However, in practice, it's much easier, because there are sets of transformation rules (Sections 3.9 and 3.10) which encapsulate these details. In fact, this is an example of a transformation which could be automated in future.

This example involved only computation, but communication is similar. MPI collective-level functions transform into set MPI individual-level functions. Collective ones, such as broadcast, transform into functions with the same name, and so no change to this code is needed. This can be seen in the code for broadcasting row j in forward elimination, which is given in Figure 7.12.

For the specific variable *rowj_v*, associated integer j and p the number of processors.

$$\begin{array}{c} mpi\_bcast\_simple\ rowj\_v\ (ARRAY\ FLOAT)\ ((j-1)`mod`p) \\ \downarrow \\ mpi\_bcast\_simple\ rowj\_v\ (ARRAY\ FLOAT)\ ((j-1)`mod`p) \end{array}$$

Figure 7.12: Transformation of the broadcast of row j to the individual level

After the applying these transformations, the code looks a bit different. It can be found at http://www.dcs.gla.ac.uk/~joy/research/thesis.

7.6.2 Pipelining

One reason for dealing with the individual level is that it brings the program closer to C+MPI, and therefore makes the final transformation to C+MPI easier. However this is not the only reason. There are also optimisations which can be done more easily at the individual level, as explained in Section 3.9.2.

In the current case study some such optimisations can be applied to the back substitution phase of the program. It uses pipelining, a technique described in detail in Chapter 6, and which can be optimised at the individual level through several steps.

First of all the values received by a processor, i.e., the previously calculated values, are passed on to the next processor as soon as they are received, overlapping the communication time of one processor with that of another. This is described in Section 6.5. Then the subsidiary function that calculates one of the result values is split up into its incremental parts. Section 7.5.3 shows how this is done for the subsidiary function in back substitution, *solve_ith_eqn*. These parts are then moved around so that their calculation time overlaps the communication time, as described in Section 6.6.

Although these optimisations could be done at the collective level, it is easier to describe details such as one processor passing on a value as soon as it's received than to calculate and express the resultant pattern of communication for the whole parallel system.

It is also useful to introduce other optimisations, not involving communication, at this point. One such optimisation involves joining the matrix, a, and vector, b, to form a combined matrix m. It would be more efficient to store the input values directly into the appropriate places in m rather than storing them somewhere else and then copying their values. This optimisation could have been performed earlier, but it's useful to be able to do it now without having to change the previous code.

This optimisation focuses on the input functions, $enter\_matrix_{Cyclic}$ and $enter\_vector_{Cyclic}$, although there are also minor modifications to *gauss* so that it takes the combined matrix as a parameter instead of a separate matrix and vector.

The vector elements are now attached to the ends of rows of the matrix instead of being stored in a separate vector:

```
xs ← enter_vector 1
  — for each row, store the appropriate vector value
for [0..matrix_size − 1]
    (λ i →
          do input ← retrieve input_v
             store input_v (replace_S input i
                ((input !!_S i) ++_S (cons_S (xs !!_S i) empty_S))))
```

The size and displacement arrays which are used to indicate the indices of the rows for scattering must also be changed to reflect this change in the length of the rows.

7.6.3 Profiling

It would be useful to profile the program before and after parallel optimisations, such as pipelining, have been introduced. This was done using GpH in Section 5.5 when static load balancing was introduced to the map-triangle case study.

However, that was earlier in the derivation when the transformation was more abstract. At the present stage, the parallel optimisations deal with details of communication and message-passing, and rely heavily on location-awareness as messages pass between specified processors. GpH is not location-aware, and therefore it is usually not possible to observe the effect of the optimisations on GpH programs.

In addition, GpH is a collective level language, observing the whole system at once. A program at the individual level must be converted to the collective level if it is to be run and profiled using GpH. This can be done either by producing an implementation of the individual level, as described in Section 2.11.1, or by rewriting it by hand. The former is outside the scope of this thesis, and the latter involves much effort on the part of the programmer. It also reintroduces the difficulties discussed above with performing these communication optimisations at the collective level.

7.6.4 Removal of cyclic functions and more optimisations

Before the final transition can be made, there are more details that need to be sorted out. In particular, the cyclic functions need to be replaced with ordinary functions since C+MPI doesn't provide cyclic variants of its operations. This can be done using a set of transformation rules, which are given in Section 3.10.

In the Gaussian elimination case study these rules don't cause any problems. They are only used to transform variable manipulation functions and a few communication functions which are only in the input functions. The other communications already use non-cyclic functions.

This paves the way for more optimisations on the program. In particular the back substitution phase can be optimised further as described in Section 6.8. This optimisation is specific to the cyclic distribution, yet it is easier to do it once the cyclic communications have been removed, because then the individual communications become visible and can be manipulated.

7.6.5 Details to get close to C+MPI

There are other more detailed transformations which need to be done to make the program more like C+MPI. Any remaining *maps* need to be replaced with loops, and there may be details of arrays or types which need to be dealt with. Such transformations are discussed in general in Section 3.9.2.

maps: Let us first look at the *map* function. This can commonly be implemented in C using a `for` loop, as shown in Figure 3.4 and discussed in Section 3.9.2. This case study uses a related function, *map*2, in forward elimination, in the function *deal_with_jth_row*, as follows:

```
result ← return (map2_S (put_0_in_jth_pos_S rowj j) matrix row_nos)
store matrix_v result
```

This transforms in a similar way to *map* to give the following:

```
for [0..no_in_proc]
(λ i →
      do new_row ← return (put_0_in_jth_pos_S rowj j (matrix !! i)
                                   (row_nos !! i))
         matrix ← retrieve matrix_v
         store matrix_v (replace_S matrix i new_row))
```

Recursive functions: The program also contains recursive functions, *enter* and *enter_line*, used to input the values of the matrix and of one row respectively. In general, recursive local computation functions may appear elsewhere in the program. These can be implemented in C using recursion, although it is often more efficient to use `for` loops. They can therefore be transformed in a similar way to *map*.

For example, *enter* is transformed as follows:

```
enter :: Int → IOPST[[Float]]
  — Reads in m rows each of length n
enter m | m == 0 = return []
        | m/ = 0 = do xs ← enter_line (n − m + 1) 1
                      xss ← enter (m − 1)
                      return (xs : xss)
...
xss ← enter n
store input_v (list2seqfs (map list2seqfs xss))
```

becomes

```
...
for [1..n]
(λ i → do xs ← enter_line i 1
          input ← retrieve input_v
          store input_v (replace input (i − 1) (list2seqfs xs)))
```

The code for the *enter* function is inlined into the main code, so that the variable, *input_v*, can be modified each time a new row is entered instead of once at the end.

Communicating arrays: So far in Haskell arrays have been communicated in collective communication functions using a special type, *ARRAY*. For example, row j is broadcast using:

mpi_bcast_simple rowj_v (ARRAY FLOAT) ((j − 1)'mod'p)

However, MPI doesn't provide this special type. Instead arrays are communicated by specifying their starting address, the number of bytes to transfer, and the element type. For example, the following code also broadcasts row j:

```
MPI_Bcast(rowj, matrixsize+1, MPI_FLOAT, mod(j-1,p), MPI_COMM_WORLD);
```

This gives the starting address of the row, `rowj`, its length, `matrixsize+1`, and its element type, `MPI_FLOAT`.

This can be reflected in Haskell by introducing a new version of *mpi_bcast* which takes similar parameters to the C version as follows:

$mpi\_bcast'\ rowj\_v\ (matrix\_size + 1)\ FLOAT\ ((j-1)`mod`p)$.

Matrices: Nested sequences, such as *SeqFinSeq* (*SeqFinSeq Float*), are used to implement matrices in the Haskell programs. They should translate into C with no difficulty. However, in practice, 2-dimensional arrays are hard to deal with in C, especially when they are being passed between functions as parameters. Index arithmetic may cause problems, and my version of C wouldn't let index arithmetic be performed at all on such arrays passed as parameters. This can be dealt with by representing a matrix as a 1-dimensional array with the rows stored one after the other, and by doing index arithmetic on it accordingly. This is not ideal but serves the purposes of this case study. The necessary transformations of the program can be encapsulated in transformation rules, as shown in Figure 7.13.

In this figure, the annotations on the indexing function, !!, are removed so that it's easier to see what's going on. The rules also use the function *slice* to extract one row from a matrix, *a*. *slice* works as follows:

$$\begin{aligned} &slice\ i\ j\ [x_1, \ldots, x_n] = [x_{max\ i\ 1}, \ldots, x_{min\ j\ n}] \\ &= \{usually\}\ [x_i, \ldots, x_j] \end{aligned}$$

For any matrix a, positive integers i, j.
A single element of a matrix, a, transforms as follows:

$$(a!!i)!!j$$
$$\downarrow$$
$$a!!(i * \text{row length} + j)$$

A row of a matrix, a, transforms as follows:

$$a!!i$$
$$\downarrow$$
$$slice_S(i * \text{row length})\ ((i+1) * \text{row length})a$$

Figure 7.13: Transformation rules when representing a matrix as a 1-dimensional array

The whole program after all these transformations can be found on the internet at URL http://www.dcs.gla.ac.uk/~joy/research/thesis.

7.7 C+MPI

The previous stage is very close to C+MPI code, and can be translated using the transformation rules given in Section 3.11. As before, the resultant program can be found on the internet at http://www.dcs.gla.ac.uk/~joy/research/thesis.

Scattering the cyclic data requires the use of a temporary variable that holds a version of the matrix with its rows in the order they would be when distributed cyclicly. This isn't hard, but it's also not very elegant.

I originally thought of using a derived datatype for the scattering. MPI derived datatypes allow one to communicate regular sections of arrays made up of blocks of elements of a set length separated by a set number of other elements. More information about how they do this can be found in the MPI manual [MPI97].

A derived datatype can be set up to represent a row of the matrix. This would make the communication operations involving the rows (their initial scattering and the broadcast of row j) easier. However, this doesn't work because `MPI_Scatter` assumes that items to be sent to processor $i + 1$ all lie after those to be sent to processor i. On the contrary, to obtain a Cyclic distribution some rows to be sent to later processors occur before some to be sent to early ones. Therefore derived datatypes were not used for this, although they could be used in other situations.

7.7.1 Timings

This code can be compared with code written by hand without certain of the optimisations, to see the effect that different choices had on the performance.

Cyclic versions - with and without pipelining

The derived C+MPI program was compared with a similar program, also using the Cyclic data distribution, but without the pipelining optimisations, to see their effect. Timings were taken on a Beowulf cluster, and the cost of the calculations was increased by introducing busy work before arithmetic operations in back substitution. This was necessary to allow the times to be observed, but alters the communication/calculation cost ratio significantly. This is considered below.

Timings were taken for matrix sizes, $n = 8, 16, 24$ and 32 as well as intermediate values 10 and 20, to ensure that patterns in the times weren't caused by the n being a multiple of p. The times are shown in Figure 7.14.

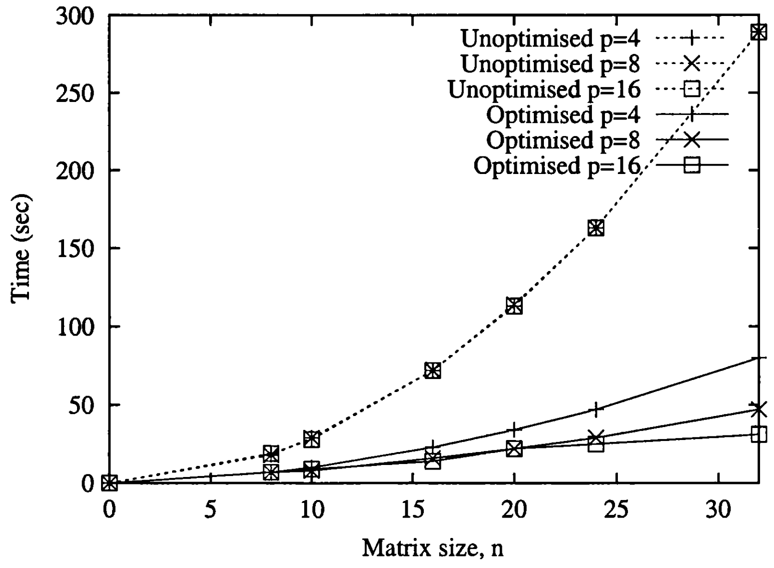


Figure 7.14: Gaussian Elimination timings with and without pipelining optimisations

The unoptimised version takes the same amount of time for a fixed matrix size regardless of the number of processors used. This is very inefficient, and shows no speedup. The optimised version works much faster and shows an improvement with an increased number of processors, if $n > p$.

Decreased communication/calculation cost ratio

The long calculation time in the previous case reduces the communication/calculation ratio, simulating a parallel machine with much faster communication. The calculation time can be changed to simulate other machines. Figure 7.15 gives some times for the expensive calculations of the previous section, and also for calculations many times faster, but still slower than the ordinary arithmetic calculations.

The same difference between the optimised and unoptimised versions of the program can be seen.

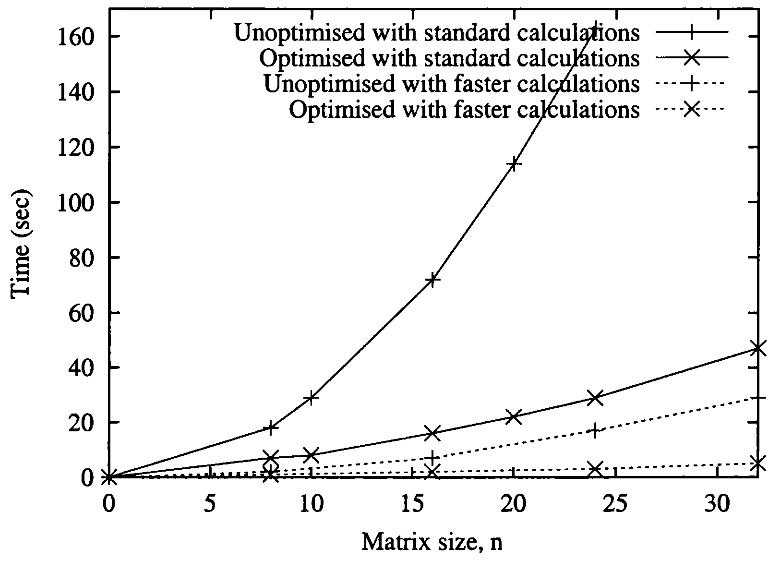


Figure 7.15: Gaussian elimination timings for p=8 with different calculation costs

7.8 Summary

This chapter has described a longer case study carried out using the APM methodology. This case study stretched the methodology in many ways. Simply by being larger and more complex than the previous study, it forced me to make the lemmas and transformation rules more general, and made me look at areas which I hadn't examined before. In particular, this study involved data distributions and detailed optimisations of an imperative nature. Part of the work on this study is given separately in Chapter 6, which discusses a particular form of pipelining used within Gaussian elimination.

The study took quite a long time to do, but most of this time was spent dealing with the new issues which arose and refining the methodology accordingly. Some of these changes are outlined in this chapter, although others are simply incorporated into the information given in previous chapters, for the sake of clear presentation. Doing the study was interesting, and brought up many unexpected points and problems. This was useful because it allowed the methodology to be refined.

Chapter 8

Related Work

This chapter has two main aims. Firstly, it gives a critical review of the main approaches to several aspects of parallel programming that are closely connected to this thesis. Secondly, it investigates how these approaches are related to the work in the thesis.

The chapter is organised according to various issues in parallel programming, but there are occasional interludes to discuss a parallel system or language in more detail.

8.1 Decision making in parallel programming

Parallel programming involves making many decisions about a variety of aspects of the program. For example, tasks must be assigned to processors, and data must be allocated and communicated. These decisions may be made by the programmer or automatically. This section discusses several parallel programming languages and systems and the ways in which they support such decisions.

8.1.1 Parallelising compilers

At one end of the spectrum of programming languages lie those which are implicitly parallel. A program in these looks basically the same as a sequential program. It is fed into a compiler which makes all the decisions about the parallelism, and produces a program which can then be run on a parallel computer.

This is especially suited to functional programming languages, because they don't enforce an order of execution or any unnecessary data dependencies. Different parts of a program can be executed in parallel without producing incorrect results [Bac78]. For more information about parallel functional programming see, for example, [Ham94].

In the 1980's there was research into novel parallel architectures such as GRIP [PJCSH87] which were specifically designed to run such languages. However these were rapidly overtaken by developments in stock hardware. Now standard parallel machines are used, because these machines improve at such a rate that they render investment in specialised hardware unproductive. This, however, raises a problem with small grain parallelism. If one executes all potentially parallel parts of a program in parallel, this generates an incredibly large quantity of very small grain parallelism. Standard machines don't cope well with this because it requires more time in communication than is gained in computation time. One must choose which operations are worth carrying out in parallel and which not. This research has therefore to a large extent given way to research based on user annotations to guide the compiler [PJL92], and hence to languages such as GpH (described in Section 2.3.6. However implicitly parallel functional languages still exist, e.g., pH [NAH$^+$95].

A similar idea can be applied to logic languages since the resolution of a logic query involves many activities that can be performed in parallel. Examples of such languages can be found in [Tal94].

Parallelising compilers, however, still exist. For example, there are parallelising Fortran compilers [BGA90], which parallelise parts of Fortran 90 code, such as loops. The interest in such compilers has arisen partly due to the large amount of legacy Fortran code in existence. Such compilers use a variety of program transformations. For example, [AK87] discusses how data dependencies can be determined in a Fortran program in order to figure out which parts of the program can be executed in parallel. This is also an issue in the methodology described in this thesis, as data dependencies play a large part in deciding on parallelism (Section 4.3). This could be done using similar techniques.

ZPL [LS93] is another implicitly parallel language which is still in use. It is focused on data-parallel programs, and is based on arrays. As with other implicitly parallel languages, the compiler makes the decisions about the parallelism.

Such programs are easy for the programmer to write, but they often have the disadvantage of sub-optimal performance. Parallelising compilers cannot make intuitive decisions, so-called "Eureka" steps. They aren't as good at parallelising as humans, and may not make such good decisions. They also cannot, as yet, apply some of the more formal methods, such as the use of cost models (see Sections 4.2.2 and 8.1.4), which require algebraic analysis of costs. However much work has and is being done on such compilers and they have undergone much improvement.

Nevertheless parallelising compilers fail to take advantage of many features of the target machine, and still fail to produce code as fast as that hand-written in explicitly parallel languages.

Compiler directives

As previously mentioned, parallelising compilers sometimes have problems in making basic decisions, for example about what should be in parallel and what not. One way to handle this is to let the programmer supply hints or directives to the compiler. The program is still implicitly parallel because these are only hints: the compiler doesn't have to obey them. However they can prove very useful in helping the compiler to make more effective decisions. Although the burden for the decisions is still on the compiler's shoulders, the programmer can choose to share it.

Examples of languages with such hints include GpH, which is discussed in more detail below and Caliban [CHK$^+$93, KT99]. The latter uses annotations to allow the programmer to specify data partitioning and placement.

GpH

In GpH (see Section 2.3.6) parallelism is identified by the programmer, but not schedule, allocation or load balancing strategy. Some details can, however, be controlled through the use of run-time system (RTS) options. The programmer must therefore make decisions both when writing the program, and when setting the RTS options. Usually these are made on the basis of simulations, profiling and sample timings using visualisation tools. An example of how such profiling can improve the decisions made about a program can be found in [LTB01]. The profiles can be used to identify bad load balances, bottle necks, and other problems in the program's execution.

8.1.2 Explicit and semi-explicit languages

It is more common for parallel languages to be explicit about some of the parallel details. The programmer has to specify these himself, while the remainder are still decided by the compiler. Skillicorn and Talia [ST98] survey many of these languages, dividing them up by the details supplied by the programmer. There are many such languages, and here is not the place to discuss them all.

Instead this section presents a representative sample, and discusses how the programmer makes his decisions.

At one end of the spectrum lie languages which are only explicit about what's executed in parallel and what not, not about any other details such as where these things are executed or what communication is needed. These include languages such as HPF (described and discussed below), Strand [FK94] and NESL [Ble95, Ble96].

Somewhat more explicit is BSP [SHM97, Val90]. This is a programming model with an associated cost model, which estimates the costs of a BSP program. In the BSP model the program is structured as a set of computation steps separated by collective communication steps in which all the communication takes place, although this structure has, in some cases, been relaxed. This is an instance of the restricted SPMD programming model described in Section 8.2.3.

There are various BSP languages. They provide a fixed library of BSP functions, such as BSPLib [HMS+97]. Decisions about programs are usually made by hand using the associated cost model, but some are made by the compiler, such as the placement of data.

Eden [BLOMPM96, BL97], a parallel functional language based on Haskell, is also semi-explicit. Processes are explicitly created and connected, and communication and synchronisation is also given explicitly, but it remains abstract about the placement of these processes.

At the other end of the spectrum lie languages which are explicit about everything. These include the message-passing libraries, MPI (described in Section 2.10.1 and discussed further on the following page) and PVM [Sun90]. These are widely used, and have a degree of portability. However the programmer has to make all the decisions herself. Programs in these languages can achieve a high degree of efficiency because they can be hand-crafted, and can make use of details of the parallel computer.

There is a connection between the degree to which a language is explicit, the difficulty in programming in that language and the speed achievable. Fully explicit languages allow programmers to make full use of parallel optimisations by hand-tuning their programs. This can produce very efficient programs, but requires a high degree of programmer effort and can be difficult. Implicit languages, on the other hand, hide these details from the programmer, making programming easier, but disallowing hand-crafted optimisations. Another issue is portability, as many of the optimisations are specific to a particular target, and including them means that the program will run fast on that target, but possibly slowly on other ones.

The approach in this thesis uses a mixture of the implicit and explicit approaches, employing the technique of incremental programming, described in Section 8.2. At the start of a derivation the program is implicit, but as the derivation proceeds the parallelism becomes more and more explicit. This gives the methodology the benefits of implicit programming in the early stages when optimisations are not necessary, while allowing the benefits of explicit languages in the final program. In particular, explicit languages allow us to give a lot of parallel details and use detailed optimisations. This thesis targets one such language, C+MPI, for this reason, although the methods described are applicable to the whole range of parallel languages. The approach is also designed to help alleviate the problems with explicitly parallel programming such as the difficulty of the programming and the lack of portability.

HPF

It is worth mentioning the semi-explicit language, HPF(High Performance Fortran) [MH95, MC97] in more detail because it provides an example of the way in which such languages deal with data distributions, one aspect dealt with in this thesis.

HPF is an informal standard for extensions to Fortran 90 [BGA90]. These extensions include compiler directives for expressing data distribution and a few constructs for parallel assignment loops. Directives start with `!HPF$` to distinguish them from the code.

For example, consider the code:

```
!HPF$ DISTRIBUTE a (BLOCK, BLOCK) ONTO procs.
```

This specifies that both dimensions of the matrix a should be distributed in a blockwise fashion over the corresponding dimension of the set of processors, *procs*. Each dimension can be specified separately, and distribution formats include * as well as the more standard ones. The * indicates that the corresponding dimension is collapsed, i.e., each entire "row" is considered as a single indivisible element, to be stored in a single processor.

Alternatively the programmer can align pieces of data with other pieces of data, so that corresponding elements are stored in the same processors. For example,

```
!HPF$ ALIGN b(:,:)  WITH a(:,:)
```

means that each element of b must be mapped to the same processor as the corresponding element of a. Again extra array dimensions can be collapsed using *.

It is the compiler's job to generate a data distribution for each piece of data so that it satisfies all of these constraints.

MPI

MPI (see Section 2.10.1), in contrast to many of the languages discussed so far, is explicit about most areas of parallelism. This allows very fast programs to be written through coding the communications and data placement by hand. Therefore it has gained a fair degree of popularity, both as a programming language, and as the target of derivation and compilation systems. Another key factor is its portability. MPI is implemented on a range of tightly-coupled, massively-parallel processing machines and networks of workstations. Although this doesn't cover all parallel systems, these are the key systems in use today.

Partly for these reasons, C+MPI was used as the target for the methodology in this thesis. In principle the methodology can target a wide range of parallel systems and languages, but MPI provides us with a widely-used and efficient representative target.

However MPI also has some disadvantages. In particular, because it is so explicit about the parallelism, the programmer is responsible for many of the parallel decisions. The compiler provides little support for this. These decisions are also intertwined so it is hard to introduce them one at a time. Therefore many programmers try to write the final code straight away, making all of the decisions at once, using methods such as those given in Section 8.1.4. These disadvantages can, however, be overcome by altering the normal method of writing programs. Such a technique is used in this thesis.

8.1.3 Parallel combinators

Parallel combinators are an idea that arose from work in functional programming on expressing programs through a set of combining forms or functions [Bac78]. The parallelism is encapsulated in a set of higher-order functions, such as *map* and *scan*, which capture a specific communication pattern. This set may or may not be fixed.

This idea was picked up and modified for *algorithmic skeletons* [Col89, DFH$^+$93], which provide the user with a fixed set of higher-order functions or program "templates" for expressing the parallelism. They were originally designed as a portable library for imperative languages, but were quickly integrated into functional languages. Each skeleton can be implemented for multiple target machines, one implementation being given for each target. A skeletal compiler, such as P^3L [BDO+95] translates the programmer's calls to skeletons into their implementations on the appropriate architecture.

Skeletons have the advantage that their implementations can be made very efficient by writing them by hand. Considerable effort can be invested in their development because there's only a limited number of such implementations which need to be written. But the programmer herself still doesn't have to deal with such details, and is thereby enabled to write her programs at a high level of abstraction.

There has been a lot of work in the skeletons community and on parallel combinators in a more general sense. In fact, some of the work on skeletons broadens the definition of a skeleton. In particular, people have realised that single implementations for each skeleton/machine pair aren't always a good idea, because an efficient implementation in one instance might not be efficient in another. Sometimes it's better to allow a choice of implementations.

P^3L [BDO+95] does this by parametrising the implementations with the costs of the low-level operations on the target, and finds good values of these parameters using cost formulae.

Bratvold [Bra92] looks at the choice between parallel and sequential implementation of a combinator, and considers how to allocate tasks so that a good load balance is achieved. These decisions are made by hand, although they could in theory be done automatically. Performance models estimate the performance of different alternatives.

Performance models can also be used for other decisions. Darlington [DGT93, DFH+93] describes how they can be associated with each skeleton/machine pair to help one choose between different implementations of skeletons.

As well as standard performance models, specialised ones for skeletons have therefore also been developed [HC00].

FAN [GP99] extends work on skeletons to allow higher-level decisions. It automatically applies transformation rules to give a set of alternatives with cost estimates, on the basis of which choices can be made. Choices are made by hand, but the estimates are generated automatically.

8.1.4 Decision making methods used

As previously mentioned there are many decisions to be made in parallel programming: the placement of the data, the scheduling of the tasks, the choice of communications, and many others. The languages and systems described above show that there are many techniques and methods for making such decisions. These vary widely in formality and style, and in the ways in which they are applied. Programmers commonly use informal techniques, sometimes even just relying on programmer intuition and experience. Compilers, on the other hand, use precisely-defined algorithms. Sometimes such algorithms are too complex and detailed to be applied by hand, but this is not always the case.

General guidelines

Programmers often use guidelines to help them make decisions. Such guidelines come in many forms. Although some are merely intuitive, others are taught in courses on parallel programming or given in introductory textbooks. For example, the textbook [Akl89] describes basic techniques in designing parallel algorithms. Others, such as [WA99], give strategies for parallel programming, such as methods of partitioning data or pipelining, illustrating them with examples of when and how they can be applied.

Other, more detailed guidelines also exist. For example, much work has been invested in determining the best ways to parallelise particular problems. Textbooks, such as those mentioned above, give standard algorithms with appropriate data distributions for some common problems. In particular, [Rob90] focuses on Gaussian Elimination and shows how it can be coded for a variety of parallel systems. These techniques can also be applied, with some imagination, to other algorithms.

Other research has focussed on classes of problems. This has often been in the context of particular systems, noticeably skeletons (e.g., divide-and-conquer [GL95]), but often the insights gained can also be applied to other situations.

Profiling and Simulations

Several languages and systems, such as GpH, have ***profilers*** which can pass run-time information to the system or the programmer about the execution of a program. This includes the time the execution takes, but often also other information, such as the load on the processors and the status of the tasks. Such profilers may be specialised for particular versions of a language or for a particular system. For example, in the Digital Parallel Software Environment, HPF can be profiled using the `ppref` utility [HPF97].

Such information can be used to determine where the ***hotspots***, in a program are. These are the places in the program where the parallel behaviour is particularly bad. These parts can then be further examined and improved. The profiler might also help to identify the type of problem, for example, many idle processors are symptomatic of a bad load balance or lack of parallelism. An example of this is given in [LTB01] for GpH.

Even without detailed profile information, run-times can be used to compare alternative implementations to determine which is better.

However profiling requires an executable, final-version program to be written, and run on the target architecture. Even worse, when comparing implementations, more than one program must be written. In many situations, the time to write a final program is high, as may be the execution time. This can be a costly exercise. Therefore, if possible, if is useful to time just one module or part of the program instead of the whole.

An alternative approach is to use a *simulator*, which simulates the program's execution rather than actually running it on a parallel machine. This is particularly useful if the target machine is not available during the development process. Some simulators therefore model specific machines. [HGL+93] gives one of many examples—a simulation of the data diffusion machine. Other simulators, such as GpH's GranSim [Loi96], are designed to model a variety of machines. These have the additional advantage of letting the programmer easily see the program's operation on a range of machines. It is also sometimes possible to simulate the operation of a program which is not complete—perhaps without all of its parallel details decided. This is particularly useful because one doesn't have to write the entire program in its final form, a time consuming task especially if several versions are to be tested.

A simulator doesn't give such accurate timings, but it is simpler and quicker. If the program isn't in its final form, the simulator may have to make some assumptions about it. These are kept as reasonable as possible, but in many cases they cannot be made usefully. The different decisions in a program all interact with each other, and sometimes all of them have to be made before sensible timings can be obtained.

Prewritten packages

Many programmers make use of packages written and optimised by others, which has the advantage that the programmer himself need not make the decisions. Skeletons take this approach, by providing standard implementations of a fixed set of combinators. Collective MPI library functions can also be viewed in this way, although on a much lower level of abstraction. These functions are implemented in an efficient way which is hidden from the user. The programmer need not worry about what individual communications are needed to obtain the collective communication.

There are also many higher-level prewritten packages or libraries of functions, which implement whole algorithms efficiently, especially for certain areas of programming, such as numerical analysis. An example is ScaLAPACK [BCC$^+$97] which is a library of linear algebra routines for distributed memory MIMD machines and networks of workstations. It supports both MPI and PVM.

This method frequently produces very good code for the provided functions. However, when dealing with a situation outside the scope of the packages, it is of little help. In some cases, it can also be restrictive, discouraging or forbidding the programmer from expressing parallelism in other ways. In other cases, such as in MPI, the functions provided are still fairly low-level, and their use still involves much decision-making.

Cost models

Cost models, described in Section 4.2.2, are frequently used to make decisions because they provide a quantitative analysis. They are applied, for example, in work on skeletons [DGT93, GP99], and in the derivation approach, TwoL (see Section 8.2.3).

Cost models are often used to model the cost of a whole program, different alternatives for which can then be compared. However it is also possible to produce costs for each part of the program and then work out the best combination of these with an algorithm. This method is used, for example, in [To95]. It serves to simplify the cost calculations involved.

In some cases it is possible to parametrise the model with descriptions of the decision which is to be made. The costs can then be manipulated to determine what the best values of the parameters would be. For example, TwoL [RR96] uses parameters to describe the data distribution. This thesis makes use of a similar idea (see Chapter 4).

There are a variety of cost models available. Skillicorn in [Ski99] explains some of the difficulties involved and some of the approaches taken in these models.

Many people have made up cost models suited to their particular needs—for a particular machine or class of problems. There are also more general cost models available which apply to a range of machines and languages. However they do not have the accuracy of a model that deals with the specific concrete details of a particular machine. These general models abstract away from these details by parametrising the cost measures. For example, BSP, described in Section 8.1.2, uses a general cost model for SPMD programs. Parameters represent the number of processors, the cost of barrier synchronisation and the cost per word of delivering a message. LogP [CKP$^+$93] is another general model, related to BSP, but it can be used for less structured programs.

Since machine-specific models can only be used for particular machines, late in the methodology, they get in the way of using the derivation for multiple targets. However, some decisions are very tied to the target machine, and different decisions need to be made for different targets. In such cases these models are useful.

However, other decisions aren't so dependent on the target, and for these, it's better to use a more general cost model. In principle any cost model could be used in the methodology described in this thesis, and different cost models could be used in different places. O'Donnell and Rünger [ORR01] have proposed formalising this as a set of different cost models for the methodology.

For illustration purposes, this thesis uses the cost model used by TwoL (Section 8.2.3). It is described in this thesis in Section 6.3.1. This is a simplification of the LogP cost model which is designed to give simple, general but fairly accurate cost approximations for use in making decisions about programs.

In general, cost models allow us to reason about the program clearly and mathematically, using established techniques. However they can produce complicated equations which are difficult to manipulate.

Solving constraints

As well as being used directly to compare the costs of different alternatives, cost models can be used in more oblique ways. Section 4.2.2 describes how a set of constraints on the program can be specified and then manipulated (perhaps using cost models) to determine good values for parameters.

For example, in HPF (see Section 8.1.2) the programmer specifies a set of constraints on the placement of data. Certain pieces of data have to be placed in the same processors as certain other pieces, and some of the distributions can also be specified. The algorithm produces a set of data placements which solves this set of constraints.

There may also be constraints on the functions available from a library. For example, a blockwise distribution might be more efficient for a procedure in the context of a particular program, but the library may only provide a cyclic implementation. In such cases, the programmer has a choice between using sub-optimal distributions, or taking the extra effort of re-implementing library functions with different distributions.

An example in this thesis of making decisions by satisfying a set of constraints is given in Section 5.4. It involves static load balancing. A good load balance often (although not always) improves a program's performance. Rather than examining the performance directly, it is therefore possible to speed up the program by considering only the load balance. This could be expressed in equations which can be manipulated.

Combination

In most informal systems, a mixture of decision-making methods is used. Programmers commonly use a mixture of intuition, guidelines and profiling when programming directly in the target language. Some of the more formal approaches also allow a choice between decision-making methods, for example, TwoL gives a choice between automatic decision-making and user choice.

Different methods are suited to different circumstances. Sometimes a decision is easy to make, especially for an experienced programmer, and formal approaches are not necessary. For example, one data distribution may place all the data which is used together in the same processor and so requires no communication while another requires a lot of data to be communicated constantly. The difference in the cost of these two choices is clear without formal calculation. However other decisions are harder to make, and mathematical methods, both those based on cost models and key properties, may be able to help in such cases.

This thesis attempts to maintain a balance by allowing different methods to be used in different situations. A cost model can be used when its formalism and increased accuracy are needed. However more ad-hoc methods can also be used when they can give good results or when non-optimal programs are acceptable. This is discussed in Chapter 4.

8.2 Incremental programming

The previous section discussed methods for making decisions during parallel programming, but separate from the methods used, there is another question to be addressed. Should all of the decisions be incorporated into the program at once, or one at a time?

It is common practice to write programs directly in their final form. These programs can be modified if profiling shows problems, but in many programming languages it is hard to isolate separate decisions—they all affect multiple parts of the code. In essence the programmer deals with multiple decisions at the same time.

A more effective approach is *incremental programming*, in which the decisions are made and introduced into the program one at a time. The program goes through a series of well-defined versions before it arrives at its final form, and the transformations between them are also well-defined. The term "incremental programming" is also sometimes used more generally to mean that programs are modified incrementally, but here I focus on the more specific definition.

Incremental programming is especially useful for parallel programming because of the large number of decisions which must be made.

8.2.1 Compilation by transformation

One of the most common uses of incremental programming is in source-to-source transforming compilers. Compilers commonly have fixed stages or phases which transform a program's representation. Examples of such stages are the lexical analyser, the intermediate code generator and the code optimiser [ASU86].

Each stage has its purpose, though probably the one most closely related to incremental decision-making is the optimisation phase, since decisions are usually made in order to optimise a program. This phase may or may not be incremental, with one optimisation applied after another, and may not even be included in a compiler.

Several compilers use an idea called *compilation by transformation.* In this, the compilation process is expressed as a sequence of correctness-preserving transformations, each of which modifies the program in a way that makes it more efficient or brings it closer to the target. This is the idea used in GHC [PJS94], and the parallel functional language Eden, as discussed in [PPRS00], and is closely related to the transformation process often used in incremental parallel approaches.

8.2.2 Incremental parallel approaches

Incremental parallel programming is often used informally, without a fixed structure or stages. The programmer writes a program, analyses its performance, and then rewrites it to make it more efficient. An example of this is given in [LTB01] in the parallel functional language, GpH. Such languages have the advantage that the program can be transformed formally using equational reasoning, in the same way as the transformations in this thesis. However, the *framework* is still informal.

There are a number of formalised incremental approaches. Gorlatch in [GL95] presents an incremental system based on BMF. Clint, Fitzpatrick et al. [CFH$^+$94, FHB94, FCHK96] have developed a method for deriving imperative data-parallel implementations from non-explicitly parallel functional specifications, by automatically applying a series of rewrite rules. Loyens and Van de Vorst [LV90] manipulate functional specifications using invariants to derive parallel programs.

The following sections examine a few incremental parallel approaches in more detail.

8.2.3 TwoL

TwoL [RR96, RR95] is an incremental parallel approach, introducing decisions one at a time. It is focussed on the *group-SPMD* (GSPMD) style of programming, allowing the expression of both task and data parallelism.

In the *restricted SPMD* model, programs are structured in a series of stages or *macrosteps.* Each macrostep is made up of a computation step in which each processor only performs local computations, followed by a communication step when processors can communicate with each other. Group SPMD divides a program up into modules, each of which executes according to the SPMD programming model. Modules may be executed one after the other or concurrently, in which case each module is executed by an independent group of processors in parallel.

Such a program structure means that the main decisions to be made are *scheduling* or allocation, in which modules are assigned to processor groups, *load balancing*, in which these groups are allocated to processors so that the load on each processor is more evenly distributed, and *data distribution*, when the internal data distributions of the modules are determined.

TwoL uses a fixed derivation structure. The program starts off as a parallel specification including notes of all potential parallelism. Then each of the decisions is made for the program, one after the other, in the order given above. The decisions can be specified by the programmer, or alternatively can be worked out by the compiler using algorithms as in Section 8.1. In order to aid this, TwoL provides a simple cost analysis model for modular programs.

TwoL uses an imperative language in which information about potential parallelism and the decisions can be expressed using directives or annotations, and calls to library redistribution functions. The language is specialised to TwoL, but the derivation process targets standard languages, basically C with communication operations, either ones specific to a particular machine, or ones from the message-passing libraries MPI or PVM.

8.2.4 Combinator methods

Combinator programs have been produced and implemented in a variety of ways, some of which are incremental. A standard such skeletal derivation involves transformations from a specification to a program using skeletons or from one skeleton or set of skeletons to another in order to improve efficiency [DGT93, BGP93].

For example, Fradet and Mallet [FM00] present an incremental derivation process for skeletons. The program is formally transformed from one version to another through a fixed series of stages within a functional language, before it is transformed to its implementation in C+MPI. FAN [GP99] and SAT [Gor96] are other general transformational frameworks for skeletons, operating in a similar way, and Pepper [Pep93] gives another step-wise skeletal derivation process, this time for systolic algorithms.

This work is similar in several ways to the work in this thesis. Programs are transformed formally, through a series of stages during which the program is optimised. These transformations may take place within a functional language, before being finally transformed into a language such as C+MPI.

However this work still restricts the combinators to a fixed set with fixed implementations. Skeletons are only transformed into other skeletons at a similar level of abstraction, whereas, in this thesis, multiple levels of abstraction are used, so that the penultimate program can be quite similar to the final implementation, making the final transformation particularly simple. In contrast to this, the skeletal methods tend to have a sizable jump in this final transformation.

Incremental methods have also been applied to other parts of skeleton derivation, such as the compilation of a skeletal program, as was mentioned in Section 8.2.1. An example of this is the compilation of individual skeleton functions. HDC (Higher-order Divide and Conquer) [Her00, HL00, HLG+99] is a skeletal functional language, originally aimed at divide-and-conquer algorithms. It is syntactically a subset of Haskell but is strict. In HDC, incremental methods are used to derive skeleton implementations manually, which can then be linked with HDC code. This is more closely related to the work in this thesis than the previous methods, because a sequence of equational transformations are applied to the body of each skeleton to bring it closer to C+MPI, before it is finally transformed into C+MPI itself.

This skeletal compilation process is not an instance of compilation by transformation, as the actual compilation process uses the implementations which have previously been derived by hand, as in this thesis.

8.2.5 Summary

Despite its usefulness, incremental programming also has some disadvantages. It can sometimes increase program development time, for example, if the program only involves a few simple decisions. In such cases it may be quicker to make all of the decisions at once and write the program directly in its final form. Another disadvantage is the time and energy needed to set up the framework of an incremental approach in the first place. For example, there are often transformation rules which need to be given or a specialised language or subset of a language to specify. Lastly some, though not all, incremental approaches have a fixed structure, requiring decisions about aspects of the program or communication which are not relevant to or important in the current situation. They may also hamper one from making other decisions and optimisations which (are) relevant.

However, incremental methods can produce many benefits, particularly for large problems or problems involving many decisions. In these cases they help to separate out the concerns, making the decisions easier and the program development process clearer. They can also clarify and simplify options, which has a side-effect that cost models are easier to apply to the decision-making process.

Incremental methods also have the advantage that earlier stages of the derivation are applicable in a variety of situations. During these stages, several decisions about the algorithm to be used or the system to be targeted haven't yet been made. Therefore, if one wanted to derive the same program for a different situation, such as a different target architecture, it wouldn't be necessary to do the whole derivation all over again—the initial stages could be reused.

8.3 Abstraction

Incremental methods start with programs that are fairly abstract— there are aspects that are not specified, decisions still to be made and details to be introduced. As these decisions and details are specified during a derivation the programs become more concrete.

This raises two questions. Firstly, how can we write a parallel program and still be abstract about the parallel details? And secondly, how can we cope with the different levels of abstraction in a single methodology?

This section considers various answers to both of these questions, some of which are used in existing methods.

8.3.1 Methods in existing incremental approaches

Parallel combinator methods such as those described in Section 8.2.4 rely on higher-order functions, such as skeletons, to abstract away from the parallel details. These combinators usually represent whole patterns of communication and computation, and the program merely calls the combinators without knowing how they are implemented.

Combinator derivations usually replace combinators by other ones from the same set, especially in traditional skeletal methods. Since the combinators in most methods are all at a fairly high level of abstraction, the level of abstraction in the program doesn't change drastically. However there may be minor changes in abstraction level as some combinators may be a bit more specific than others.

However, some methods use combinators which aren't so abstract. For example, Fradet and Mallet [FM00] use combinators which model particular communications operations, even single point-to-point communications. These combinators are much more explicit about the parallelism and hence much less abstract than traditional skeletons. However, as with other methods, most of the combinators provided are on a similar level of abstraction, even though this level is different from usual. As before, programs at different stages have similar levels of abstraction.

HDC (see Section 8.2.4) is another skeletal method that uses incremental programming. Initially the parallelism is encapsulated in these skeletons, but later on list comprehensions are used to express data-parallelism in the derivation. Operations on elements of a list are assumed to be executed in parallel.

Clint, Fitzpatrick et al. [FCHK96] also use functional languages, although they use a version of lambda calculus instead of Haskell. They focus on matrix algorithms which map easily to array and vector processors, so that arrays and array functions can be used to represent the data structure and the set of processors.

Other methods, as well as functional languages, are possible. Loyens and Van de Vost [LV90] use predicate calculus to express constraints such as data distributions. These then modify the program so that it is specialised to these constraints. Another example is BMF, on which the incremental system proposed by Gorlatch and Lengauer in [GL00] is based.

TwoL abstracts away from parallel details by using a non-executable form of program called a *parallel frame program*. As it isn't executable, there is no need for it to deal with all the parallel details. The program can be made more concrete by adding specialised pieces of code which give particular information. For example, the annotation `[on p]` specifies that the given module is executed on p processors. This is clear and simple, but has the disadvantage of not being executable, which limits the ability to check the correctness of intermediate stages and to profile the initial program.

8.3.2 Other abstraction methods

Any programming language is in a sense an abstraction mechanism, abstracting away from the details of the registers and machine code instructions. However some are more abstract than others, and parallel programming is one area in which this is clearly true.

I have already discussed languages which are explicit about the parallelism to varying degrees. The more implicit a language, the more it abstracts away from the details of the parallelism.

MPI, an example of an explicit language, encapsulates the communication in pre-defined functions. Although these are explicit about the parallelism, they still abstract away from the details of the network and routing. Collective communication functions, furthermore, implement patterns of communication across the whole system. The programmer can use them without knowing what individual communications are needed to implement them.

Alternatively one can specify the parallel details separately from an executable program. This approach is taken in TwoL as previously discussed. This has the advantage that there is no need to give all the details in a specification—it's not going to be run.

It is also possible to write programs which simulate a parallel machine's execution on a single processor, rather than actually running on a parallel machine. This can be done by setting up a model of the parallel machine which is then modified as the program executes. As it is only a model and not a real machine, only as many details as required need be given. Such models can also be later mapped to real machines during a compilation stage and run. This method is often used in compilers and interpreters for sequential machines to represent the state of different parts of the computer, and is used in this thesis.

It is fairly common in parallel programming to use arrays or sequences to model parallel machines, especially data-parallel ones. Each element of the array is assumed to lie in a different processor. ZPL

[LS93] uses this method, as does NESL [Ble95, Ble96], which also models data-parallel operations by single operations applied to whole sequences.

8.3.3 Separating coordination and computation

Another way to increase abstraction is to separate the parallel aspects from the algorithmic ones.

Implicit languages do this by leaving all of the parallel aspects to the compiler, so that only the functional aspects are given in the program (see Section 8.1.1). Other approaches specify both coordination and computation aspects within the program, but do so separately.

Some languages, such as GpH and HPF, use compiler directives or functions to specify the parallelism, isolating the parallelism to these directives. However these can sometimes be fairly low-level, and therefore occur in the program so frequently that the parallelism aspects end up being intermingled with the program code anyway.

Strategies, as described in Section 2.3.6, aim to improve this situation in GpH. They encapsulate the dynamic behaviour (e.g., evaluation degree and extent of parallelism) of a data structure in a function or strategy. This can then be attached to an expression to control how its value is evaluated. The main expression then need not contain the details of the dynamic behaviour. Although this is a good way of separating the concerns and achieving greater abstraction, it is limited. It can only specify the behaviour of distinct parts of a data structure. If an expression evaluates to a single value, it cannot control how the evaluation proceeds.

The method used in this thesis, APM functions [OR97], described in Section 1.1.3, is related to this as it also encapsulates the parallelism within specialised functions. However, unlike strategies, APM functions may also include algorithmic content, and are used in the same way as ordinary functions. This allows greater flexibility and greater compatibility with target languages such as C+MPI, at the cost of possibly less separation between algorithm and parallelism. If a language such as GpH itself were used as the target, then the APM functions could be more like strategies.

APM functions are a form of parallel combinators which isolate the parallelism by restricting it to sets of higher-order functions. A program can be built up from combinators, without the programmer having to include the details explicitly. In general, parallel combinators may have to belong to a fixed set, as in the skeleton approach, or they can be distinguished in some other way, e.g., using annotations.

SAT [Gor96, GL00] is one combinator approach which uses BMF [Bir89] to express and transform the programs, and is based on a higher-order function called a list homomorphism. This is a general combinator capturing many different patterns of communication and computation. For example, *map* and *fold* are both list homomorphisms. This means that the set of parallel combinators is very small, and that transformation rules can be reduced mainly to those involving this homomorphism function, thus simplifying matters. This is at the cost of a loss in generality.

Another approach is to use a coordination language [GC92]. This can be closely related to the computation language, which is the approach taken in the functional coordination language Caliban (see Section 8.1). Alternatively, it can be a completely separate language, such as Linda [CG89], which can then be used with a variety of computation languages. In these, computation is separated from communication allowing each to be dealt with separately. For example, Caliban uses annotations on functional expressions to specify task placement (and hence communication).

8.3.4 Summary

There are many methods of abstracting away from the parallel details. This thesis picks up on some of the ideas which have been described in this section. It models the parallel system using a functional language. Due to the nature of such models, different models can be used for different stages, allowing different levels of abstraction. Parallel combinators are also used, and they allow the separation to some extent of parallel and algorithmic concerns. Parallel operations are encapsulated in combinators which are annotated to identify them as such. This is closely related to the strategy approach, but there isn't such a clean separation as in Linda, for example, because we want the retain the ability to model languages such as C+MPI where the communication is closely coupled with the program.

8.4 Formality in parallel programming

Formality is a word which is often used in computer science. It forms an important part of the methodology in this thesis. Webster's dictionary [Por13] defines "formal" as

> Devoted to, or done in accordance with, forms or rules; punctilious; regular; orderly; methodical; of a prescribed form; exact; prim; stiff; ceremonious.

Clearly, the definition of the word itself is far from exact. Many people use "formal" to mean different things, even in the restricted area of computer science.

8.4.1 Formality in expression

When some people say something is formal, they mean that it is written down in a precise or prescribed form and not just a fuzzy idea in one's head. However this is still general and vague. For example, both a contract and a programming language are formal in this sense, but the semantics of a program is much more precise than the semantics of an English description.

However, for many people this is not precise enough. If the semantics of a programming language isn't clearly defined then the one cannot be sure what the program does. The same program can behave in different ways on different machines or at different times. For example, among many other things, the C language doesn't specify what the result of division by zero is [HSJ95]. The effects are unpredictable.

Without a clear correspondence between a program and its meaning it is hard to transform a program so that its meaning is retained, and these transformations cannot be proved correct. Nevertheless this rather vague description of a program's meaning is the predominant method of dealing with programs as most people are satisfied with running the program and observing its behaviour in a variety of situations.

More precise descriptions of a program's meaning, or semantics, have been produced for several languages because of the problems above. These can be used to reason about and transform programs. For example, BSP [SHM97] has a formal semantics for which algebraic laws have been given [JMC96] which could be used to develop programs. Formality can also be used to prove operational properties. GpH has a formal operational semantics [BKT00]. This allows us to apply equational reasoning to the parallel behaviour of GpH, and reason about properties such as number of processors, space and time used. NESL is another example with an operational semantics, and time and space bounds have been proved for its implementation on various machine models [BG96].

Sometimes the "prescribed form" definition of formal is taken further. For something to be formal, it must be written down in a very particular way, often involving many mathematical symbols. This is the approach taken by BMF [Bir89]. This method has many useful features. Mathematical notation is usually concise, and this makes it easier to manipulate and transform. It also has the potential to be precise, although simply using mathematical notation doesn't give it that property.

8.4.2 Formality in proofs

Other uses of the word "formal" refer to proofs of correctness rather than to programs themselves. Proofs may be formal since they use set forms or rules and are regular and orderly. They can also be formal in other ways.

Many members of the formal methods community want everything done formally from first principles. Nothing should be assumed, even basic arithmetic laws, without proof that they hold in the current language or system. Indeed they do not always hold. This leads to robust proofs. But the proofs tend to be long, tedious and complicated. It is difficult to prove non-trivial things in this way. Therefore there are various mechanical and computer-aided systems to help with this. They start from first principles and build up theorems. Examples include HOL [HOL], which is a theorem prover for higher-order logic, and Isabelle [Nip01], another theorem prover based on HOL, but which uses functional programming syntax and can be used to prove theorems about functional programs. These can be helpful but are often still fairly restricted.

It is possible to take the opposite tack and transform programs without proving the soundness of the transformations at all. This approach is often taken in compilers, especially in imperative languages where there is often no semantics which could be used to prove the rules sound. For example, Allen and Kennedy [AK87] give some transformation rules for parallelising Fortran programs. Although they prove properties of dependence graphs, and in view of these the rules intuitively make sense, they don't actually prove that the rules maintain the meaning of the program. It is, in fact, not possible to prove the rules correct as there is no formal semantics for Fortran, yet their approach is useful and can help a lot in parallelising programs.

Equational reasoning is one technique that combines the advantages of both of these methods. It uses techniques that have been shown to be sound, but without insisting on proving every detail. Some people complain that equational reasoning is not sound as Haskell has no standard semantics. However, a semantics has been given for the core subset of Haskell [PJW91] and for various extensions of it. In addition Haskell's relationship to the lambda-calculus [Rév88, Mic89] makes its meaning clearer than that of many programming languages. Extensions to Haskell are often defined using rewrite rules which convert them to core Haskell, therefore all that is needed in these cases is to prove that these rewrite rules are sound.

8.4.3 Formality in incremental systems

Proofs are closely related to incremental programming, which derives a program by taking it through a series of stages, as described in Section 8.2. One of the key parts of this process is the transformation between stages, which can be done in a variety of ways. The sequence of transformations can, if they have been done formally enough, be viewed as a proof that the resultant program produces the same results as the initial one. Therefore the issue of formality is again an issue here.

As mentioned above, some systems, especially compilation systems, simply apply transformations without proving them correct at all. These are formal in the sense that they are precisely defined but they do not form a proof of correctness.

Clint and Fitzpatrick in [FCHK96] define a set of rewrite rules using a formal grammar for their incremental approach. Although these are not proved correct, they are based on similar principles to equational reasoning and such proofs are possible.

Skeleton derivation systems have a good potential for formality, especially those which use a functional language. Transformation rules are often applied to a program to transform it from one version to the next, and in a functional language these can often be proved to preserve semantics [DGT93]. FAN [GP99] and HDC [HL00] are examples of systems which use this method.

Other skeletal systems, such as SAT [Gor96, GL00], use BMF to prove correctness. The method in [LV90] uses predicate calculus.

In these systems, proofs are possible in principle, although not all the details may have been carried through. For example, BMF has no formal semantics, although such a semantics is possible. The authors of the systems feel that there are more important things to do than fill in the details.

In the APM methodology, described in this thesis, equational reasoning can be used to prove transformation rules which are then used to derive programs. This is formal. However the methodology is also flexible and doesn't insist that such a degree of formality is always used. In proofs, it is possible to leave proof obligations to be filled in later when a more formal derivation is needed. It is especially useful to leave proof obligations of obvious facts such as arithmetic properties.

8.4.4 Automation

It is possible to automate both proof and derivation processes. Transformational compilers are, in one sense, simply automated program derivation systems.

Completely automated systems starting from an abstract specification have the advantage that the programmer need not worry about any of the parallel details. However, as mentioned in Section 8.1, these parallel details are often difficult to determine, and may involve "Eureka" steps which require human insight, as least at the moment.

The approach taken in most computer languages, such as C+MPI, is to allow the programmer to specify these details in the initial program and then the automation (compilation) process takes over and produces the final program.

Incremental systems sometimes involve automation by asking the programmer to make decisions during the earlier phases in the process, and then carrying out the remaining stages automatically. An example is the system proposed by Fradet and Mallet [FM00] in which the final compilation stage contains the fiddly but straightforward details of the transformation to C. This is a characteristic of several skeleton systems because the programmer produces a skeletal program incrementally but isn't involved with its implementation on the target, e.g., [DGT93].

Other systems, such as TwoL, allow automatic decision-making, but also give the user the choice of making the decisions by hand, and then feeding the results in to the compiler.

The work in this thesis is still at an early stage so that all the transformations must be done by hand. However, many of the stages, in particular, the tedious detailed stages, could be automated—it doesn't really matter whether a person or the computer does these stages. Other stages require decisions, so that human input may be useful. A choice could be offered in these cases between human and automatic transformation.

There are also other machine tools which can be used to ease the transformation process. Era (Equational Reasoning Assistant) [Win] is designed to aid equational reasoning and so is particularly relevant for this thesis. It offers a half-way house between automation and a completely manual transformation. The programmer can give the initial program or expression and choose which rules are to be applied to which parts of it. Era also supports induction, and lets one build up a library of already-proven theorems. However, it is not currently being developed, and further work on it is required before it can be used in the methodology.

8.5 Flexibility and portability

Flexibility is a useful property for a programming approach to have. It can be applied to many facets of that approach, from the way it expresses the parallelism to the systems on which it can run. This section examines some of these facets in more detail.

8.5.1 Flexibility in expressing parallelism

Many systems use a fixed set of operations or functions to express the parallelism. This is not necessarily a bad thing. If the fixed set is chosen carefully it can be used to express a large number of problems, and express them well. A fixed set, particularly of small size, can lead to simplicity in programs, and it makes the implementation easier.

Some systems, for example, skeletons and BSP, use a fixed set of fairly high-level functions to express the parallelism. This has the advantage that each of these operators can be carefully written and optimised to run efficiently. However this also reduces the expressiveness of the language. It is still Turing-complete, but as the functions encapsulate whole patterns of communication it can be difficult or even impossible to use different patterns.

In other cases, the fixed set of operations is much lower-level, for example, in MPI or even GpH. In such cases the set may need to be fixed, as allowing the user to add to it would create many implementation difficulties. There may also be a fixed set of operations which the underlying system is capable of performing. However, as the operations are low-level, the user can use them to create many patterns of communication. This makes them more flexible than higher-level systems.

However in such methods, the parallel details get mixed up with the algorithmic details as described in Section 8.3, making the programming process difficult. In GpH, strategies [THLPJ98] have provided one successful way of dealing with this, keeping parallel details separate while at the same time letting the programmer write new parallel strategies and patterns of parallelism.

APM functions in this thesis aim to provide a similar possibility, separating out parallel details and aiding reuse, while letting a programmer write his own functions.

8.5.2 Flexibility in the methodology

Flexibility is a useful property for a methodology to have. If it's possible to do something in multiple ways, instead of in only one way, then it's more likely that one of these ways will fit a given situation. It is also helpful for a methodology to allow new methods to be developed if new, and possibly unexpected, situations may arise. In a general methodology, not all of its stages are relevant to all programs, or at least not always in the same order. Sometimes it would be useful to delay a decision until more information is known, or to apply known information to one part of the program, but leave another part until later. These are all ways in which a methodology may be flexible.

If one writes a program without using a methodology, then, of course, there is great flexibility. You can write your programs in any way you like. But this flexibility is unstructured. At the other end of the spectrum lie systems such as compilers, which usually have a fixed, inflexible structure [ASU86].

However occasionally compilers obtain a slightly greater degree of flexibility by allowing programmer input. For example, Fradet and Mallet in [FM00] describe a compiler in which the programmer can help to make the decisions about data distributions. This is useful in situations when it is difficult for the computer to make the decision well.

Programming methodologies tend to be more flexible because human interaction is allowed. Despite this, there is often a fixed set of stages in a fixed order, e.g., TwoL and, to a lesser extent, FAN [GP99]. Using a fixed set of stages like this has many advantages. Sets of transformation rules only need to be developed between stages that are next to each other in the fixed order. The programmer is clearer about what needs to be done at each point, and cost models are easier to apply and the possibility of automation more feasible.

In general there is a tradeoff between flexibility and structure. A structured approach organises the process, making sure that the key stages are carried out and aiding the process. However this reduces the flexibility of the approach—program development must be fitted into the given structure which might not be applicable to any given situation. On the other hand extremely flexible approaches often do not organise or guide the process at all. However, there are approaches that lie between the two extremes. They are structured but that structure is flexible and can be changed. TwoL is an example of such an approach. It allows the programmer at each stage the choice of deferring a decision to the compiler. This method still uses a fixed set and order of stages, just hiding some of them from the programmer. In many systems, trivial decisions can also be made for stages which aren't important in the current situation.

This thesis uses a similar idea to TwoL, but tends more towards the flexible than the structured approach. It chooses to give up some of the advantages of fixed stages for the advantages which flexibility brings.

8.5.3 Flexibility in languages

Sometimes it is useful to be able to target a particular language. This may be to produce compatibility with legacy code. If the existing system is written in Fortran, say, then it may be necessary to write the new code also in Fortran, or at least to have the ability to call Fortran subroutines. Programmers may also prefer to program in languages they are familiar with rather than having to use a new language.

Linda (see Section 8.3.3), for example, is a coordination language which can be used with a variety of computation languages. This eases the parallelisation of legacy code, and allows the programs on different processors to be written in different languages. A programmer still has to learn a new language, Linda, but much of the code remains in a familiar language.

Similar points apply to other languages and systems. For example, the message-passing library, MPI, works with C or Fortran.

The methodology in this thesis can target a variety of languages, although the structure is not yet in place to do so. This can help with legacy issues. However much of the derivation involves Haskell, and the programmer would therefore have to learn at least the basics of this language. Nevertheless Haskell is used because of its various suitable qualities (see Section 1.3.3).

8.5.4 Flexibility in target systems

Portability is also a facet of flexibility. A portable program is flexible in that it can be run on more than one machine or type of machine. Portability is an important issue in parallel programming, because of the wide range of machines available and the rate at which these machines become obsolete. However it has also proven problematic, in part due to this wide range of machines. There is no unifying model for parallel machines in the same way that there is for sequential ones. Many programs written to run on one machine simply cannot run on another. Different machines also have different features, and so programs for them have different optimisation techniques. A program optimised for one machine will often have terrible performance on another one.

Many languages nowadays are designed to run on more than one machine, therefore attaining a degree of portability. It is common to focus on one type of machine or architecture. For example, MPI is designed for distributed-memory message-passing MIMD machines. It is therefore implemented on a range of tightly-coupled, massively-parallel processing machines and networks of workstations. Although this misses out a lot of parallel systems, these are the key systems in use today, and therefore MPI remains popular.

HPF is designed to be largely architecture-independent, so that it can work with a large spectrum of multi-processor models, SIMD as well as MIMD, shared as well as distributed memory. Coordination systems [GC92], such as Linda, can also in principle be used to coordinate all kinds of parallel systems, and, in fact, all kinds of ensembles of processing items.

Portability is also an important issue for skeletons. In order to gain portability, multiple implementations for a single skeleton can be developed, one for each machine or architecture [DFH+93]. When a skeletal program is compiled the appropriate implementations of its skeletons are chosen.

Some systems take an alternative approach by targeting a portable language. For example, HDC (Section 8.2.4) and TwoL target C+MPI. In this thesis, we use this approach to target C+MPI, but (in common with systems such as TwoL) the ideas behind the methodology are more general than this and can target multiple languages.

The methodology also achieves a degree of portability by virtue of its nature as an incremental methodology. As details are introduced the program becomes more concrete and more tailored towards implementation on a particular machine. The transformations and decisions in the methodology are ordered so that the ones most closely linked to the machine come late in the derivation. This means that the earlier stages may be applicable to a variety of machines, and can therefore be reused in a derivation of the same program for a different machine.

Chapter 9

Conclusions

This chapter summarises the content and contributions of the thesis, before discussing the feasibility of the methodology and its effect on several areas of parallel computing. Finally some suggestions for further work are presented.

9.1 Summary

This thesis has presented a methodology for developing parallel programs, in which executable parallel programs are derived incrementally from high-level specifications. A specification is given initially in mathematical notation and transformed into an abstract functional specification. This is then transformed through a series of stages, during which additional information is given about the program, the target architecture and the parallelism. Finally the program is transformed into the target language, which may or may not be functional.

In particular, this thesis has developed and extended earlier ideas and preliminary work. It has presented the basic infrastructure needed in the methodology, in the process of which it has investigated how parallel systems can be modelled and manipulated in Haskell, and how the resultant programs can be transformed. The basic methodology has been augmented with the ability to introduce and reason about some key parallel programming features, including data distributions and program optimisations. This work was supported and demonstrated using case studies.

9.2 Contributions

The goals of the thesis were set out in Section 1.2 at the start of this thesis. They have now been fulfilled as described below:

- The thesis has clarified the essence of the methodology, explaining and discussing important issues throughout the thesis, but particularly in Chapters 2 to 4.
- Chapter 2 presented a detailed structure for the methodology. It identified the methodology's key stages, and gave details of the APMs and their implementations for these stages and sample other stages.
- This chapter also showed how parallel systems can be modelled in Haskell at different levels of abstraction, providing different amounts of information about the parallelism.
- Chapter 3 focused on the transformations in the methodology, and demonstrated how the Haskell models and APMs can be used to support these transformations.

- It then gave the transformation rules and lemmas needed to perform a range of simple derivations of C+MPI programs from start to finish. These included rules for transforming mathematical specifications into Haskell, performing various optimisations, dealing with monads, and introducing input and output details.
- The chapter also divided these transformations into logical and manageably-sized steps, which dealt with a single feature or decision at a time. Some of these introduced optimisations, such as load balancing, while others increased the level of parallelism detail. Particularly complicated steps were divided into smaller, more manageable ones. For example, the introduction of monads was split into two main parts. Firstly the basic monad was introduced, and then variables were used to store intermediate values.
- Chapters 2 and 3 also considered data distributions and various program optimisations within the context of the methodology. They extended the basic methodology with the ability to introduce and reason about these features.
- Chapter 4 described how decisions can be made in the context of the methodology, and illustrated this with the choice of parallel or sequential implementation of a function, the choice of data distributions and static load balancing.
- The last few chapters, 5 to 7, then demonstrated the methodology in practice through two case studies. The first, in Chapter 5, was simple: it demonstrated and motivated the main points of the basic methodology. The second, Gaussian Elimination, described in Chapter 7, was larger and hence placed the methodology under greater stress. In particular it demonstrated the use of data distributions. In addition, Chapter 6 focused on part of this case study, demonstrating how detailed communication-specific optimisations can be introduced into a program.

In summary, this thesis has:

- investigated how parallel systems can be modelled and manipulated in Haskell, and how the resultant programs can be transformed.
- brought the methodology significantly closer to a usable state, providing the support needed for the derivation of basic C+MPI programs, and augmenting this to cope with some additional features.
- critically assessed APMs after carrying out detailed work on them. This assessment continues below.

9.3 Feasibility

Even though this thesis has demonstrated the APM methodology on case studies, it is important to ask if it is feasible in practice for real-world programs. Even in the small case studies presented in this thesis, it is obvious that the derivation process is not easy. Several of the transformation rules and lemmas are complicated and some familiarity with program transformation is needed in order to apply them accurately. In addition, the APMs and code in the later stages use a complicated model of the system, and can therefore be hard to manipulate and use, at least at first.

Nevertheless, there are situations in which this difficulty is worth it. A complete formal derivation constitutes a proof that the final program meets the initial specification. Such proofs can otherwise often be difficult to produce.

However, in general, it is not reasonable to expect a programmer to use the methodology in the same way as in the case studies, carrying out every transformation formally and by hand. The complicated transformations increase programming difficulty, rather than decreasing it, unless the programmer has a high degree of familiarity with program transformation and functional programming. However, this is not the only way in which the methodology can be used.

One alternative is a large-step derivation, in which some of the steps are combined, carried out informally or skipped altogether. The transformations presented in this thesis have been kept small to simplify them, but this is not necessary. Such small steps can be tedious to do by hand, and so it may be preferable to skip some of them, or to combine several into a larger transformation. Alternatively, steps can be performed informally if the programmer feels capable of doing so correctly.

This decreases the time needed to carry out a derivation, and removes some of the complex program transformation. Although a formal proof of correctness no longer exists, it is possible to obtain it later on, if necessary, by redoing the missing steps formally. This method also allows more rapid prototyping, as intermediate versions of the program are obtained more quickly. Running these can help the programmer to catch mistakes and to be convinced of the program's correctness.

However, even this method requires detailed program manipulation and transformation, and it seems that, for the methodology to be really useful in practice, something more is required.

This can be provided in the form of tool support for the transformations, especially the more complex ones. For example, tools can apply transformation rules chosen by the programmer, and can keep track of the current state of the program. In addition, those transformations which don't involve user decisions can be automated. This reduces the amount of work the user has to do, and the amount of detail he has to keep track of. Such support has not yet been provided, but is described further in Section 9.5 as a possibility for future work.

9.4 Conclusions

The methodology presented in this thesis helps with parallel programming by addressing four key challenges mentioned in the introduction. Firstly, parallel programs can be very difficult to write because of the increase in the number of details to keep track of. This is exacerbated by an increase in the number of design decisions, such as the choices about the placement and movement of data. Portability is also a challenge because the efficiency of a program depends heavily on the target machine. Finally, there is the challenge of proving a program correct.

This section discusses how well the methodology handles each of these challenges.

Programming difficulty

The APM methodology addresses parallel programming difficulty in two main ways. Firstly, it makes it easier to keep track of the parallel details by introducing them gradually within a structured framework instead of all at once. Secondly, it eases the introduction of these details by encapsulating them in APMs when they are first introduced. This was demonstrated in particular for data distributions. In later stages these details are moved from the APMs into the main program. Sets of program transformation rules ease this movement.

However, as mentioned above, the programming process is still not easy. Several of the transformation rules and lemmas are complicated and require familiarity with program transformation. The code in later stages can also be complicated. It is, however, possible to simplify this process as discussed above.

Decision making

The methodology uses an incremental derivation approach that allows decisions about the program and its parallelism to be made one at a time, instead of all at once. This simplifies the decision making process because only those issues relevant to the current decision need to be considered at each stage.

The methodology is also flexible, allowing multiple decision making techniques to be employed, depending on the situation. More complex, formal methods can be used when required, while simpler, informal methods can be used if they are adequate and applicable. This can simplify the programming process, and allows advances in decision making techniques from other areas of programming to improve the decision making in the methodology.

Although the methodology makes decision making easier, it is still not always easy. The choice of data distributions in Chapter 6 reveals that the separation of decisions is not always as clear-cut as might be desired. The possible future optimisations for each distribution had to be taken into account when making the decision. This example also demonstrates that the cost analysis can be complicated and lengthy.

Nevertheless, without the degree of separation provided by the methodology the decision would have been appreciably harder. It may also be possible to reduce the complexity of the decision-making process through the use of work in areas such as cost models. Some research on combining cost models more closely with APMs has already been carried out [ORR01].

Portability

It is desirable to have programs that can be run efficiently without alteration on different target machines, and this area has been the subject of some research, especially in implicitly parallel languages and skeletons, both of which are described in Chapter 8. Some of the work in this area can be reused in the methodology by targeting the appropriate programming languages and systems.

However, the methodology's general approach to portability is slightly different. Rather than trying to produce programs that run efficiently without alteration, it aims to reduce the amount of alteration required. This allows a larger range of machines and languages to be targeted.

This aim is achieved through the methodology's incremental structure, which orders the transformations and decisions so that those more closely linked to the target machine come later in the derivation. Early stages apply to a variety of machines and languages, and can therefore be reused when deriving the same program for a different target. The more closely related two machines or languages are, the more stages their derivations can share, and the less extra work is needed.

Proofs of correctness

The methodology is designed to allow proofs of program correctness to be produced if required. This increases the reliability of and the users' confidence in a piece of software. All of the intermediate programs are written in Haskell, a pure functional language, thus allowing equational reasoning to transform them from one stage to the next. This technique preserves correctness, and so the transformations can be proved correct—the final Haskell version can be proved to give the specified results.

However, it is often the case that the target language does not have a formal semantics, and the final transformation to the target cannot be proved correct. The methodology attempts to overcome this problem by bringing the final Haskell version as close as possible to the target. This decreases the user's uncertainty in the program's correctness, but does not remove it completely.

Another problem arises because correctness proofs can be at odds with the methodology's program development role, as explained in Section 9.3. If all of the steps are done formally to produce a correctness proof, the program development process is lengthened and complicated.

The methodology can therefore be used in more than one way. If proofs are needed, each step can be carried out in detail, but otherwise, steps can be skipped and simplified as mentioned above. It is likely that, in general, proofs will only be produced for individual, particularly complex steps in a derivation, and that complete proofs will only be needed in special circumstances.

9.5 Further work

As a whole the methodology shows promise, tackling the four areas of weakness identified above. However, there are still many difficulties and weaknesses to be addressed, providing much scope for future work.

- A natural continuation of the work in this thesis is to develop the methodology to cope with a greater range of parallel features and target languages. As yet, the infrastructure is only in place for simple, numerical programs in C+MPI. If the methodology is to be useful for real-world programs, this must be extended.

  In particular, more program optimisations need to be made available, and the existing ones need to be extended. For example, static load balancing is, at present, limited to a small range of programs, but it could be adapted to many more.

  This necessitates a greater range of APMs and transformation rules. They can be built using the principles in this thesis, but the greater the range of features, the greater the number of challenges which are likely to be involved.

- There are also many specialised parallel features whose investigation should prove interesting. For example, nested parallelism is mentioned in this thesis only briefly, and yet the set-up of the APMs fits well with its expression. More difficult is its implementation in languages, such as C+MPI, which don't provide (explicit) support for it. It should be interesting to see how previous work on languages which do support nested parallelism, such as Nepal [CKLP01], can be used within the methodology.

- Some parts of the derivation are not supported very well at present. In particular, individual level programs are not runnable. It would be useful to incorporate the ability to run them to allow greater prototyping ability. Further investigation of the individual level semantics suggested in [O'D01] may also prove interesting.

- Further case studies would be valuable for further evaluation of the methodology and identification of weaknesses and areas for future work. The case studies in this thesis were instrumental in the development of the methodology to its current state. In particular, they highlighted data distributions and program optimisations as areas for detailed work, and provided test-beds for these features. Other case studies could serve a similar role in other areas. It would be particularly interesting to try the methodology out on larger programs, and investigate the separate development of different parts of a program.

- The methodology's claim to portability is particularly weak due to a lack of case studies. It would be useful to investigate this area in practice, and see how well the initial stages stand up to reuse. The infrastructure needed for a different kind of target language, for example, HPF [MC97], could be developed and the derivation of a program in that language could be compared to that of the same program in C+MPI. This may lead to alterations to the initial stages to make them more widely applicable.

- The possibility of tool support and automation to ease the program development process has been previously mentioned in this thesis. Many of the stages, especially the latter ones, are detailed and require little human "intelligence". Such steps could be automated, as in a compiler. However, it is not useful to automate all of the stages, as many require decisions that are difficult to make with automatic methods, and it is also useful to retain the ability to incorporate human insight, or "Eureka", steps. A choice could be offered at each stage in the methodology between human and automatic transformation.

  The non-automated steps do not have to be carried out solely by hand. They can be supported by a reasoning assistant such as Era [Win] to ease transformation and help make decisions. To do this, further work on reasoning assistants may be needed. Work on improving automatic decision making may also prove useful.

Although there are problems and difficulties with the methodology, few of these are inherent in the nature of the methodology itself, and ways of overcoming these problems have been suggested. The methodology itself, although still in its preliminary stages, shows promise, and goes some way towards removing several of the key problems with parallel programming.

Appendix A

Lemmas and Rules

This appendix contains the Haskell definitions, properties, lemmas and rules which are given and used in this thesis along with some other associated ones which may be useful in the methodology.

The lemmas can be proved using equational reasoning, sometimes using associated techniques such as structural induction. As only *finite* sequences are used, induction is usually sufficient and techniques such as co-induction are not necessary. In the interests of space, and in order not to bore the reader, only a selection of such proofs are given. These are in Section A.6, and others are similar. The rules are vertical transformations, as discussed in Section 3.4, and can be proved using observation functions together with equational reasoning, as discussed in that section.

The following notation is used in this appendix and throughout the thesis. A lemma has the general form $E_1 == E_2$ and can be applied in either direction. In fact, the symbol $==$ indicates the equivalence of expressions in general. It is used instead of $=$ to avoid ambiguity with the $=$ in programs. A transformation rule, on the other hand, has the general form $E_1 \Rightarrow E_2$, and is unidirectional. In addition, the notation $E[]$ stands for an arbitrary expression with zero or more holes. $E[e]$ is this same expression with the holes filled in with e, while $E[x/y]$ is the expression E with every free occurrence of y replaced by x. Fs is a finite sequence of expressions, usually equations in **let** expressions or monad bindings in **do** expressions. All lemmas and rules assume that renaming is performed to avoid name capture problems.

The lemmas and rules are divided into a few categories to order the presentation.

A.1 Lemmas dealing with general function manipulation

There are various transformations which can be done on Haskell programs in general. Most basically, alpha, beta and eta conversion can be performed to rename variables, apply functions to parameters and extend functions. Equational reasoning is built on this basis.

In particular, inlining is a common name used for equational reasoning when it is used to replace a function call with the function definition in an expression.

Here are some more lemmas:

Lemma 1 (Parametrise function)

For any function f, expressions E, a of appropriate types, and equations Fs.

$$\begin{array}{lcl} f \;=\; E[a] & == & f\;x \;=\; E[x] \\ Fs & & Fs[f\;a/f] \end{array}$$

This introduces an extra parameter, x, to f, making f more general. This is useful when restricting specifications (see Section 5.3).

Rule 2 (Instantiate types)

For any $f :: T(\alpha_1, \ldots, \alpha_n)$, $x_i :: T_i$, $g :: T(T_1, \ldots, T_n)$
where α_i are polymorphic types and T_i specific types such that
$f\ x_1 \ldots x_n = g\ x_1 \ldots x_n$.

$$f\ x_1 \ldots x_n \quad \Rightarrow \quad g\ x_1 \ldots x_n$$

This is rather a trivial lemma, but it means that a call to a function, f, which uses specific types, can be replaced by a call to a function, g, which only has to work for those particular types.

A Haskell program is composed of multiple function definitions. Therefore it can be considered to be a big implicit **let** expression in which the result of the **let** is whatever is typed into the interpreter in hugs or is given in *main*. Therefore all of the lemmas in the next section also apply to it.

A.2 Lemmas for let expressions

Haskell **let** expressions limit the scope of function and variable definitions. They can be defined as follows:

Definition 3 (Let expression)

For any variable x such that x does not appear in $E2$.

$$\begin{array}{lcl} \textbf{let} & == & E2[E/x] \\ \quad x = E & & \\ \textbf{in} & & \\ E2 & & \end{array}$$

This binds the value of a fresh variable x to E in $E2$. This is similar to the definition of **where** and so the following lemma holds:

Lemma 4 (Equivalence of let and where)

$$\begin{array}{lcl} \textbf{let} & == & E2 \\ \quad x = E & & \textbf{where}\ x = E \\ \textbf{in} & & \\ E2 & & \end{array}$$

In this section,

$$\begin{array}{l} \textbf{let} \\ \quad Fs \\ \quad x = e \\ \textbf{in} \\ E \end{array}$$

means that $x = e$ occurs somewhere in the set of let equations, and Fs are the remaining equations.

Property 5 (Rearrange equations in a let expression)

For all Fs let equations, $x, E :: \alpha$, $y, E2 :: \beta$, $E3$ an expression of any type.

$$\begin{array}{lcl} \textbf{let}\ Fs & == & \textbf{let}\ Fs \\ \quad x = E & & \quad y = E2 \\ \quad y = E2 & & \quad x = E \\ \textbf{in} & & \textbf{in} \\ E3 & & E3 \end{array}$$

As the equations in a **let** expression are simply bindings in an environment, it doesn't matter what order they are given in.

This can be applied multiple times to rearrange the order of equations however one likes. So all the following lemmas are given with the relevant equations are the end, but can be applied wherever they occur.

Lemma 6 (Nested lets)

For any expressions $E1$, $E2$, $E3$, equations Fs and variables x, y of appropriate types such that x only occurs in $E2$.

$$\begin{array}{lcl}
\begin{array}{l}\textbf{let } Fs\\ \quad y = \textbf{let } x = E1\\ \qquad\quad \textbf{in}\\ \qquad\quad E2\\ \textbf{in}\\ E3\end{array} & == & \begin{array}{l}\textbf{let } Fs\\ \quad x = E1\\ \quad y = E2\\ \textbf{in}\\ E3\end{array}
\end{array}$$

Since x only occurs in $E2$, it is new for Fs and $E3$ and therefore it doesn't matter if it's bound to a value.

Lemma 7 (Remove redundant equation)

For any expressions E, $E2$, equations Fs, variable x of appropriate types such that x does not occur in Fs or $E2$.

$$\begin{array}{lcl}
\begin{array}{l}\textbf{let } Fs\\ \quad x = E\\ \textbf{in}\\ E2\end{array} & == & \begin{array}{l}\textbf{let } Fs\\ \textbf{in}\\ E2\end{array}
\end{array}$$

Lemma 8 (Split let expression)

For any expressions E, $E2$, function f, variables x, y of appropriate types such that y is fresh.

$$\begin{array}{lcl}
\begin{array}{l}\textbf{let } Fs\\ \quad x = f\,E\\ \textbf{in}\\ E2\end{array} & == & \begin{array}{l}\textbf{let } Fs\\ \quad y = E\\ \quad x = f\;y\\ \textbf{in}\\ E2\end{array}
\end{array}$$

This is a simple consequence of Definition 3 and Lemmas 6, and is proved in Section A.6

Lemma 9 (Move a function in a let expression)

For any expressions $E, E2$, equations Fs, functions f, g and variables x, y of appropriate types such that x does not occur elsewhere in Fs or $E2$.

$$\begin{array}{lcl}
\begin{array}{l}\textbf{let } Fs\\ \quad x = E\\ \quad y = f(g\,x)\\ \textbf{in}\\ E2\end{array} & == & \begin{array}{l}\textbf{let } Fs\\ \quad x = g\,E\\ \quad y = f\,x\\ \textbf{in}\\ E2\end{array}
\end{array}$$

Lemma 10 (Move a function in a let expression 2)

For any expressions $E, E2$, equations Fs, function f and variable x of appropriate types.

$$\begin{array}{lcl}
\begin{array}{l}\textbf{let } Fs\\ \quad x = f\,E\\ \textbf{in}\\ E2\end{array} & == & \begin{array}{l}\textbf{let } Fs[f\;x/x]\\ \quad x = E\\ \textbf{in}\\ E2[f\;x/x]\end{array}
\end{array}$$

This is based on Lemmas 8 and 7 and alpha conversion.

Lemma 11 (Move result of let expression into let)

For any expression E, questions Fs and fresh variable res.

$$\begin{array}{lcl} \textbf{let } Fs & == & \textbf{let } Fs \\ \textbf{in} & & \quad res = E \\ E & & \textbf{in} \\ & & res \end{array}$$

This is obvious (by Definition 3), but it's given explicitly because it can be useful when tidying up **let** expressions.

Lemma 12 (Move let expression)

For any expressions E, $E2$, function f and variable x of appropriate types such that
x does not occur in f separately from $E2$.

$$\begin{array}{lcl} f(\textbf{ let } x = E & == & \textbf{let} \\ \quad \textbf{in} & & \quad x = E \\ \quad E2) & & \textbf{in} \\ & & f\ E2 \end{array}$$

This is useful for moving **let**s around so that they encapsulate more or less of the program. It is used in Section A.6 to prove Lemma 38. It itself is proved simply using the definition of **let**.

Lemma 13 (Separate tuples in a let expression)

For any expressions $E1$, $E2$, $E3$ and variables x, y of appropriate types.

$$\begin{array}{lcl} \textbf{let } (x, y) = (E1, E2) & == & \textbf{let} \\ \textbf{in} & & \quad x = E1 \\ E3 & & \quad y = E2 \\ & & \textbf{in} \\ & & E3 \end{array}$$

This can be used to help simplify complicated **let** expressions so that only one APM function is on each line.

A.3 Lemmas dealing with specific functions

Specific functions have specific properties which it can be useful to use in transforming a program. This section looks at some standard functions, most of which are used in this thesis. These are valid only for *finite* lists or sequences, and many of them can be proved by induction on the list structure. Many of them are standard and have been proved many times before. Example proofs are given in Section A.6.

These lemmas are stated in terms of functions without annotations but because of the equality between the APM functions on *FinSeq*, *SeqFinSeq* and *ParFinSeq* (see Section 3.4), they apply also to those annotated with S and P.

Lemma 14 (Change *foldl*1 to *foldl*)

For any $f :: \alpha \rightarrow \alpha \rightarrow \alpha$, $a :: \alpha$ a left unit of f and $xs :: [\alpha]$ such that
$\#xs \geq 1$.

$$foldl1\ f\ xs\ ==\ foldl\ f\ a\ xs$$

Lemma 15 (Split *foldl*)

For any $f :: \alpha \rightarrow \alpha \rightarrow \alpha$ associative, $a :: \alpha$ a unit of f and $xs,\ ys\ ::\ [\alpha]$.

$$foldl\ f\ a\ (xs\ +\!\!+\ ys)\ ==\ f\ (foldl\ f\ a\ xs)\ (foldl\ f\ a\ ys).$$

Lemma 16 (Split *foldl* using *take* and *drop*)

For any $f :: \alpha\ \rightarrow\ \alpha\ \rightarrow\ \alpha$ an associative function, $a :: \alpha$ a unit of f, $xs :: [\beta]$, $m :: Int^{+}$.

$$foldl\, f\ a\, xs\ ==\ f\ (foldl\, f\ a\ (take\ m\ xs))\ (foldl\, f\ a\ (drop\ m\ xs)).$$

This is a more specific version of Lemma 15 above.

Lemma 17 (Divide up a list)

For any $xs :: [\alpha],\ m :: Int^{+}$.

$$xs\ ==\ take\ m\ xs\ +\!\!+\ drop\ m\ xs$$

Lemma 18 (Move data in *map*)

For any function F and list permutations $move$ and $move'$ such that $move' \circ move\ =\ id.$

$$map\ F\ ==\ move'\ \circ\ (map\ F)\ \circ\ move$$

This lemma is often used to move data around between processors. In this case the version of *map* used is map_P.

Lemma 19 (Function composition in *map*)

For any $g :: \alpha\ \rightarrow\ \beta,\ f :: \beta\ \rightarrow\ \gamma,\ xs :: [\alpha]$.

$$map\ (f \circ g)\ xs\ ==\ map\ f\ (map\ g\ xs)$$

Lemma 20 (Function composition in *map* 2)

For any functions f, g_1, g_2, lists $xs,\ ys_1,\ ys_2$ of appropriate types, function F such that
- $Fx = f(g_1x)(g_2x)$
- ys_1 and ys_2 are fresh

$$map\ F\ xs \qquad == \qquad \begin{array}{l}\textbf{let}\\ \quad ys_1\ =\ map\ g_1\ xs\\ \quad ys_2\ =\ map\ g_2\ xs\\ \textbf{in}\\ map2\, f\ ys_1\ ys_2\end{array}$$

where $map2\ =\ zipWith$

This is an extension of the last lemma. It may seem a bit obscure, but is given because it is used in the map-triangle case study in Section 5.4.

Lemma 21 (*length* and *take*)

For any list $xs,\ m :: Int^{+}$.

$$length\ (take\ m\ xs)\ ==\ \min(m,\ length\ xs).$$

Lemma 22 (*length* and *drop*)

For any list $xs,\ m :: Int^{+}$.

$$length\ (drop\ m\ xs)\ ==\ \max(length\ xs - m,\ 0).$$

Lemma 23 (Indices in reverse)

For any list $xs,\ i, n :: Int^{+}$ such that
- $n = length\ xs$
- $0 \leq i < n$

$$xs!!(n - i - 1)\ ==\ (reverse\ xs)!!i$$

Lemma 24 (*repeat* and *map*)

For any $x :: \alpha$, $f :: \alpha \rightarrow \beta$, $n :: Int^+$.

$$mapn\ (f\ x)\ == \ map(n+1)\ f\ (repeat\ x)$$

Where
- $map0 = repeat$
- *mapn* is *map* with n list parameters.

Lemma 25 (*repeat* and *map* - Special case)

A special case of the above lemma for *ParFinSeq* is:

For all $x :: T$, $f :: T \rightarrow T'$ (for any T, T').

$$repeat(f\ x)\ == \ map_P\ f\ (repeat_P\ x)$$

which can also be phrased as a vertical transformation rule:

$$f\ x\ \Rightarrow\ map_P\ f\ (repeat_P\ x)$$

Lemma 26 (*repeat* and *head*)

For all x.

$$x\ == \ head\ (repeat\ x)$$

Lemma 27 (Inverse of transpose)

For the standard function *transpose* described below, and any $xss :: [[\alpha]]$ such that $\#(xss!!i)\ =\ \#(xss!!j)\ \forall\, i,j :: Int^+$.

$$transpose(transpose\ xss)\ == \ xss.$$

The function *transpose* is a standard Haskell function used to swap the rows and columns in a matrix. Its implementation and use in the methodology are described in Section 3.3.4. The condition on the lemma simply states that *xss* is a matrix. The proof of this lemma (in Section A.6) needs three subsidiary lemmas below:

Subsidiary Lemma 28 (transpose xss)

For any $xss :: [[\alpha]]$ such that $\#(xss!!i)\ == \ \#(xss!!j)\ >\ 0\,\forall\, i,j :: Int^+$ and $\#xss\ >\ 0$.

$$transpose\ xss\ =\ (map\ head\ xss)\ :\ transpose\ (map\ tail\ xss)$$

Subsidiary Lemma 29 (Map head over transpose)

For any $xs :: [\alpha]$, $xss : [[\alpha]]$ such that $\#(xss!!i)\ =\ \#(xss!!j)\ =\ \#xs\,\forall\, i,j :: Int^+$.

$$map\ head\ (transpose(xs : xss))\ == \ xs$$

Subsidiary Lemma 30 (Map tail over transpose)

For any $xs :: [\alpha]$, $xss :: [[\alpha]]$ such that $\#(xss!!i)\ =\ \#(xss!!j)\ =\ \#xs\,\forall\, i,j :: Int^+$.

$$map\ tail\ (transpose(xs : xss))\ == \ transpose\ xss$$

Another *transpose* lemma is:

Lemma 31 (Transpose and map)

For all $f :: \alpha \rightarrow \beta$.

$$map(map\ f)\ == \ transpose\ \cdot\ (map(map\ f))\ \cdot\ transpose$$

A.4 Monadic lemmas

Monads are used in Haskell to express and encapsulate side-effects. They are used for this purpose in this thesis as shown in Section 2.8. This appendix is not meant to describe monads—some description is given in Section 2.8.1 and fuller details can be found in [Wad92].

Monads in Haskell are characterised by two functions.
$unitM :: \alpha \rightarrow M\ \alpha$, often called *return*, puts a value into the monad.
$bindM :: M\ \alpha \rightarrow (\alpha \rightarrow M\ \beta) \rightarrow M\ \beta$ joins together two monads, passing a value from the first to the second.

However, this second one is not usually given explicitly in Haskell programs. Syntactic sugar is used in its place.

Definition 32 (Monadic syntactic sugar)

For any monadic expressions m_1, m_2, variable x.

$$\begin{array}{lcl} \textbf{do}\ m_1 & == & bindM\ m_1\ m_2 \\ \quad m_2 & & \end{array}$$

$$\begin{array}{lcl} \textbf{do}\ x \leftarrow m_1 & == & bindM\ m_1\ (\lambda\, x \rightarrow m_2) \\ \quad m_2 & & \end{array}$$

There are a few standard monad laws that must hold for a construct to qualify as a monad. They hold for all the standard monads. These are given below using the **do** syntax, but are rarely used directly:

Rule 33 (Left unit of a monad)

return x is a left unit of *bindM*:

$$\begin{array}{lcl} \textbf{do}\ return\ x & == & m\ x \\ \quad m & & \end{array}$$

Rule 34 (Right unit of a monad)

return () is a right unit of *bindM*

$$\begin{array}{lcl} \textbf{do}\ m & == & m \\ \quad return\ () & & \end{array}$$

Rule 35 (Associativity of monads)

$$\begin{array}{lcl} \textbf{do}\ m_1 & == & \textbf{do}\ \ \textbf{do}\ m_1 \\ \quad x \leftarrow \textbf{do}\ m_2 & & \qquad x \leftarrow m_2 \\ \qquad y \leftarrow m_3 & & \quad y \leftarrow m_3 \end{array}$$

This means that the **do** notation can be written without parenthesis and in an unnested form.

Monadic lemmas can be derived from these. Some are given in the literature, for example in [BF94]. This section lists some lemmas that are of particular use in this thesis.

Lemma 36 (Introduce monads)

For any expression E.

$$E \qquad \Rightarrow \qquad \mathit{return}\ E$$

This is an example of a vertical transformation (see Section 3.4). The symbol $\Rightarrow$ indicates equality using an appropriate monadic observation function, $o(x) = \mathit{return}\ x$, rendering the above equivalence trivial. The left hand side of the lemma can be transformed into the right at the appropriate place in the derivation but not vice-versa.

Property 37 (let into do 1)

For any variable x, expression e and monadic expression M.

```
let x = e        ==       do x ← return e
in                           M
M
```

This is another basic property, resulting from the definitions of **let**, **do** and the monad laws.

Lemma 38 (let into do 2)

For any variable x and expressions $E1$, $E2$
such that • $E1$ does not depend on $E2$
• x has no parameters.

```
return ( let x = E1        ==       do x ← return E1
         in                            return E2
         E2)
```

This is a simple consequence of Property 37, as shown in Section A.6, but in a form which is more useful for this thesis. It can be applied multiple times to convert a let expression into a monadic **do** expression.

Various other lemmas allow one to move equations around or remove them if conditions on their side-effects are fulfilled. These are more useful if they are specialised to particular types of monad. In this thesis, monads are used for IO and state manipulation, and most of the lemmas needed involve state.

State monads have types of the form:

$$\mathit{State} \rightarrow (\mathit{value},\ \mathit{State})$$

or, in this thesis,

$$\mathit{State} \rightarrow \mathit{IO}(\mathit{value},\ \mathit{State})$$

where the state pairs variables with values.

In the following lemmas, $M, M1$ and $M2$ represent state monadic expressions. Property 35 lets the lemmas be used with multiple monadic expressions. These lemmas do not guarantee the same side-effects on the state, in the same order, just that the result is the same.

Lemma 39 (State Monads—Remove redundant equations)

For any variable v and monadic expressions E, M
such that • x does not appear in M before another $(x \leftarrow F)$ clause
• E does not change any variables in the state which are used by M.

```
do x ← E        ==        M
   M
```

Lemma 40 (State Monads—Move independent equations)

For any variable x and monadic expressions E, $M1$, $M2$
such that • x does not occur in $M1$
• E and $M1$ do not change any variables in the state which the other uses.

$$\begin{array}{lcl} \textbf{do}\ M1 & == & \textbf{do}\ x \leftarrow E \\ \quad x \leftarrow E & & \quad M1 \\ \quad M2 & & \quad M2 \end{array}$$

Such lemmas can be specialised for particular monads and monadic functions, as in the following example for the *IOPST* monad.

Rule 41 (Move *create_var*s to the top)

For all $M1, M2$ IOPST expressions and x_v a variable name such that x_v does not occur in $M1$.

$$\begin{array}{lcl} \textbf{do}\ M1 & == & \textbf{do}\ x\_v \leftarrow \mathit{create\_var}\ X \\ \quad x\_v \leftarrow \mathit{create\_var}\ X & & \quad M1 \\ \quad M2 & & \quad M2 \end{array}$$

A.5 Transformation rules for particular APMs

Many of the transformation rules given in this thesis are specific to particular APMs and can only be used to transform within or between those given APMs. This section collects these together for ease of reference. Most of these were initially given in Chapter 3.

A.5.1 Parallel and sequential APMs

The following two rules can be used to transform programs to use the parallel and sequential APMs. They use the functions *changeseq* and *changepar* given in Section 3.4.

Rule 42 (Make functions sequential)

For any expression e.

$$e \Rightarrow \mathit{changeseq}\ e$$

Rule 43 (Make functions parallel)

For any expression e such that e does not involve any data nested inside items of type $\mathit{FinSeq}\ \alpha$.

$$e \Rightarrow \mathit{changepar}\ e$$

A.5.2 APMs based on IOPST

IOPST is a specialised monad used to model and manipulate the state of a parallel system. It is described in Section 2.8. Several of the APMs in this thesis use this monad and Section 3.6 shows how various aspects of it can be introduced into and manipulated within a program, using the following rules. Other lemmas and rules, such as Lemma 41 above, also deal with the *IOPST* monad.

Rule 44 (Introduce main)

For all $prog :: T_1 \to T_2 \to \ldots \to T_n \to T'$, $p_i :: T_i$ $(i = 1, \ldots, n)$, $F :: T'$.

```
    pst_putStr (show (prog p_1 ... p_n))
      where
        prog x_1 ... x_n = F
 ⇒

    main p = do let enter_1 = ...
                     ...
                     enter_n = ...
                let prog x_1 ... x_n = F
                q_1 ← enter_1
                ...
                q_n ← enter_n
                result ← prog q_1 ... q_n
                pst_putStr (show result)
      where
        enter_i = the appropriate input function for p_i.
                  It may have parameters.
```

This rule introduces the key function, *main*, to an *IOPST* program.

Rule 45 (Set up the parallel system)

For all IOPST expressions E.

```
    main p = E        ⇒        main p = do start p
                                           E
```

If variables are to be used in an *IOPST* program, the model of the parallel system must be set up using *start* as in this rule.

Rule 46 (New variable)

For all x, E :: *ParFinSeq* T for some type T,
there exists $T0 :: T$ which can be used for initialisation and x_v :: *VarFn* T such that

```
    x ← return E   ⇒   x_v ← create_var E
                       x ← retrieve x_v
               or  ⇒   x_v ← create_var (repeat_P (T0 :: T))
                       store x_v E
                       x ← retrieve x_v
```

This rule allows new variables to be created and then used within an *IOPST* program.

Rule 47 (Remove an extra variable)

For all x_v, y_v :: *VarFn* T, V, $V2$ :: *ParFinSeq* T (for some type T), *Fs*, *Gs* IOPST expressions such that

- x and x_v don't occur in *Gs*,
- x, y refer to the values obtained from x_v,y_v respectively.

```
    x_v ← create_var V     ⇒    y_v ← create_var V
    Fs                          Fs[y_v/x_v, y/x]
    y_v ← create_var V2         Gs
    Gs
```

This rule doesn't introduce a new feature to a program, but allows the existing code to be manipulated, removing a variable which isn't necessary.

Rule 48 (Add a global variable)

For all $x, E, T0 :: T$ (for any type T), $p :: Int^+$, $F, F2$ IOPST expressions and Fs local function definitions.

$$\begin{array}{l} \begin{array}{lcl} x & = & E \\ main\ p & = & \mathbf{do}\ F \\ & & \quad \mathbf{let}\ Fs \quad \text{— local function definitions} \\ & & \quad F2 \end{array} \\ \Rightarrow \\ \begin{array}{lcl} main\ p & = & \mathbf{do}\ F \\ & & \quad x\_v \leftarrow create\_var\ (repeat_P\ (T0 :: T)) \\ & & \quad \mathbf{let}\ Fs \quad \text{— local function definitions} \\ & & \quad x \leftarrow return\ E \\ & & \quad store\ x\_v\ x \\ & & \quad x \leftarrow retrieve\ x\_v \\ & & \quad F2 \end{array} \end{array}$$

Sometimes it is useful to have global variables in an *IOPST* program. This rule introduces such a variable, declaring it before the main code, $F2$, so that it can be used within that code.

Rule 49 (Duplicate a value across the processors)

For $T \neq D\alpha$ for $D = ParFinSeq$ or any data distribution type.
For all $E, x :: T$ (on the left hand side)

$$x = E \quad \Rightarrow \quad x = repeat_P\ E$$

and
For all $E :: IOPST\ T$, $x, x' :: T$ (on the left hand side).

$$\begin{array}{lcl} x \leftarrow E & \Rightarrow & x' \leftarrow E \\ & & x \leftarrow return\ (repeat_P\ x') \end{array}$$

As described in Section 3.7.3, it is sometimes necessary to have a copy of a variable in each processor. This rule accomplishes this.

Rule 50 (Reference parameters)

For all f, x, y, T of appropriate types.

$$\begin{array}{lcl} f :: \ldots \rightarrow ParFinSeq\ T \rightarrow \ldots & \Rightarrow & f :: \ldots \rightarrow VarFn\ T \rightarrow \ldots \\ f \ldots x \ldots = & & f \ldots x\_v \ldots = \\ \quad \mathbf{do} \ldots & & \quad \mathbf{do}\ x \leftarrow retrieve\ x\_v \\ & & \qquad \ldots \\ E[f \ldots y \ldots] & & E[f \ldots y\_v \ldots] \end{array}$$

This rule is used to change a parameter to be passed by reference instead of by value.

Rule 51 (Returning a value from a function)

For all $x :: T$, f, ps, E of appropriate types and $x\_v :: VarFn\ T$ such that x_v is either a new variable (create it using *create_var* before f is called) or the variable corresponding to x.

$$\begin{array}{lcl} x \leftarrow f\ ps & \Rightarrow & f\ ps\ x\_v \\ & & x \leftarrow retrieve\ x\_v \end{array}$$

and

$$\begin{array}{lcl} f\ ps = \mathbf{do} \ldots & \Rightarrow & f\ ps\ x\_v = \mathbf{do} \ldots \\ \qquad return\ E & & \qquad store\ x\_v\ E \end{array}$$

This returns a value from a function by storing it in a reference parameter instead of returning it directly.

A.5.3 The MPI APM

The MPI APM is described in Section 2.10 and the rules and lemmas that manipulate programs using it are described in Section 3.8. These are also given below for ease of reference.

Rule 52 (*mpi_comm_size*)

For all $x\_v :: VarFn\ \alpha$, $xs :: ParFinSeq\ \alpha$ (any α) such that
xs hasn't been shortened using, for example, *take* or *filter* (the initial data structure on which the program operates is a good choice for xs).

$$store\ x\_v\ (size_P\ xs) \quad \Rightarrow \quad mpi\_comm\_size\ x\_v$$

Rule 53 (repeat/broadcast)

For all $val :: \alpha$, $var\_v :: VarFn\ \alpha$.

$$\begin{array}{lll} store\ var\_v\ (repeat_P\ val) & \Rightarrow & \mathbf{do}\ store_{indiv}\ 0\ var\_v\ val \\ & & \quad mpi\_bcast\_simple\ var\_v\ T\ 0 \end{array}$$

Rule 54 (scatter)

For all $xs :: [\alpha]$, $var\_v :: VarFn\ \alpha$.

$$\begin{array}{lll} store\ var\_v\ (list2parfs\ xs) & \Rightarrow & tmp\_v \leftarrow create\_var\ (repeat_P\ (T0 :: T)) \\ & & store_{indiv}\ i\ tmp\_v\ (list2seqfs\ xs) \\ & & mpi\_scatter\ tmp\_v\ 1\ T\ tmp\_v\ T\ i \\ & & tmp \leftarrow retrieve\ tmp\_v \\ & & new\_vals \leftarrow return\ (map_P\ heads_S\ tmp) \\ & & store\ var\_v\ new\_vals \end{array}$$

where
- $T = SeqFinSeq\alpha$
- $T0$ is any value of type T

Rule 55 (reverse)

For all $xs :: ParFinSeq\ \alpha$, $xs\_v :: VarFn\ \alpha$ (any α).

$$\begin{array}{lll} xs \leftarrow retrieve\ xs\_v & \Rightarrow & mpi\_jointpt2pt\ xs\_v\ x\_v\ T\ (\lambda\ i \rightarrow (p-1-i))\ (\lambda\ i \rightarrow (p-1-i)) \\ store\ x\_v\ (reverse_P\ xs) & & \end{array}$$

where
- p is the number of processors. It can be set using: $\mathbf{do}\ p \leftarrow retrieve\ p\_v$
  $\qquad p \leftarrow return\ (head_P\ p)$
- T is the *ItemType* corresponding to α (*ItemType* is an enumerated Haskell type described in Section 2.8.2.)

A.5.4 Individual level

Some of the later stages in the methodology view the parallel system from the viewpoint of a single individual processor, rather than of the whole system. Rules and lemmas dealing with these stages are given in Sections 3.9 and 3.10.3, and are also collected below.

Rule 56 (Transformation to the individual level)

This rule is really a set of rules that can be applied to different fragments of program. They are explained in more depth in Section 3.9

- $\mathit{mpi\_init}\ p \Rightarrow \mathit{mpi\_init}$ (for any $p :: \mathit{Int}^+$)
- $\mathit{get\_size} \Rightarrow \mathit{mpi\_comm\_size}$
  $\mathit{get\_pid} \Rightarrow \mathit{mpi\_comm\_rank}$
- For all $x\_v :: \mathit{VarFn}\ T,\ v :: T$.
  $x\_v \leftarrow \mathit{create\_var}\ (\mathit{repeat}_P\ v\ ::\ \mathit{ParFinSeq}\ T) \quad \Rightarrow \quad x\_v \leftarrow \mathit{create\_var}\ (v :: T)$
- Individual stores and retrieves using store_{indiv} and $\mathit{retrieve}_{indiv}$ are replaced by **if** expressions on the processor id.
- Collective communication functions are not changed.
- For all $s\_v,\ r\_v :: \mathit{VarFn}\ T$ (any type T), $\mathit{sendfn},\ \mathit{recvfn} :: \mathit{Int}^+ \rightarrow \mathit{Int}^+,\ \mathit{source},\ \mathit{dest} :: \mathit{Int}^+$.
  $\mathit{mpi\_jointpt2pt}\ s\_v\ r\_v\ T\ \mathit{sendfn}\ \mathit{recvfn} \quad \Rightarrow \quad$ **do** $\mathit{mpi\_send}\ s\_v\ T\ (\mathit{sendfn}\ \mathit{pid})$
  $\qquad \mathit{mpi\_recv}\ r\_v\ T\ (\mathit{recvfn}\ \mathit{pid})$
- and
  $\mathit{mpi\_spt2pt}\ s\_v\ T\ \mathit{source}\ r\_v\ T\ \mathit{dest} \quad \Rightarrow \quad$ **if** $(\mathit{pid} == \mathit{source})$ **then** $\mathit{mpi\_send}\ s\_v\ T\ \mathit{dest}$
  else if $(\mathit{pid} == \mathit{dest})$ **then** $\mathit{mpi\_recv}\ r\_v\ T\ \mathit{source}$
  else $\mathit{return}\ ()$
- For all $x_i :: T_i,\ f :: T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$.
  $\mathit{mapn}_P\ f\ x_1 \ldots x_n \Rightarrow f\ x_1 \ldots x_n$

The following rules manipulate functions and data that use the Cyclic distribution.

Rule 57 (Individual level cyclic state functions)

For all $j :: \mathit{Int}^+$, pid containing the processor id and p the number of processors.

$\mathit{retrieve}_{Cyclicindiv}\ j$
$\Rightarrow$
if$(\mathit{pid} == j\ `\mathit{mod}`\ p)$ **then** $\mathit{retrieve}_{indiv}\ (j\ `\mathit{div}`\ p)$
else $\mathit{return}\ ()$

$\mathit{store}_{Cyclicindiv}\ j$
$\Rightarrow$
if$(\mathit{pid} == j\ `\mathit{mod}`\ p)$ **then** $\mathit{store}_{indiv}\ (j\ `\mathit{div}`\ p)$
else $\mathit{return}\ ()$

Rule 58 (Cyclic point to point send)

For all $var\_v :: VarFn\ T1$, $var2\_v :: VarFn\ T2$, $source$, $dest :: Int^{+}$ (for some types $T1$, $T2$).

$$\begin{array}{l} mpi\_spt2pt_{Cyclicindiv}\ var\_v\ T1\ source\ var2\_v\ T2\ dest \\ \quad \Rightarrow \\ \textbf{if}(pid\ ==\ source`mod`p) \\ \quad \textbf{then}\ mpi\_send'\ var\_v\ T1\ (dest`mod`p)\ (source`div`p) \\ \textbf{else if}\ (pid\ ==\ dest`mod`p) \\ \quad \textbf{then}\ mpi\_recv'\ var2\_v\ T2\ (source`mod`p)\ (dest`div`p) \\ \textbf{else}\ return\ () \end{array}$$

Rule 59 (Replacement of cyclic state functions)

For any α, s, val, $vals$, i, var_v of appropriate types and new name old_vals.

- $VarFn_{Cyclic}\ \alpha\ \Rightarrow\ VarFn(SeqFinSeq\ \alpha)$
- $create\_var_{Cyclic}\ s\ \Rightarrow\ create\_var\ s$
- $val\ \leftarrow\ retrieve_{indiv}\ i\ var\_v$

  $\Rightarrow$

  $old\_vals\ \leftarrow\ retrieve\ var\_v$

  $val\ \leftarrow\ return\ (vals\ !!_S\ i)$
- $store_{indiv}\ i\ var\_v\ val$

  $\Rightarrow$

  $old\_vals\ \leftarrow\ retrieve\ var\_v$

  $store\ var\_v\ (replace_S\ old\_vals\ i\ val)$

  where $replace_S\ xs\ i\ x$ returns xs with its ith element set to x and its other elements as before
- $val\ \leftarrow\ retrieve_{Cyclic}\ var\_v\ \Rightarrow\ val\ \leftarrow\ retrieve\ var\_v$
- $store_{Cyclic}\ var\_v\ vals\ \Rightarrow\ store\ var\_v\ vals$

Rule 60 (Replacement of $mpi\_scatter_{Cyclic}$)

For any $sendbuf :: VarFn\ T$, $recvbuf$, $sendtype$, $recvtype$, $root$ of appropriate types
and new names, p, n, pid, tmp_v, etc.
p_v storing the number of processors, $matrix\_size\_v$ the matrix size and pid_v the processor id.

$$mpi\_scatter_{Cyclic}\ sendbuf\ sendtype\ recvbuf\ recvtype\ root$$

$$\Rightarrow$$

```
  — access global variables
$p \leftarrow$ retrieve $p\_v$
$n \leftarrow$ retrieve $matrix\_size\_v$
$pid \leftarrow$ retrieve $pid\_v$
  — create new variables
$tmp\_v \leftarrow create\_var\ (X :: T)$
$sizes\_v \leftarrow create\_var\ (repeats_S\ (0 :: Int))$
$displs\_v \leftarrow create\_var\ (repeats_S\ (0 :: Int))$
  — set up variables for sending the values
if $(pid == root)$ then
    do  — set up tmp_v with the cyclic layout
      $scattervals' \leftarrow retrieve\ sendbuf$
      $scattervals \leftarrow return\ (makecyclic\ p(seqfstoList\ scattervals'))$
      $store\ tmp\_v\ (concat_S\ scattervals)$
        — set up size and displs
        — the first few procs may have one more item than the others
      $store\ sizes\_v\ (cat_S\ (replicate_S\ (n`mod`p)\ (n`div`p + 1))$
                        $(replicate_S\ (p - (n`mod`p))\ (n`div`p)))$
      $sizes \leftarrow retrieve\ sizes\_v$
      let $displvals = (0 : [displvals\ !!_S\ i + (sizes\ !!_S\ i)$
                                $\mid i \leftarrow [0..n-2]])$
      $store\ displs\_v\ (toSeqFinSeq\ displvals)$
else $return\ ()$
  — send the values
$mpi\_scatterv\ tmp\_v\ sizes\_v\ displs\_v\ sendtype\ recvbuf\ recvtype\ root$
```

A.5.5 C+MPI

Section 3.11.2 discusses how the program can be finally transformed into C+MPI. It gives the following main transformation rule. Several smaller transformation rules are also given in Table 3.2.

Rule 61 (C+MPI top level)

```
main = do start
          variable declarations
          local function definitions
          code

⇒   variable declarations
    local function declarations

    main(int arc, char *argv[])
      {
        int errcode;

        errcode = MPI_Init (&argc, &argv);
        code
        errcode = MPI_Finalize ();
      }

    local function definitions
```

Once the program is in C+MPI, some transformations can still be done. The following are a few examples:

Rule 62 (Remove initialisations)

For all types T, variables x and values y of type T.

```
                T x = y;          ⇒          T x;
```

Provided • the value of x is updated before the first time it is used.

Rule 63 (take rule)

For any arrays `xs`, `res`, positive integers `m`, `res_size` and C statements `Fs` such that

- `size` is the number of elements in `xs` and
- `Fs` never accesses `res[i]`, `i` $\geq$ `res_size`.

```
take(xs,size,m,res,& res_size);    ⇒    res_size = min(m,size);
Fs;                                     Fs[xs/res];
```

Rule 64 (drop rule)

For any arrays `xs`, `res`, positive integers `size`, `m`, `res_size` and C statements `Fs` such that

- `size` is the number of elements in `xs` and
- `Fs` never accesses `res[i]`, `i` $\geq$ `res_size`.

```
drop(xs,size,m,res,& res_size);    ⇒    res_size = size - min(m,size);
Fs;                                     Fs[xs+min(m,size)/res];
```

A.6 Proofs of selected lemmas

This section presents a few representative examples of proofs of lemmas in this thesis.

Lemma 8

```
                            {by A.3 as y is fresh}
let Fs              ==      let Fs
    x = f E                     let y = E
in                              in
E2                              x = f y
                            in E2

                    ==      {by A.5 as y is fresh}
                            let Fs
                                y = E
                                x = f y
                            in E2
```

Lemma 14

$\#xs \geq 1 \Rightarrow xs = y : ys$ for some ys

```
foldl1 f xs = foldl1 f (y:ys)
            = {definition of foldl1}
              foldl f y ys
            = {a is a left unit of f}
              foldl f (f a y) ys
            = {definition of foldl}
              foldl f a (y:ys)
            = foldl f a xs
```

Lemma 21

This is an example a proof by structural induction.

$\underline{xs = []}$

`length (take m []) = length [] = 0 = min(0,m)`

$\underline{xs = (x : xs)}$

This has two cases, depending on m:

$\underline{m = 0}$

```
length (take m (x:xs)) = length [] = 0
                       = min(length (x:xs), 0)
                       = min(length (x:xs), m)
```

$\underline{m > 0}$

```
length (take m (x:xs)) = length (x: take (m-1) xs)
                       = 1 + length (take (m-1) xs)
                       = {by induction hyp}
                         1 + min (length xs, m-1)
                       = min (length (x:xs), m)
```

Hence, the result holds, by structural induction.

Lemma 27

This proof that *transpose* is its own inverse uses three subsidiary lemmas 28 to 30. These lemmas were produced for use in this proof although it is also likely that they would be useful for other proofs involving *transpose*. This proof illustrates the use of subsidiary lemmas and structural induction.

Lemma 28 is proved as follows:

As $\#xss > 0$ and $\#(xss!!0) > 0$, xss can be written as $(y : ys) : yss$. Then:

```
transpose xss = {By the definition of transpose}
                (y:map head yss): transpose (ys: map tail yss)
              = {As (y:ys) = head xss, y = head(head xss),
                 ys = tail(head xss) and yss = tail xss}
                (head(head xss):map head (tail xss)):
                    transpose(tail(head xss): map tail(tail xss))
              = {By definition of map}
                (map head (head xss: tail xss)):
                    transpose(map tail (head xss: tail xss))
              = {As (head xs:tail xs) = xs}
                (map head xss): transpose(map tail xss)
```

Lemmas 29 and 30 are proved by structural induction on xs, and the latter proof also uses Lemma 28.

The main lemma 27 can then be proved by structural induction, with three cases, since the definition of *transpose* (in Section 3.3.4) has three cases. In the following *transpose* is sometimes written `tr` for conciseness.

$\underline{xss = []}$

$transpose(transpose\ []) = transpose\ [] = []$

$\underline{xss = ([] : yss)}$

$\#(xss!!i) = \#(xss!!0) = \#[] = 0 \Rightarrow xss = []$

So this is the same as the previous case.

$\underline{xss = ((x : xs) : xss)}$

```
transpose(transpose ((x:xs):xss))
        = {apply definition of transpose to inner function}
          tr((x:map head xss): tr(xs: map tail xss))
        = {apply def of transpose to outer function}
          (x:map head (tr(xs: map tail xss))):
            tr((map head xss): map tail (tr(xs:map tail xss)))
        = {By  Lemma A.25 applied to the first term
           and Lemma A.26 applied to the last term}
          (x:xs): tr((map head xss): tr(map tail xss))
        = {By  Lemma A.24 applied right to left
           to the second term}
          (x:xs): tr(tr xss)
        = {By induction hypothesis}
          (x:xs):xss
```

Hence, the result holds, by induction.

Lemma 38

This is an example of a basic monadic proof.

```
                                {by Lemma A.11}
return (let x = E1      ==      let x = E1
        in                      in
        E2)                     return E2

                        ==      {by Lemma A.30}
                                do x <- return E1
                                   return E2
```

Lemma 39

This is an example of a state monad proof. To do it, more notation is helpful. $\mathcal{M}[E]\rho S$ gives the meaning of the expression E in an environment ρ, which gives the values of variables bound using functions and λ notation.

For example, after $\mathbf{do}\, x \leftarrow \mathit{return}\, 1$, $\rho = \{< x, 1 >, < y, 2 >\}$
$y \leftarrow \mathit{return}\, 2$

In addition, S gives the value of the State to be input to E.

The notation $\rho[x := E]$ means ρ with the value of x bound to E.

Then the proof looks like this:

$$\begin{array}{lcl} \mathcal{M}[\mathbf{do}\, x \leftarrow E]\,\rho\, S & = & \{\textit{since E does not affect S}\} \\ \quad M & & \mathcal{M}[M]\rho[x := \mathcal{M}[E]\rho\, S]\, S \\ & = & \{\textit{since M does not use x until its value is reset}\} \\ & & \mathcal{M}[M]\rho\, S \end{array}$$

Therefore the meaning of the two expressions is the same, and they are equivalent.

Index

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–641, 1978.

[BCC$^+$97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. The Society of Industrial and Applied Mathematics, May 1997. Also available from: http://www.netlib.org/scalapack/slug.

[BDO$^+$95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[BF94] Alexander Bunkenburg and Sharon Flynn. Expression refinement: Deriving Bresenham's algorithm. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 1–17. Springer, 1994.

[BG96] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[BGA90] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to Fortran 90*. Intertext Publications, McGraw-Hill Book Company, 1990.

[BGP93] E. A. Boiten, A. M. Geerling, and H. A. Partsch. Transformational development of (parallel) programs using skeletons. In *Computing Science in the Netherlands*, pages 97–108, 1993.

[Bir89] Richard Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computer Science*, volume 55 of *NATO ASI Series F*, pages 151–216. Springer, 1989.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, January 1998.

[BKT00] C. Baker-Finch, D.J. King, and P.W. Trinder. An Operational Semantics for Parallel Lazy Evaluation. In *International Conference on Functional Programming (ICFP'00)*, Montreal, Canada, 18–20 September, 2000.

[BL97] Silvia Breitinger and Rita Loogen. Channel structures in the parallel functional language Eden. In *Glasgow Workshop on Functional Programming 1997*, 1997.

[Ble95] Guy E. Blelloch. NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, September 1995.

[Ble96] Guy Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996. Also available at http://www.cs.cmu.edu/~scandal/cacm.html.

[BLOMPM96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Eden — the paradise of functional concurrent programming. In *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*. Springer, 1996.

[Bra92] T. A. Bratvold. Determining useful parallelism in higher order functions. In *Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages*, 1992.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[CBS97] Wolfram Kahl Chris Brink and Gunther Schmidt, editors. *Relational Methods in Computer Science.* Advances in Computing 1997. Springer-Verlag, Wien, 1997.

[CFH+94] M. Clint, S. Fitzpatrick, T. J. Harmer, P. L. Kilpatrick, and J. M. Boyle. A family of data-parallel derivations. In *High-performance Computing and Networking. International Conference and Exhibition Proceedings Vol. 2: Networking and Tools*, volume 797 of *Lecture Notes in Computer Science*. Springer, 1994.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CHK+93] Stuart Cox, Shell-Ying Huang, Paul Kelly, Junxian Liu, and Frank Taylor. Program transformations for static process networks. *ACM SIGPLAN Notices*, Jan 1993.

[CKLP01] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal – nested data parallelism in Haskell. In *Seventh International Euro-Par Conference*, volume 2150 of *LNCS*, pages 524–534. Springer, 2001.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company/MIT Press, 1990.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, 1989.

[DFH+93] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In *PARLE'93: Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer, June 1993.

[DGT93] J. Darlington, M. Ghanem, and H.W. To. Structured parallel programming. In *Massively Parallel Programming Models Conference*, Berlin, Sept 1993.

[DOSW96] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, July 1996.

[Dwy95] Barry Dwyer. Libra: A lazy interpreter of binary relational algebra. Technical Report 95-10, Department of Computer Science, University of Adelaide, 1995.

[EGL98] Nils Ellmenreich, Martin Griebl, and Christian Lengauer. Applicability of the polytope model to functional programs. In Herbert Kuchen, editor, *Proc. 7th Int. Workshop on Functional and Logic Programming.* Institut für Wirtschaftsinformatik, Westf. Wilhelms-Universität Münster, April 1998.

[ELG99] Nils Ellmenreich, Christian Lengauer, and Martin Griebl. Application of the polytope model to functional programs. In Larry Carter and Jeanne Ferrante, editors, *Languages and Compilers for Parallel Computing, 12th International Workshop, LCPC'99*, volume 1863 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 1999.

[FCHK96] Stephen Fitzpatrick, M. Clint, T.J. Harmer, and P. Kilpatrick. The tailoring of abstract functional specifications of numerical algorithms for sparse data structures through automated program derivation and transformation. *The Computer Journal*, 39(2):145–168, 1996.

[FHB94] Stephen Fitzpatrick, T.J. Harmer, and J.M. Boyle. Deriving efficient parallel implementations of algorithms operating on general sparse matrices using automatic program transformation. In *Parallel Processing: CONPAR94 - VAPP IV*, volume 854 of *LNCS*, pages 148–159. Springer, September 1994.

[FK94] Ian Foster and Carl Kesselman. Language constructs and runtime systems for compositional parallel programming. In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 - VAPP VI*, volume 854 of *LNCS*, pages 5–16. Springer-Verlag, September 1994. Invited talk.

[FM00] Pascal Fradet and Julien Mallet. Compilation of a specialized functional language for massively parallel computers. *Journal of Functional Programming*, 10(6):561–605, November 2000.

[GA00] Attila Gürsoy and Murat Atun. Neighbourhood preserving load balancing: A self-organizing approach. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 234–241. Springer, 2000.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[GHC] The GHC Team. *Glasgow Haskell Compiler Users' Guide, Version 5.00.* Available at http://www.haskell.org/ghc/documentation.html.

[GL95] Sergei Gorlatch and Christian Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

[GL00] Sergei Gorlatch and Christian Lengauer. Abstraction and performance in the design of parallel programs: an overview of the SAT approach. *Acta Informatica*, 36(9–10):761–803, May 2000.

[GO99] Joy Goodman and John O'Donnell. Nondeterminism in the APM methodology. In Phil Trinder and Greg Michaelson, editors, *Proceedings of SFP '99: First Scottish Functional Programming Workshop*, pages 115–124. Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, September 1999. Technical Report RM/99/9.

[GO01] Joy Goodman and John O'Donnell. Introduction of static load balancing in incremental parallel programming. In *Seventh International Euro-Par Conference*, volume 2150 of *LNCS*. Springer, 2001.

[Goo98] Joy Goodman. A methodology for the derivation of parallel programs. Workshop UMDITR03, Departamento de Informática, Universidade do Minho, September 1998.

[Goo01a] Joy Goodman. Incremental program transformation using abstract parallel machines: Website for the PhD thesis. http://www.dcs.gla.ac.uk/~joy/research/thesis, 2001.

[Goo01b] Joy Goodman. Introduction of pipelining optimisations into Gaussian elimination. In *Proceedings of the 3rd Scottish Functional Programming Workshop*, pages 21–32, August 2001.

[Gor94] Andrew Gordon. A tutorial on co-induction and functional programming. In *Glasgow functional programming workshop*, Workshops in Computing, pages 78–95. Springer, 1994.

[Gor96] Sergei Gorlatch. Stages and transformations in parallel programming. In *Abstract Machine Models*, pages 147–161. IOS Press, 1996.

[GOR98] Joy Goodman, John O'Donnell, and Gudula Rünger. Refinement transformation using abstract parallel machines. In *Glasgow Functional Programming Group Workshop 1998*, September 1998. Slightly updated version available at http://www.dcs.gla.ac.uk/~joy/research.

[GP99] Sergei Gorlatch and Susanna Pelagatti. A transformational framework for skeletal programs: Overview and case study. In Jose Rohlim et al., editors, *Parallel and Distributed Processing. IPPS/SPDP'99 Workshops Proceedings*, Lecture Notes in Computer Science 1586, pages 123–137, 1999.

[GSSW98] C. Grelck, S. B. Scholz, A. Sievers, and H. Wolf. *Syntax of SAC v0.8*, November 1998.

[GY92] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, December 1992.

[Ham94] Kevin Hammond. Parallel functional programming: An introduction. In *First Intl. Symposium on Parallel Symbolic Computation*. World Scientific, September 1994.

[HC00] Yasushi Hayashi and Murray Cole. BSP-based cost analysis of skeletal programs. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, chapter 3, pages 20–28. Intellect, 2000.

[Her00] Christoph Armin Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, Universität Passau, 2000. Published by Logos-Verlag, Berlin: ISBN 3-89722-556-5.

[HGL$^+$93] Erik Hagersten, Mats Grindal, Anders Landin, Ashley Saulsbury, Bengt Werner, and Seif Haridi. Simulating the data diffusion machine. In *PARLE'93: Parallel Architectures and Languages Europe*, volume 964 of *Lecture Notes in Computer Science*, pages 24–41. Springer, June 1993.

[HL00] Christoph A. Herrmann and Christian Lengauer. The HDC compiler project. In Alain Darte, Georges-André Silber, and Yves Robert, editors, *Proc. Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 239–254. LIP, ENS Lyon, 2000.

[HLG$^+$99] Christoph Armin Herrmann, Christian Lengauer, Robert Günz, Jan Laitenberger, and Christian Schaller. A compiler for HDC. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Universität Passau, May 1999.

[HMS$^+$97] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: The BSP programming library. Technical Report PRG-TR-29-9, Oxford University Computing Laboratory, May 1997.

[HO89] John Hughes and John O'Donnell. Expressing and reasoning about non-deterministic functional programs. In *Functional Programming, Glasgow 1989*, Springer-Verlag Workshops in Computing (1990), pages 308–328. Springer-Verlag, 1989.

[HOL] Tutorial for the HOL theorem proving system. Available on the internet at URL http://lal.cs.byu.edu/lal/holdoc/tutorial.html.

[HPF97] Digital Equipment Corporation, Maynard, Massachusetts. *Digital High Performance Fortran 90 HPF and PSE Manual*, January 1997. Order number AA-Q62LC-TE.

[HSJ95] Samuel P. Harbison and Guy L. Steele Jr. *C, A Reference Manual.* Prentice-Hall, fourth edition, 1995.

[Inc88] D.C. Ince. *An introduction to Discrete Mathematics and Formal System Specification.* Oxford Applied Mathematics and Computing Science Series. Oxford University Press, 1988.

[JMC96] He Jifeng, Quentin Miller, and Lei Chen. Algebraic laws for BSP programming. In *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 359–368. Springer, 1996.

[Kam99] Ernic Kamerich. *A Guide to Maple.* Springer, 1999.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1988.

[KT99] Paul Kelly and Frank Taylor. Coordination languages. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 14, pages 305–321. Springer, 1999.

[Loi96] Hans-Wolfgang Loidl. *GranSim User's Guide Version 0.03.* Glasgow University, July 1996. Available from: http://www.dcs.gla.ac.uk/fp/software/gransim/user_toc.html.

[LPJ95] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8:293–341, 1995.

[LS93] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*, volume 768 of *LNCS*, pages 96–114. Springer, August 1993.

[LTB01] H.-W. Loidl, P.W. Trinder, and C. Butz. Tuning task granularity and data locality of data parallel GPH programs. *Parallel Processing Letters*, 2001. Selected papers from HLPP2001 — International Workshop on High-level Parallel Programming and Applications, Universite d'Orleans, France, March 26–27, 2001.

[LV90] L. D. J. C. Loyens and J. G. G. Van De Vorst. Two small parallel programming exercises. *Science of Computer Programming*, 15(2–3):159–69, 1990.

[MC97] John Merlin and Barbara Chapman. *High Performance Fortran 2.0.* VCPC (European Center for Parallel Computing at Vienna, University of Vienna, Liechtenstein Strasse 22, A-1090 Vienna, Austria, November 1997.

[MH95] John Merlin and Anthony Hey. An introduction to High Performance Fortran. *Sci. Prog.*, 4:87–113, November 1995.

[Mic89] Greg Michaelson. *An Introduction to Functional Programming Through the Lambda Calculus.* Addison Wesley, 1989.

[MPI97] University of Tennessee, Knoxville, Tennessee. *MPI-2: Extensions to the Message-Passing Interface*, 1997. Available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[NAH$^+$95] R.S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J. Maessen, and Y. Zhou. pH language reference manual. Technical Report CSG-Memo-369, Computation Structures Group, MIT Lab for Computer Science, January 1995.

[Nip01] Tobias Nipkow. Isabelle HOL: The tutorial. To be published by Springer, 2001. Available at http://www.in.tum.de/~nipkow/pubs/tutorial.html, March 2001.

[O'D94] John O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.

[O'D01] John O'Donnell. The collective and individual semantics in functional SPMD programming. In *IFL 2001*, 2001. Submitted for publication.

[OR94] John O'Donnell and Gudula Rünger. A case study in parallel program derivation: the heat equation algorithm. In *Glasgow functional programming workshop*, Workshops in Computing, pages 167–183. Springer, 1994.

[OR95] John O'Donnell and Gudula Rünger. An explanatory formal derivation of a parallel binary addition circuit. Technical Report Computing Science Department TR-1995-19, University of Glasgow, 1995.

[OR97] John O'Donnell and Gudula Rünger. 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, 1997. Preliminary, shorter version of [OR00].

[OR00] John O'Donnell and Gudula Rünger. Abstract parallel machines. *Computers and Artificial Intelligence*, 19:105–129, 2000.

[ORR01] John O'Donnell, Thomas Rauber, and Gudula Rünger. Cost hierarchies for abstract parallel machines. In *13th International Workshop on Languages and Compilers for Parallel Computing*, LNCS. Springer-Verlag, 2001. To appear.

[Pep93] Peter Pepper. Deductive derivation of parallel programs. In Robert Paige, John Reif, and Ralph Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, chapter 1, pages 1–53. Kluwer Academic Publishers, 1993.

[PJ01] Simon Peyton Jones. Tackling the awkward squad: monadic I/O, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001. Presented at the 2000 Marktoberdorf Summer School. Also available at http://www.haskell.org/bookshelf/.

[PJCSH87] Simon Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a high-performance architecture for parallel graph reduction. In *FPCA'87 — Intl. Conf. on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 98–112, Portland, Oregon, September 1987. Springer Verlag.

[PJea99] Simon Peyton Jones and John Hughes et al. Report on the programming language Haskell 98. Available on the internet at http://www.haskell.org/onlinereport, 1999.

[PJL92] Simon Peyton Jones and David Lester. *Implementing Functional Languages.* Prentice Hall, 1992.

[PJL94] Simon Peyton Jones and John Launchbury. Lazy functional state threads. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'94)*, pages 24–35, June 1994.

[PJS94] Simon Peyton Jones and André Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer, 1994.

[PJW91] Simon Peyton Jones and Philip Wadler. The static semantics of Haskell. Unpublished Technical Report, University of Glasgow. Available on the internet at http://www.haskell.org/definition, 1991.

[PJW93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[Por13] Noah Porter, editor. *1913 Webster's Revised Unabridged Dictionary*. G & C. Merriam Co., 1913.

[PPRS00] Cristóbal Pareja, Ricardo Peña, Fernando Rubio, and Clara Segura. Optimizing Eden by transformation. In Stephen Gilmore, editor, *Trends in Functional Programming*, volume 2, pages 13–26, St Andrews, 2000. Intellect.

[Rév88] György E. Révész. *Lambda-Calculus, Combinators and Functional Programming*. Number 4 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[Rob90] Yves Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*. Manchester University Press and Halsted Press, 1990.

[RR95] Thomas Rauber and Gudula Rünger. Deriving structured parallel implementations for numerical methods. *The Euromicro Journal*, 41:589–608, 1995.

[RR96] Thomas Rauber and Gudula Rünger. The compiler TwoL for the design of parallel implementations. In *Proceedings of the 4th International Conference on Parallel Architecture and Compilation Techniques*, pages 292–301. IEEE Computer Society Press, 1996.

[RR00] Thomas Rauber and Gudula Rünger. Data distributions for task-parallel programs. In *Proc. of the 8th Workshop on Compilers for Parallel Computers*, 2000.

[SDPZ98] David Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising data-parallel programs using the BSP cost model. In *4th International Euro-Par Conference*, volume 1470 of *Lecture Notes in Computer Science*, pages 698–703. Springer, 1998.

[SHM97] David Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[SKGF01] Rizos Sakellariou, John Keane, John Gurd, and Len Freeman, editors. *Seventh International Euro-Par Conference*, volume 2150 of *LNCS*. Springer, 2001.

[Ski99] David Skillicorn. Cost modelling. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 8, pages 207–218. Springer, 1999.

[ST98] David Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, pages 123–169, June 1998.

[Sun90] Vaidy Sunderam. PVM:a framework for parallel distributed computing. *Concurrency: Practice & Experience*, December 1990.

[Tal94] Domenico Talia. Parallel logic programming systems on multicomputers. *Journal of Programming Languages*, 2(1):77–87, March 1994.

[THLPJ98] P. W. Trinder, Kevin Hammond, H.-W. Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[THMJP96] P.W. Trinder, K. Hammond, J.S. Mattson Jr, and A.S. Partridge. GUM: a portable parallel implementation of Haskell. In *Proceedings of Programming Language Design and Implementation, Philadelphia, USA*, May 1996.

[To95] Hing Wing To. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, University of London Imperial College of Science, Technology and Medicine Department of Computing, September 1995.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[WA99] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, 1999.

[Wad92] Phil Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming Languages.* ACM Press, 1992.

[Win] Noel Winstanley. Era user manual version 2.0. Available on the internet at URL http://www.dcs.gla.ac.uk/~nww/Era/Man.

[Win99] Noel Winstanley. Parallel programming by transformation. In *Euro-Par'99 - Parallel Processings*, volume 1685 of *LNCS.* Springer-Verlag, August/September 1999.

[Win01] Noel Winstanley. *Staged Methodologies for Parallel Programming.* PhD thesis, Department of Computing Science, University of Glasgow, April 2001.