Practical Structured Parallelism Using BMF

David C. Crooke

Doctor of Philosophy
University of Edinburgh

1998

To Rik

Abstract

This thesis concerns the use of the Bird-Meertens Formalism as a mechanism to control parallelism in an imperative programming language.

One of the main reasons for the failure of parallelism to enter mainstream computing is the difficulty of developing software and the lack of the portability and performance predictability enjoyed by sequential systems. A key objective should be to minimize costs by abstracting much of the complexity away from the programmer.

Criteria for a suitable parallel programming paradigm to meet this goal are defined. The Bird-Meertens Formalism, which has in the past been shown to be a suitable vehicle for expressing parallel algorithms, is used as the basis for a proposed imperative parallel programming paradigm which meets these criteria.

A programming language is proposed which is an example of this paradigm, based on the BMF Theory of Lists and the sequential language C. A concurrent operational semantics is outlined, with the emphasis on its use as a practical tool for increasing confidence in program correctness, rather than on full and rigorous formality.

A prototype implementation of a subset of this language for a distributed memory, massively parallel computer is produced in the form of a C subroutine library. Although not offering realistic absolute performance, it permits measurements of scalability and relative performance to be undertaken.

A case study is undertaken which implements a simple but realistic algorithm in the language, and considers how well the criteria outlined at the start of the project are met. The prototype library implementation is used for performance measurements.

A range of further possibilities is examined, in particular ways in which the

paradigm language might be extended, and the possibility of using alternative BMF-like type theories. Pragmatic considerations for achieving performance in a production implementation are discussed.

Acknowledgements

The full-time period of my research was funded by a grant from the Engineering and Physical Sciences Research Council. EPSRC is also the major owner of the Cray T3D used for performance measurements, which is operated by the Edinburgh Parallel Computing Centre.

My current employers at DERA have exercised considerable generosity in allowing me to schedule leave of absence and working hours to enable me to complete this work on a part-time basis.

Research for a thesis does not happen in a vacuum, and many people in the Department of Computer Science and elsewhere have helped me with advice and ideas. The original concept was inspired by a presentation on BMF given by Roopa Rangaswami during my early days here. Stephen Gilmore provided generous guidance on early versions of the semantics, though the remaining flaws are mine alone. Todd Heywood and Murray Cole both made their time available for many technical discussions. Other people whom I bounced ideas off include Colin Stirling, Marcus Marr, John Davy, Sergei Gorlatch and doubtless others I have forgotten.

I hope that it is evident from the text that the work presented directly follows on from that of David Skillicorn, who first adopted BMF as a potential vehicle for parallelism.

Most of all, I'd like to thank my supervisor Rosemary Candlin, who has an invaluable talent for casting incisive and pragmatic comments in a positive manner. I cannot recall a single occasion where I left a meeting with her without feeling more positive and more inspired than I did when I arrived.

Declaration

I declare that the contents of this thesis are my own composition, and that the research referred to therein is my own, except where explicitly stated otherwise in the text. Certain concepts and results presented have been previously published in [13]

Table of Contents

List of Figures

Chapter 1

Introduction and Motivation

This thesis is about parallel computing. Parallelism has long been heralded as the solution to the desire for ever more powerful computing systems, but it has so far been restricted to a few specialised niches. The reasons for the failure of massive parallelism to explode onto every desktop are many and complex, and will not be explored here. What we will do is propose a different way of looking at the role of parallel computing, and explore one technique for writing software for parallel systems.

1.1 Laws of Nature?

There are two famous homilies of the computer industry which have had a particular impact on the development of parallel computing.

The first of these was coined by Gene Amdahl, who was the founder of a computer company bearing his name which was set up to produce traditional mainframe computer systems in competition with IBM. In a keynote paper [17] Amdahl argued the case against parallel computing, on the grounds that any particular computation has a limited scope for concurrency, and that there is a diminishing return associated with exploiting it.

The basis of the formula which has become known as Amdahl's Law is that only certain sections of a program can be parallelised, and that however well this is done the resulting speedup will be hobbled by the remaining sequential part. The formula which Amdahl gives for speedup is

$$\text{speedup} = \frac{1}{\frac{p}{N} + 1 - p} \tag{1.1}$$

where p is the fraction of the work done by the program which is potentially parallelizable, and N is the number of processors onto which the work of the parallel part is divided. The speedup is defined as the ratio of time taken to complete the program on sequential and parallel computers. Clearly, if p is not close to 1, then the effect of increasing N becomes negligible above a certain point.

In fact, this law is places an unreachable upper bound on speedup; a realistic view can be even more pessimistic. There is a finite limit to how divisible the work represented by p is, which in turn limits N, and in some cases this will be less than the number of processors available in a parallel computer. In a practical situations, the parallelism can only exploited at the cost of additional administrative overheads in the program which eat away at the net speedup; these typically manifest themselves as the need for the communication of intermediate results between parallel processes. Another limitation is that realistic problems are generally not exactly divisible into an arbitrary number of evenly sized portions.

At first glance, Amdahl's Law would seem to effectively rule out any usefulness for parallelism with a single, simple observation. While the above formula is certainly accurate, in practice it does not have quite so devastating an effect. The reason for this is that it assumes a computation containing a fixed proportion of potential parallelism, and calculates how this would speedup on a given number of processors.

In fact, for the majority of computations which are sufficiently complex to warrant parallelism, as the size of the data set increases, the amount of parallelizable work increases faster than the amount of sequential work, with a ratio between the two which is at least linear; as the problem size increases, p tends to 1. This means that the number of processors which can effectively be used also scales at least linearly with the problem size.

Another facet to this is that the nature of many applications to which parallel computers are put means that the size of the computation is not determined by the real world problem which is being solved. These typically involve some kind of simulation or model, where the computational problem size is proportional to the level of detail. In these cases, using a larger computation for a given problem produces similar but more useful output, for instance due to its increased accuracy. Often, the criteria which determines the size of the computational problem is not the complexity of the real problem, but the amount of time or computing resources available - [25] is a typical exploration of this argument in a succinct fashion. This thesis takes the view that most of the interesting but tractable challenges faced by parallel computing lie in this domain.

There is also class of useful computations to which Amdahl's Law has no practical significance at all, which are known as *embarrassingly parallel* problems. These are typified by three characteristics:

- p is very close to 1 for realistic problem sizes
- the amount of communication overhead is negligible or zero
- the effective limit on N is very large

A typical example of an embarrassingly parallel problem is rendering a scene for a pixel display using the technique of ray tracing. This requires independently executing the same algorithm on the same scene data for each pixel of the display; the only significant limitation on the scale of parallelism is the number of pixels, which is over 1 million for a typical modern display screen.

While there is a great deal of craftsmanship in fully exploiting an embarrassingly parallel problem, they are not the focus of this thesis and will not be considered further.

The second computer industry truism which we will consider has a more subtle effect, both positive and negative, on parallel computing. This is an empirical observation which has no mathematical or other theoretical basis, but has nevertheless proven to be surprisingly accurate in practice and shows no signs of wavering. The principle is known as "Moore's Law" after its originator Gordon Moore; his original observation, made in 1965, was that the number of components on an integrated circuit (computer chip) doubled every year. The key point is that the growth in component density is exponential rather than linear with time.

This in fact applies to almost every metric of performance or capacity of computer systems which is not directly limited by fundamental physical constraints, with the time constant more typically averaging a year and a half. It can be observed that the performance and capacity of computer systems of a given class doubles every 18 months for a given cost in real terms, or conversely that the cost of a system of a given specification halves every 18 months.

There have been many advocates of parallelism, as well as advocates of exotic materials, who have at various times predicted limitations to the speed or performance of single processors made using the conventional CMOS process in which all modern computer circuits are fabricated, and an end to Moore's Law. These prophets of doom are always plausible at first because the numbers involved in future projections of an exponential formula always seem fantastic. With hindsight, these predicted limitations have thus far been breached on the schedule which Moore's Law has predicted, and there is as yet no tangible slowing down in the pace of technology.

The negative impact of Moore's Law on parallelism is the argument that there is no need to deal with the complexities of parallelism to deliver a given performance level for a given computational problem; we need only wait a while and there will be a sequential system which meets our needs. The claims to the contrary which have emerged from the parallel computing industry have only served to damage its reputation.

Stipulating the accuracy of Moore's Law as stated, there is a consequence which is positive for parallelism yet rarely observed. It is a reasonable observation that a given level of technology takes about a year longer to appear in parallel systems than in sequential ones. Consider the type of problem above which will consume any available amount of computing power, and for which time rather than money is a more critical limiting factor. If we postulate a parallel algorithm which offers a speedup of thirty on a parallel system of a realistic size, then Moore's Law tells us that at any point in time a parallel system has a four to six year head start in delivering performance over its sequential contemporaries.

We now go on to address a central question at which we have thus far been hinting...

1.2 Parallelism - Why Bother?

The majority of published work in the field of parallel computing is written from the viewpoint of abstract and pure science. There is nothing wrong with this viewpoint and it does not detract in any way from the validity of the research reported. However, the mindset which underlies the ideas in this thesis is a pragmatic one, and it is therefore necessary to question the role of parallelism in the wider field of computing.

Designing and implementing software to solve a significant problem on a sequential computer is an intellectually demanding task in itself. Performing the equivalent exercise in a typical parallel computing environment has a significant number of additional complications. Among these are:

- the work needs to be partitioned into a number of sub-problems which can be solved concurrently by a number of processes
- there is no realistic universal model of parallel computation to provide a theoretical foundation for algorithm design
- communication between these concurrent processes needs to be established
- the need for synchronisation must be addressed
- debugging implemented code is often almost intractable

- the performance of the resulting software is often highly variable on different systems

This begs the question - if it is so difficult to use, why bother with parallel computing at all? This thesis takes the view that there is only one answer to this question, and that is cost. In this context, the word "cost" is used in its broadest sense, and can be measured in any units depending on circumstances, though most often it is a function of time and money.

For almost any application, there are only two reasons which require using a parallel computer to solve a problem; the first is that it would take unacceptably long to solve on a sequential computer, or conversely that using a parallel computer allows scope for a larger problem size, while the second is that using a parallel system based on mundane technology may be cheaper than employing a state of the art sequential supercomputer. Until recently, parallel computers have been complex and expensive systems, and the first reason has been the predominant one, but the latter is increasing in popularity [3].

1.3 The Cost-Minimising Philosophy of Parallelism

The bulk of the cost of any information processing system does not derive from anything tangible. The thing which gives a system its identity is not the hardware, but the software, and this is increasingly the major cost of any system, as it involves the most human creative input. Mass production manufacturing techniques turn out processor chips with millions of transistors for a few dollars apiece, but they are sold for hundreds of dollars each to amortize the enormous cost of designing them - it is this intangible design we pay for, not the few grains of sand which are the raw materials.

In this situation where design costs dominate, the key to efficiency is re-use of the design effort. Open standards are now widely adopted in the hardware industry, driving the cost of common components down ever more rapidly. As early as

1946, it was proposed by Admiral Grace Hopper that this approach could apply to an entire computer system; it would be composed of a large number of identical, inexpensive, disposable modules. A massively parallel computer is, to some extent, a modern realisation of this ideal, the usual objective being to realise the performance of a large, expensive supercomputer at a lower cost.

The answer to the question of why we should even consider parallelism is a purely pragmatic one of cost - parallel computing is *only* justified where is reduces the total cost for a computation, whether that cost is measured in money, time, or some other metric.

Considering the term "costs" in its widest sense, a key aim should be to improve the portability of parallel software, to simplify its development and maintenance, to encapsulate parallel operations in a language and manner familiar to the user of existing sequential ones, and meanwhile to maintain a reasonably (though perhaps not optimally) efficient use of computational resources. The system we propose below is somewhat unusual, but we believe it has the potential to meet these requirements for a useful range of real problems.

1.4 Parallel Computing Models

As the reader will be aware, techniques for programming parallel computers are still in a state of infancy. Successful application of the tremendous computational power of parallel machines for even relatively simple problems requires a massive intellectual effort, as too much of the complexity inherent in parallelism is left to the software designer to manage. Getting the best performance from a parallel application requires persistent experimentation and a baroque inventiveness akin to the skill of *optimal programming* for drum store sequential computers of the 1950s.

There have been a number of proposals aiming to produce a simple and universal model of parallel computation, to fulfill the rôle that the Von Neumann / Harvard model has done so successfully for sequential computing. This aim is still far from being realised, and the lack of such a model, and the resulting lack of portability in parallel software, is a major obstacle to the proliferation of parallel computing. If we wish to make parallel computing widely accessible, we need to find ways of providing ordinary programmers with the expressiveness, portability and relative ease of software development that they enjoy with sequential systems.

We take the view that a wide diversity of techniques will be needed to handle different classes of problems with parallel computation. One type of approach which shows promise is based on *structured parallelism*, where the programmer may only express a parallel computation using a range of limited abstract forms, whose implementational detail (which is guaranteed to be reasonably efficient) and complexity is hidden as much as possible. Much of the burden of managing the parallelism is thus lifted from the programmer who can concentrate on solving the problem at hand.

1.5 Desirable Properties in a Parallel Software System

In an ideal world, for each parallel computer, there would be a compiler which would take any sequential program and transform it into a correct, reliable parallel one which would operate at one hundred percent efficiency. Unfortunately this is not the case, so it becomes necessary to identify the characteristics which are desirable in a parallel programming environment, and to see what compromises can be achieved between them:

Portability It is now a recognized fact that large software systems far outlive the hardware initially used to develop and run them, so portability is crucial. In the sequential case, true software portability is beginning to be achieved through rigorous standards for everything from programming languages to user interfaces. Though difficult, this is purely an administrative and commercial exercise, since a universal model of computation exists for these systems. For a parallel system to be truly portable, it has to be shown to fit into a computational model which can be efficiently realised on a wide range of parallel architectures.

Abstractness Having to directly contend with the complications of parallelism significantly adds to the burden of the programmer; a good system will abstract away as much of this detail as possible.

Performance There is no point in developing parallel software unless it achieves a certain level of performance, because it will always be easier and cheaper to build a sequential implementation. This notwithstanding, it is the overall cost in money, time or other resources which really matters.

Scalability Parallel software should scale; given a larger problem size, it should be possible to run the same program with a reasonable degree of efficiency on a machine with more processors, with little or no modification. A key feature of scalable parallel architectures is that they have a *constant valence*, and a good software system should allow for this, and not become limited by communication bottlenecks.

Soundness This last item is somewhat subjective, but is easily identified by considering systems where it is missing such as some of those described in section 2.1. If a system is to be usable it needs to be supported by techniques derived from a sound theoretical basis. It should be possible to perform complexity analyses which hold across the range of target architectures. It is desirable to have a strong semantics for a parallel language, which can be used to achieve a degree of confidence in the correctness of programs.

1.6 Parallelism and the Bird-Meertens Formalism

The Bird-Meertens Formalism [4] is a way of giving a sound theoretical basis to certain kinds of abstract data structure. A Bird-Meertens theory consists of an abstract data type and a number of second-order functions which are operations over the data type. When combined with a suitably rich *base algebra* of fundamental data types and first-order operations, a Bird-Meertens theory can be used to express meaningful computations. Although similar theories can be constructed for a surprisingly wide variety of data types [18], by far the most well developed is the Bird-Meertens Theory of Lists, and it is this data type which we focus on in this project.

A key feature of this theory is the fact that näive, inefficient algorithms expressed in BMF form may be successively transformed into implementations which are asymptotically more efficient. This approach to algorithm design is known as *program calculation* [4].

It has been demonstrated [43] that the operations commonly found in the BMF theory of lists may all be implemented with a high degree of efficiency on a wide range of parallel computer architectures. This is a key part of our approach; if the structure of a parallel computation is based solely around these operations, we can be certain that it can be made to deliver acceptable performance on any of these architectures.

1.7 Approach and Scope of the Project

We propose a paradigm of structured parallelism based on adding Bird-Meertens data types to an imperative, sequential programming language. The data structures built from these data types will be stored in a distributed fashion, and their associated second-order operations implemented in parallel, and they will form the sole mechanism for parallelism in the resulting language. In effect, we use an entire programming language as a base algebra for a Bird-Meertens theory.

Although not an obvious choice, this approach fulfills most if not all of the desiderata listed in section 1.5.

Portability is assured because the programmer expresses parallel operations solely in the form of high level BMF constructs; this provides architecture independence without sacrificing performance.

This type of system offers an excellent level of abstraction; many of the problems which the programmer is expected to tackle in a conventional parallel software environment can be dealt with automatically. Process thread creation and scheduling is implied by the structure of the program, as is synchronisation. There is no requirement for the programmer to set up complex sequences of communication between threads; communications necessary for thread management can be handled by the implementation, and data management tasks can usually be handled efficiently using Bird-Meertens operators.

The task which does remain for the programmer is ensuring that their software is free from undesired side effects and race conditions; for certain applications, this issue may be side-stepped by programming the base algebra functions in an entirely sequential style. For others this is not suitable, but we can support the programmer with theoretical tools to guide and structure their analysis.

The performance realised by programs developed this way is inevitably inferior to that of code which has been carefully designed and hand tuned for a particular architecture, but can be brought within an acceptable constant factor of optimum. This has to be traded off against the significant gains a high level approach can offer in reduced software development costs and greatly improved portability, considering the overall costs of performing a computation. The established trend for hardware prices to plummet while software costs soar shows no sign of slowing down, so this trade off can only get better with time, mirroring the shift from hand-coded assembly instructions to increasingly abstract high level languages that sequential programming has experienced in the last forty years.

The Bird-Meertens Formalism provides a sound basis for the system and makes available its theoretical results and tools. The transformational calculus can be used to improve the algorithms on which a program is based at an early stage in its design.

The modest requirements of the Bird-Meertens operations in terms of connectivity and communication bandwidth in the parallel machine ensure that they are scalable on a constant-valence architecture to massively parallel systems with thousands of processors. Communication bottlenecks are often the factor which limits performance in other types of parallel software.

For the purposes of this work we have created a parallel programming language based on adding the Bird-Meertens list data type to the sequential programming language C [2]. The choice of C as a target vehicle represents a compromise. Our proposal is for a system to be used for software engineering; although far from ideal in this regard, C does offer some support for good software engineering practices, and crucially, it delivers on performance.

We illustrate how it is possible to construct an operational semantics for this programming language, and how this can be used to increase confidence in the correctness of a program at all levels from a quick check of a statement with side effects to a formal proof. The performance of the BMF operations and programs made from them is examined empirically.

1.8 Structure of the Thesis

The next chapter presents a survey of existing research, including not only the work which this thesis uses as a starting point, but also some of the seminal material in the field of parallel computing, and other literature which shares superficial or detailed similarities with this work but follows different objectives.

Chapter 3 proposes a paradigm for the design and implementation of parallel software, which embodies the philosophy and approach to parallel computing outlined above. It then goes on to outline the design of a programming language based on C and the Bird-Meertens Formalism theory of lists, which is an instance of the paradigm, and which is used as a vehicle for the theoretical and experimental work presented in the rest of the thesis.

In chapter 4 we present an operational semantics for the BMF-based language, and show how it can be used with a variety of levels of rigour to establish the correctness of a parallel algorithm expressed in this language.

Chapter 5 describes an experimental implementation of the programming language, and performance studies examining the characteristics of the parallel operations in the language.

Chapter 6 undertakes a case study of the implementation of a realistic application using the programming language, and contrasts it with other techniques and efforts in the same application area. The target application comes from the engineering technique of finite element analysis, which offers a significant but non-trivial scope for parallelism.

Chapter 7 discusses a number of further aspects which the experimental evidence does not cover, including issues related to realistic "production quality" implementations of the programming language, performance optimisations, and BMF-style data types other than lists.

The final chapter summarises the conclusions which can be drawn from the material presented and proposes future directions for this work to take.

Chapter 2

Background and Literature Survey

This section attempts to give an overview of publications and other material covering existing work in related areas, a numer of which will form a basis for this project.

2.1 Parallel C-derived Programming Languages

There has been a recent flurry of activity in the area of developing parallel languages based on C, and more particularly C++. Unlike this project which aims to abstract away from the complexities of parallelism for the programmer, and to investigate its methods in an experimental context, these efforts are directed at producing production quality implementations of languages which provide simple parallel extensions to C++, with an emphasis on performance rather than ease of use. We will review some of them briefly.

2.1.1 Charm++

Charm++ was developed at Urbana by Kale and Krishnan [27]. They have devised a system which has a number of asynchronously communicating processes which they call *chares*. Parallel slackness is used to absorb communication latency, much in the same manner as in the BSP computation model [50]. To allow efficient implementation of operations which collect or summarise data from across the machine into a single global variable, they provide support for "accumulators" and "monotonic variables", as well as distributed data tables (similar in concept to Linda) and "branch-office chares", which have a server on each processor to handle certain requests locally.

2.1.2 pC++

pC++ (Bodin et al. [7]) is closer to what we are trying to do, in the sense that its parallel extensions are based on adding additional data structure constructors to the language rather than using a process-based approach. pC++ defines *collections*, which are regular data structures, in particular arrays, whose elements are C++ objects. The programmer specifies data distribution using special functions in a similar manner to that of High-Performance FORTRAN. Operations can be invoked across all of the elements in parallel (c.f. Bird-Meertens *map*).

2.1.3 C**

C** (Larus et al. [30]) is a data-parallel extension of C++, also based on the data structure approach. It defines *aggregates*, which are like multi-dimensional arrays, but allow concurrent operations across elements and some other functions, notably array slicing *à la* FORTRAN-90. Reduction-like operations are explicitly provided in the language.

2.1.4 Summary

All of these languages are useful, pragmatic contributions to parallel computing. However, they are from our point of view, each slightly lacking in some respect.

Charm++ is like traditional parallel languages in that it burdens the programmer with the responsibility for managing all of the complexity of process creation and communication.

pC++ relies rather heavily on the programmer for data distribution control - this allows tweaking of simple, regular problems for individual machines, but makes more general parallel computing rather tricky.

C** is less guilty of these sins, and seems simpler for a number of regular, data-parallel tasks, but in cases where parallel objects must interact, management of this communication is largely left to the programmer.

The principal manner in which all of them differ from the approach proposed in this thesis is that they are all derived from the objective of "bolting on" some parallelism in an arbitrary fashion to an existing programming language, with no coherent basis.

2.2 Computational Models and Parallelism

Developing complex systems, whether they are computer systems or otherwise, requires breaking them down into subsystems or modules, with well defined interfaces between them. One of the fundamental interfaces in computer systems is that between the hardware and the software - this is embodied formally by the computational model.

In sequential computer systems, the Von Neumann / Harvard computational model fulfills the needs so well that it is completely universal, and it is often easy to forget that it is implicit in our assumptions when designing systems, algorithms and languages. The situation with parallel computing is much less utopian - there is no one model which neatly fits all (or even a wide range) of the useful parallel architectures, and parallel algorithms have wildly varying performance on different types of parallel computers. This is recognized eloquently by Valiant in [50].

As Skillicorn points out in [43], the key difference between parallel architectures and sequential ones from this point of view is that sequential systems can only differ from one another in performance by a constant factor - it may be very large, but it is nevertheless constant. Thus, software systems can be developed in the knowledge that what is asymptotically efficient on one sequential architecture will be efficient on most others. On the other hand, a single parallel algorithm on different parallel architectures (or with reference to different parallel computational models) will have performance which varies by a *non-constant factor,* and so the assumptions which lead to universality do not hold.

This problem has led to the situation where there are a number of competing parallel models of computation, each with its own strengths and weaknesses, which correspond in varying degrees of accuracy to different groups of real parallel architectures (or in some cases, don't correspond to any real architecture). It is the view of this thesis that the chance of any universally suitable parallel model emerging in the near future is very small, and that different models are needed for different purposes.

The goal in this project is to investigate one possibility for a programmer's model, which concentrates on abstracting away from the complexities of parallelism and providing portability, at the expense of distancing the programmer from the hardware and consequently sacrificing a little efficiency.

We now consider some of the existing models and their merits.

2.2.1 The Traditional PRAM and its Variants

If there is one model of parallel computation which stands out from the rest as being reasonably widely accepted, it is the Parallel Random Access Machine (PRAM) [16].

The great advantages of this model stem from its simplicity - it merely consists of a number of processes sharing a global random access memory, reading data from it, writing data to it and performing local computations. All processors perform one of these operations at a time in lock-step synchrony. This simplicity makes the model very suitable as an environment in which to design algorithms and in which to reason formally about their semantics, correctness and performance.

There are many variants of the PRAM, e.g. CREW, EREW, CRCW, which vary constraints on the behavior of the shared memory. The H-PRAM we will consider separately below.

The difficulty with the PRAM is that it is an inherently unrealistic model - implementing synchronization at every step in a parallel machine is cumbersome to its design; building a shared memory system in which a large number of processors can access parts of it at arbitrary granularity and without a performance penalty is simply not possible at the electronic level. The model does not scale.

There have been attempts to circumvent this problem; a recent and intriguing effort by a US based company called Tera [49] uses latency hiding techniques to give the appearance of the PRAM's uniform memory system. The disadvantage is that in order to achieve this latency hiding, it is necessary to have a significantly larger number of threads of control in the parallel program than the number of processors in the machine, in a fashion akin to Valiant's *parallel slackness* [50]. In many cases this places a large burden on the programmer, pitting them in direct conflict with Amdahl's Law [17] and negating, in performance terms, any benefit introduced by the "flat" memory model.

One approach to handling this dilemma is to ignore it, to develop software to the PRAM model, and to simulate the PRAM behavior as efficiently as possible on whatever parallel machine is to be used. The problem with this solution is of course the aforementioned non-constant factor - for some problems, the natural, efficient solution on a PRAM is an unmitigated performance disaster when implemented for real on particular architectures.

The alternative is to find a model which discards some of the synchrony and the implicit total interconnectivity of the PRAM, thus restricting the model sufficiently to make it more realistic and more reflective of the hardware, while hopefully retaining its simplicity. In [44] and [45], Skillicorn presents a fairly wide survey of parallel models at all levels, and argues for use at the higher level of a model which is quite constrained but very universal, based on the Bird-Meertens Formalism itself.

2.2.2 The BSP Model

The Bulk-Synchronous Parallel (BSP) model is proposed in [50], as a model that can relate realistically to both software and hardware. The BSP model eliminates the often excessive synchronization of the PRAM by having synchronization occur at regular intervals, given by a parameter known as L, dividing the computation into *supersteps*. Communication between processes is by unidirectional (half-duplex) messages rather than by shared memory, and takes effect at the end of a superstep. If all processes have not reached their synchronization points at the end of a time interval, the remaining processes wait for the next interval and allow them to catch up. A process can be opted out of the synchronization system to allow it to run freely on a longer sequential task, while still being able to communicate with the rest of the program.

2.2.3 Parallel Scans

The work on parallel list scans due to Blelloch [6] bears a close resemblance to the use of the Bird-Meertens Formalism as a model for parallel computation. Performance studies [38] done in this paradigm bear out the assertion that list data structures can be used efficiently in parallel computing if appropriately implemented. The main contrast in emphasis to the concepts presented in chapter 3 is that Blelloch proposes a strictly data parallel paradigm.

2.2.4 The H-PRAM Model

The Hierarchical Parallel Random Access Machine (H-PRAM) is an extension of the PRAM, developed by Heywood et al. [22, 23]. This model removes, in a controlled fashion, some of the synchronization and connection richness assumed by the basic PRAM model - while the H-PRAM is not in a direct sense any closer than the PRAM to the reality of parallel hardware, it does allow some of the natural locality and asynchrony in an algorithm to be expressed in the model, and by retaining these features, which are desirable at the implementation level, allows a more efficient realization.

The added feature (or more pragmatically, the added instruction) which the H-PRAM offers over the traditional PRAM is that it permits the algorithm to *partition* the PRAM into a number of sub-machines. These then carry on, executing their assigned sub-tasks asynchronously and independently of each other. These sub-machines can in turn be further partitioned by their sub-tasks. When all of a particular set of sibling sub-tasks is completed, the sub-machines combine again to form the larger machine from which they were divided, providing a synchronization barrier.

The traditional PRAM may be considered as a special case of an H-PRAM which never splits (and hence does not express any extra locality). Any of the access-restricted variants of the PRAM (such as CREW-PRAM, EREW-PRAM) may be used with the H-PRAM, as the two types of extension to the PRAM model are orthogonal. In addition to this, Heywood proposes two major variants on the H-PRAM theme; the stronger type is the *private* H-PRAM, wherein processors in sub-tasks may only communicate amongst themselves, sibling to sibling; the other variant is the *shared* H-PRAM - in this model, processors may communicate with others outside their sub-machine. The shared variant is weaker in the sense that it only expresses the asynchrony in the algorithm, while still assuming a global shared memory, hence failing to encompass locality, and requiring potentially more demanding features in the underlying implementation.

2.2.5 The Actor Model

The concept of an "Actor" was devised in the 1970's by Carl Hewitt of M.I.T., but the work on the Actor concurrency model (outlined in [1]) is largely due to Agha.

An Actor system consists of a number of autonomous entities called Actors. Actors communicate by asynchronous message-passing - when an Actor receives a message, it performs a sequence of actions, including sending messages, creating other Actors, and changing its behavior to become a different Actor. The message system is fairly abstract - message delivery is guaranteed, but with no time limits and no ordering constraints.

The Actor model is rather unsuitable to describe parallel BMF systems because it cannot easily capture any of the regularity of the BMF operations. It is better suited to irregular, autonomous collections of processes, e.g. to simulating a dataflow system.

2.2.6 $\pi o \beta \lambda$

$\pi o \beta \lambda$(pronounced "Pobble") is a parallel programming language designed as a vehicle for reasoning about program design methods and semantics. This ongoing work [26] involves some interesting approaches to the problem of specifying tractable semantics for parallel systems.

The semantics of the language are expressed by mapping statements into the π-calculus [34] which is then used to prove the equivalence of an original program and one which has been optimised to increase its concurrency. An idea which will be considered in chapter 7 crops up here, that of passing back a return value from a child process before it has terminated, increasing the potential for concurrency. This has particular application to the BMF operations *reduce* and *fold*.

2.3 The Bird-Meertens Formalism

The Bird-Meertens Formalism is a concept whereby programs can be constructed out of set of simple operations (primitives) which act on a data structure, using operators of some underlying algebra on the data structure elements. The formalism (henceforth referred to as "BMF") is many things to many people, and we will attempt to explore some of these uses below.

2.3.1 BMF for Lists

The original BMF data structure is a simple list, rather like the built-in list datatype in ML. Since the idea is to use the operators to act on whole lists simultaneously (literally so when we are using BMF for parallel computation), a BMF implementation of lists is in fact a lot more similar to one dimensional arrays than any other sequential data structures. Some typical BMF operators for lists are given below, with the types which their equivalents would have in ML, as an aid to comprehension. Where a BMF primitive applies a diadic operator to list elements, a + has been used in the examples. These are taken from [4] and [8].

- $map : (\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$
  $map\ f\ [a, b, ...] = [f(a), f(b), ...]$
  The *map* primitive applies the function given (which according to the type of the list elements, may be a function in the underlying algebra, or may be made from BMF operators) to each item in the list independently, returning a list of the results.

- $reduce : ((\alpha * \alpha) \rightarrow \alpha) \rightarrow \alpha\ list \rightarrow \alpha$
  $reduce\ +\ [a, b, c, ...] = a + b + c + ...$
  This can be used, among other things, to implement sums and products. With this primitive the operator is assumed to be associative.

- $foldl : ((\alpha * \beta) \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta\ list \rightarrow \alpha$
  $foldl\ +\ q\ [a, b, c, d, ...] = (((...((((q + a) + b) + c) + d)...$
  This is also known as a "directed reduce", and performs the operations in the order implied by the brackets.

- $foldr : ((\alpha * \beta) \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ list \rightarrow \beta$
  $foldr\ +\ q\ [a, b, c, d, ...] = a + (b + (c + (d + ... + q))...)))$
  This is the right-associative variant. Skillicorn in [43] specifies foldl and foldr without the identity elements (here denoted "q")

- $inits : \alpha\ list \rightarrow \alpha\ list\ list$

  $inits\ [a, b, c, ...] = [[], [a], [a, b], [a, b, c], ...]$

  This generates a list of prefix sub-lists of its argument, not useful in itself, but a necessary building block operator for constructing complex algorithms. Combinations of such operators are needed where iterative constructs would be used in imperative programming.

- $tails : \alpha\ list \rightarrow \alpha\ list\ list$

  $tails\ [...a, b, c] = [..., [a, b, c], [b, c], [c], []]$

  This is the mirror image of the *inits* operator.

- $prefix : ((\alpha * \alpha) \rightarrow \alpha) \rightarrow \alpha\ list \rightarrow \alpha\ list$

  $prefix\ +\ [a, b, c, ...] = [a, a + b, a + b + c, ...]$

  This operation gives the result of applying a reduction to all of the prefix sub-lists of the list given as an argument.

- $suffix : ((\alpha * \alpha) \rightarrow \alpha) \rightarrow \alpha\ list \rightarrow \alpha\ list$

  $suffix\ +\ [..., x, y, z] = [..., x + y + z, y + z, z]$

  The mirror image of *prefix*

- $prefixl : ((\alpha * \alpha) \rightarrow \alpha) \rightarrow \alpha\ list \rightarrow \alpha\ list$

  $prefixl\ +\ [a, b, c, ...] = [a, a + b, (a + b) + c, ...]$

  This operation is equivalent to *prefix* if the + operator is associative, but guarantees order of evaluation where it is not. The relationship to *prefix* is similar to that between *reduce* and *foldl*. The mirror operation for *prefixl* would be *suffixr*

- $filter : (\alpha \rightarrow bool) \rightarrow \alpha\ list \rightarrow \alpha\ list$

  $filter\ p\ [a, b, c...] = [a, c, ...]$

  where $p(a) = true$, $p(b) = false$, $p(c) = true$ etc.

- $zip : ((\alpha * \beta) \rightarrow \gamma) \rightarrow \alpha\ list \rightarrow \beta\ list \rightarrow \gamma\ list$

  $zip\ +\ [a, b, c, ...]\ [x, y, z, ...] = [a + x, b + y, c + z, ...]$

  The *zip* operator is like a diadic version of *map* and requires list arguments of equal length.

- $cross : ((\alpha * \beta) \to \gamma) \to \alpha\ list \to \beta\ list \to \gamma\ list$

  $cross\ +\ [a, b, c...][x, y, z, ...] = [a+x, a+y, a+z, ..., b+x, b+y, b+z, ..., c+x, ...]$

- $concat : \alpha\ list\ list \to \alpha\ list$

  $concat[[a, b], [c, d, e], ...] = [a, b, c, d, e, ...]$

There is no hard and fast rule by which one can define precisely which potential BMF operators should be implemented and which composed from others. Formal reasoning about data types of this nature can indicate criteria for completeness of a set of operations [46], but for our purposes, we are interested in performance. With such a rich base algebra, we do not require any particular operation to be parallel, but we need operations which allow us to implement algorithms efficiently on a parallel computer, and can justify inclusion of additional operations solely as a performance improvement.

2.3.2 BMF and Program Calculation

The purpose to which Bird himself puts BMF is the study of what he calls "Program Calculation" [4]. The concept is that, given a specification for a program in terms of a näive and inefficient implementation in a functional style like BMF, it is possible to derive a reasonable implementation by a process of equational reasoning, using identities on the BMF functions.

One of the classic examples of this technique is the derivation of a function to calculate the *maximum segment sum* of a list; given a list of positive and negative numbers, the maximum segment sum is the highest number which can be arrived at by taking a contiguous sub-list and adding the members. The following derivation is taken from [4].

Let us define the following functions:

- $gtr :\ int * int \to int$

Returns the numerically larger of its arguments

- $max: \ int\ list \rightarrow \ int = foldl\ gtr\ -\infty$
- $sum: \ int\ list \rightarrow \ int = foldl + 0$
- $concat: \alpha\ list \rightarrow \alpha\ list \rightarrow \alpha\ list$
- $segs: \alpha\ list \rightarrow \alpha\ list\ list = concat \cdot map\ tails \cdot inits$

$$mss = max \cdot map\ sum \cdot segs \tag{2.1}$$

$$mss = max \cdot map\ sum \cdot concat \cdot map\ tails \cdot inits \tag{2.2}$$

by substitution of the definition of *segs*. The fact that *sum* and max are both associative is now used to pull out the *concat* and replace it with another *map*. This allows three steps to be pushed together in a single map operation, which can be made more efficient by replacing the function.

$$mss = max \cdot concat \cdot map\ (map\ sum) \cdot map\ tails \cdot inits \tag{2.3}$$

$$mss = max \cdot map\ max \cdot map\ (map\ sum) \cdot map\ tails \cdot inits \tag{2.4}$$

$$mss = max \cdot map\ (max \cdot map\ sum \cdot tails) \cdot inits \tag{2.5}$$

Now define $x \ \oplus \ y \ = \ gtr(x + y, 0)$

$$mss = max \cdot map\ (foldl\ \oplus\ 0) \cdot inits \tag{2.6}$$

$$mss = max \cdot prefixl\ \oplus\ 0 \tag{2.7}$$

Defining $(x, y) \otimes z = (gtr(x, w), w)\ where\ w = y\ \oplus\ z$

$$mss = fst \cdot foldl \otimes (0, 0) \tag{2.8}$$

The first part of the derivation sets up the two key steps of the transformation, from the expression in 2.5 to 2.6, and from 2.7 to 2.8. In each case, the complexity of the function is reduced by order $O(n)$

This derivation has turned an $O(n^3)$ algorithm given as an obvious specification into a non-obvious $O(n)$ implementation. Of course, this requires knowing which transformations of the function constitute improvements, as we could just as easily have turned the latter into the former.

A transformation technique of particular interest to parallel computing is the use of *list homomorphisms* [10, 19]. A list homomorphism is any function f for which there exists a corresponding combination operator $\otimes$ such that

$$f(x \odot y) = f(x) \otimes f(y) \tag{2.9}$$

where $\odot$ is the list concatenation operator and x and y are lists. In pragmatic terms, if f is a homomorphism, then if we can find a suitable definition of $\otimes$ we have found a divide and conquer implementation. The key to using program transformation to find a potentially efficient parallel implementation is often to transform the function in question into a homomorphism.

The definition of heuristics and algorithms to perform program transformations in this way is an open problem for which a full treatment is outside the scope of this work.

2.3.3 BMF and Parallelism

The main interest we will take in BMF is its suitability as a vehicle for expressing parallelism. Skillicorn has outlined an argument which shows that the BMF list operators all have reasonably efficient implementations on all major parallel architectures [43], and has argued extensively for a BMF-based programming model [45, 44]. This property of the functions is largely independent of and orthogonal to their usefulness in program calculation, though the latter comes as a welcome bonus.

They key to the success of BMF lies in its scalability - most of the operations require only nearest-neighbour communication, so that unlike the PRAM, the parallel architecture required is much less densely connected and is not only physically realisable but scalable. All of the operators defined in section 2.3.1, can be implemented with only a Hamiltonian circuit (i.e. a ring topology with a constant valence of 2), although operations like *reduce* and *filter* require a more dense topology for optimal complexity. The *filter* is the more awkward, since, to maintain the consistent data distribution policy necessary to minimise administration overheads, we must "compress" the dead space out of the list by relocating the elements which are retained. The technique for implementing this is to use a reduction operation first to determine the final location for each list element which is being retained, and then to execute the data transfer as a many-to-many operation.

2.3.4 Using BMF Lists on Densely Connected Parallel Architectures

Although one of the useful properties of BMF lists as a parallel data structure is their ability to be used efficiently on a parallel machine with a low valency of connectivity between the nodes, it would be desirable to be able in certain circumstances to take advantage of denser connectivities when they are available.

A number of compound list operations which could be used with the BMF list data type are proposed in [28], and illustrated by algorithms for several practical problems, as well as transformations to map them onto constant valence architectures.

The principal use for these computation strategies in our BMF system would be "behind the scenes" optimization of certain operations, in particular administrative tasks, although there would be potential performance gains in offering these operations at the language level, so long as fast implementations can be guaranteed in cases where they have to be transformed.

2.4 Other Structured Parallelism Systems

The idea of using a restricted set of parallel constructs as a basis for a tractable parallel system is not a new one. There are a number of other approaches based on the idea of using predefined parallel structures, and we will consider two of them here.

2.4.1 Algorithmic Skeletons

No overview of this subject area would be complete without considering skeletons. Cole's work in this area [11] essentially meets a very similar objective - both skeletons and BMF attempt to provide the user with a limited range of structures, which while they are abstract from the details of parallel implementation, can be provided efficiently on a realistic parallel machine. The difference in approach is that skeletons are a *control* abstraction while BMF is a *data structure* abstraction.

Cole observes that the most common approach to solving a new problem with a computer program is to use a variation on an existing tried-and-tested algorithm for some other problem, while retaining a similar overall structure. Thus we can classify a large number of programs according to their basic strategies, such as "divide and conquer" and "branch and bound". An algorithmic skeleton is just such a structure, whose user (the programmer) is left to fill in the details, or to continue the metaphor, to put flesh on the bones. Each skeleton has an efficient parallel implementation which is transparent to the user, who is relieved of the burden of managing the complexity of the parallelism and is free to concentrate on getting the details correct. This characteristic is also a central objective for the BMF approach.

Darlington et al. propose a similar approach to Cole's, using functional rather than imperative languages as a basis [15].

2.4.2 P^3L

P^3L (the Pisa Parallel Programming Language) is the result of a collaborative project between the University of Pisa and Hewlett Packard [14]. P^3L consists of a subset of C++ with a set of parallel *constructors* which allow the programmer to express an algorithm in terms of well-known parallel forms (pipelines, process farms, trees, etc. and one based on geometric patterns). These are mapped onto the parallel hardware by implementation *templates,* which are particular to a given machine architecture. Optimizations exchanging different compositions of the parallel constructors are performed heuristically on a local basis.

Like Cole's skeletons, P^3L provides parallel constructions with an algorithmic rather than data oriented emphasis. However, P^3L provides a mechanism for controlling groups of processes rather than defining the basic structure of the algorithm itself.

Chapter 3

A Method for Developing Parallel Software

This first aim of this chapter is to quantify in abstract the paradigm for parallel programming being proposed. It then goes on to describe a notional programming language which embodies a particular instance of this paradigm, and which was used as a basis for the experimental and theoretical results presented in later chapters. Some implementation details are also considered, most notably the problem of data distribution.

The distinction between these concepts must be emphasised. The central tenet in this thesis as a whole is that the general paradigm proposed is one of many suitable approaches to developing parallel software. The remainder of the text is devoted to supplying evidence to support this view, based on arbitrary and specific implementations of certain features. The significant part of this evidence consists of those principles relevant to the paradigm in general which can be inferred from the specific experimental data.

3.1 The Generic Paradigm

Most simply put, and in its widest sense, the paradigm for expressing parallel algorithms which we propose in this thesis consists of using second order data types and operations, such as those of the Bird-Meertens Formalism, to express parallelism in an imperative programming language which is otherwise sequential.

One way in which this could be achieved would be to start with the same basic syntax as the original sequential language, and to add the following:

- explicit type constructors and reference elements for the data structures which will be used to structure concurrency, which distinguish them syntactically from the standard data structures of the language

  and

- a form of syntax for expressing the parallel operations in a natural style. Among the forms this could take would be a new class of expression with the parallel operators as reserved words, or it might simply be phrased using an existing syntax for invoking subprogram or library functions

The exact manner in which the additional syntax is fitted in is not important, so long as it is possible for the parallel constructs to be identified, expanded and optimised at compile time, thus ensuring scope for performance tuning. The key requirement is for the end result to be a cohesive and expressive whole.

Parallel constructs in a programming language should be fully compositional and integrated, and have no special peculiarities other than their parallelism. On a practical level this is taken to mean that parallel and sequential constructs, both of data and of program code, should be able to be nested in an arbitrary manner. Many parallel programming paradigms only permit parallelism at a single, global level, and this severely hampers the use of a modular approach which is now universal software engineering practice.

This principle has been adhered to in the specific instance of this paradigm outlined below and used in subsequent chapters.

3.1.1 Comparison with Existing Work

A new paradigm is most easily placed by reference to existing material. While the natural tendency is to list the features and concepts which the paradigm embodies, it is just as important and informative to identify what the paradigm is *not*, i.e. the concepts from other systems which might have been used but are instead replaced by something quite different or simply discarded.

The combination of concepts which makes our paradigm unique can be approximately identified by comparison with existing work in two areas. The first of these consists of existing work with BMF and parallelism, which tends to be theoretical rather than practical in character; the second area concerns other approaches to *structured parallelism* which are not based on theories like BMF.

The existing work with BMF is best exemplified by the programming methodology in some of Skillicorn's work [45]. In this style, the "programming language" consists almost entirely of BMF constructs, with a minimal underlying *base algebra* of integer arithmetic, and algorithms must be expressed solely in terms of BMF operators.

Although the work in this thesis expands on Skillicorn's idea of using BMF as a vehicle to express parallelism, it differs conceptually in quite a number of ways. The most fundamental of these is that our paradigm is based in an imperative domain, rather than a purely functional one. The imperative domain introduces the concepts of multiple threads of control, side effects, race conditions, implicit communications, etc. and all the freedom (and risks) for the programmer that these imply.

The most obvious difference in character between our paradigm and Skillicorn's is the richness of the base algebra. By using a base algebra consisting of simple, referentially transparent functions with $O(1)$ run time for operation, Skillicorn gains much in the scope for algebraic transformation [4] and performance predictability [8], but at the expense of a restrictive language with poor expressibility.

By contrast, we offer a programmer an entire sequential language as the base algebra, and allow them to compose and nest it with parallel BMF operations in an arbitrarily complex manner. This allows the expression of almost any style of algorithm, but constrains the use of tools like transformations to the algorithm design stage, since the complexity of analysing an implemented program in order to perform transformations would be far beyond any conceivable compiler.

Another main difference in our paradigm is the way in which the problem of data distribution is approached. We achieve compositionality by using a universal "flat" model of data storage, where the data distribution on entry to and exit from all parallel operations follows the same simple rules and is compatible. This allows the functions to be composed in arbitrary ways. Our suggested criteria for a satisfactory, general purpose data distribution strategy are given in section 3.3

In some of his work, the approach which Skillicorn takes is to calculate an optimal starting data distribution for any given algorithm by identifying the optimal output data distribution, and then back-tracking through the algorithm to the ideal starting point. The advantage of this is that when the algorithm is run in such a configuration, the data neatly transforms from function to function with a minimum of communication overheads. In practical situations, this would only be suitable to highly optimise a fixed algorithm with known execution behaviour for particular sizes of data sets; in the more general case it has a number of disadvantages.

The principal problem is that these optimal data distributions cannot be composed in any obvious manner; if two algorithms are combined by running one after the other (i.e. by functional composition) then the optimal data distribution for the first must be recalculated to ensure that the output suits the needs of the second. Another limiting factor is that the cost of calculating these optimal distributions can easily outweigh the efficiency savings over using a simpler but less optimal strategy.

The area in which existing work describes systems most similar to that proposed here is the field which we denote by the term *structured parallelism*; methodologies typical of this area include Cole's *Algorithmic Skeletons* [11] and P^3L[14]. The key way in which our paradigm differs from most others in this field is that the mechanism by which parallelism is created and controlled is based around the data structures, rather than by reference to certain algorithm patterns.

3.2 A Programming Language

We now introduce a programming language designed to fulfill the objectives stated above, which has been used for the experimental and theoretical work in later chapters. The vehicle used for parallelism is the Bird-Meertens Theory of Lists, which is the most widely understood and studied data type among those suitable for this purpose.

In a real world situation the choice of a sequential language to use as a basis for our parallel one would be more or less unrestricted; it would indeed be perfectly reasonable to create an entirely new language in lieu of using an existing sequential one at all. For experimental purposes there are a number of desirable characteristics, and other pragmatic constraints which guide the choice of base language.

- The language should have a sufficiently rich syntax and type system to allow the addition of the BMF operations and type constructors in a natural manner
- It should be well known in its sequential guise, so that its concepts will be familiar to the target audience and will not obscure the points being made, and so that the modified language can be easily differentiated
- It should be widely available, preferably with existing compilers which can target parallel systems
- It should not compromise performance; in particular, the type system should *not* be so complex that it requires run time checking and interpretation - after all, performance is the sole *raison d'être* of parallel computing

A pragmatic choice quickly narrows down to two popular languages - C [2], and its object-oriented cousin C++ [48]. They satisfy all of the criteria extremely well. The choice between them is a moot point, and turns out to be something of a red herring. We are advocating good software engineering practice, and the extra features of C++ are certainly useful in achieving it; however, they can almost all be boiled down to syntactic sugar (C++ can be compiled into C with no loss of performance or generality) and are essentially orthogonal to the concepts under examination.

With due apologies to Bjarne Stroustrup, the rest of this thesis takes the view that the distinction between C and C++ is more or less irrelevant to the discussion at hand, and for simplicity we will construct our prototype language using C.

3.2.1 Language Syntax

The bulk of the language syntax we will use is identical to C - indeed, the C programming language is almost a subset of the BMF prototype one, with the following exceptions:

- Pointer arithmetic is not permitted
- Type casting is not permitted

In essence, what this achieves is that pointers are "black box" handles which can only be generated by the language run time system or the compiler, and not by the programmer.

These constraints are essential to enable us to construct a semantics of the language based on the source code syntax. If pointers and casts are allowed to be used, then the binary level of the target machine is exposed, and it is possible to construct programs which have different behaviour on different architectures.

3.2.1.1 Type Constructors

The first question we come to when choosing the syntax for this language is how to represent the type constructor for a list. The way in which we do this is to use a reserved keyword, and to borrow the remainder of the syntax from the array constructor. Thus, to declare a list of integers named `fred`, we would use the syntax:

```
int list fred;
```

This would declare a variable `fred` and allocate space for a list pointer, but would not allocate any storage.

This can be compared with the C declaration

```
int *fred;
```

which allocates a pointer which can later be used as a handle for an array, but does not allocate storage for the array elements.

For the deconstructor which allows the programmer to reference individual list elements we use the same square bracket syntax as the array deconstructor. This causes no confusion for a (hypothetical) compiler, since the type of any list or array expression is known at compile time.

3.2.1.2 Storage Allocation

We take the view that the `list` type behaves in a similar manner to a pointer, in the sense that it is a fixed size descriptor for a variable sized collection of data, and that this is visible to the programmer. We therefore need a means to allocate and deallocate storage for the data itself, analogous to `malloc` and `free` for sequential C storage space.

We propose the following notations:

```
newlist(fred,26);
```

```
disposelist(fred);
```

We also define the following intrinsic function (c.f. `sizeof`) to determine the length of a list:

```
lengthof(fred)
```

The additional complication that we face is that we want lists to be distributed across the separate memories of a distributed memory parallel system, which we can view as comprising a two dimensional memory space.

The issue of a data distribution policy is discussed in section 3.3, and optimisation of this is pursued in chapter 7.

3.2.1.3 Syntax for Base Algebra Functions

The following objectives must be met by a base algebra syntax:

- It must be possible to express the full richness of the C language, with things like local automatic variables
- Since it is not possible for the programmer to define higher order functions in the way that they could in a functional language like ML, it must be possible to have arbitrary numbers of additional parameters which can be passed to the base algebra function

The syntax which has been chosen is inspired by the way that object methods are implemented in C++ [48] - the pointer to the object is passed to the method as an extra, hidden argument which is not shown explicitly in the function declaration for that method, but which can be referenced using the reserved keyword `this`.

Since different BMF operations require base algebras with different numbers of parameters, and since many operations might wish to refer to other elements of a list and not just those that they were invoked to operate upon, it was decided that these list handles and indices would be made explicit in the declaration of the base algebra functions.

A base algebra function is therefore declared with its first arguments being those representing the lists and indices upon which it must operate, followed by any additional arguments which may be passed through the BMF operator. The handles to the lists are passed as the appropriate `list` type, with indices being passed as type `int`.

For BMF operations where the return type is dependent on the input type, such as *reduce* and *foldl*, the return type of a base algebra function must be identical to the basic type of the list arguments; for those BMF operations where the return type is free, it may be any type (including `void`) but this must match the type required by the BMF operation invoking it.

For the BMF operation *filter* which strips a list according to criteria established by the base algebra, the return type must be `int` and the return value follows the usual C truth rules, i.e. non-zero means true.

It should be emphasised that although the base algebra functions are defined using a C notation which implies that they are subroutines, it is envisaged that most of the simpler ones would be inlined at compile time, thus avoiding a huge potential overhead.

3.2.1.4 Syntax of BMF Operations

For the invocation of the BMF operations themselves, we introduce a functional style syntax reminiscent of the Curried function notation available in ML. The value of the expression `map f l` is defined as the result of applying `f` to each element of `l` in parallel, and will either have a `list` type or have type `void`.

The first few arguments to the base algebra function being called are generated by the BMF operation from the list(s) being used. If the programmer requires additional arguments to be passed, these are appended to the function name in the conventional manner, e.g. `map f('a',3.5) l`

3.2.1.5 Examples

All of this can be made much clearer by considering a concrete example. Let us define the following base algebra functions:

```
float sum(float list l, int lindex, int rindex)
{
  return l[lindex]+l[rindex];
}

int bigger(float list l, int i, float ref)
{
  return (l[i] > ref ? 1 : 0);
```

```
}
```

These might be used in the following code fragment:

```
float total;
float list l[100];
float list big;
int howmany;

/* code to place some values in l deleted */

total=reduce sum l;
big=filter bigger(50.0) l;
howmany=reduce sum (map bigger(23.1) l);
```

3.3 Data Distribution

An issue which is central to the performance of any parallel system is the way in which data is distributed across a machine. Of course this concept is not applicable to a shared memory system, but we are considering truly scalable systems which must of necessity have a distributed memory architecture.

One of the key choices which had to be made for experimental purposes was the data distribution policy which would be used in an actual implementation

The principal objective which guided the choice of data distribution policy was the need to make it flexible and re-usable. The intent was to make something which would deliver reasonable performance without violating the principle that it must be self-consistent.

3.3.1 Requisite Properties

There are a number of properties which are desirable, or indeed essential, features of a data distribution policy if it is to be suitable for the BMF based paradigm.

Composability Unlike many parallel programming paradigms which only allow "parallel" data structures to be used at the top level, the paradigm proposed here is intended to allow arbitrary compositions; a structure containing lists, a list of arrays or even a list of lists are equally valid and useful.

Repeatability If we are to allow the arbitrary composition and nesting of the BMF operations, it follows that for every BMF operation, the range of permissible distributions of input data and the range of possible distributions of output data must be identical.

Locality The principle of locality of reference is fundamental to the predictable performance properties of the BMF operations. The assertion upon which this is based is the fact that, if we consider a task or thread to be associated with a list element, the majority of data references will be made to that element or its neighbours. It is therefore essential that adjacent list elements are located either on the same processing node, or on topologically adjacent ones.

Immediate Reference For flexibility in the imperative approach that we propose, we wish to be able to treat the list data structures as linear arrays when it is convenient. In order to achieve this, the data distribution must be constructed so that the shape and layout of any list can be described by a fixed size tuple of parameters, in such a way that any processing node in the machine can calculate the node and memory location of any list element using only this tuple of parameters, in $O(1)$ time.

Aggregation A naive method of data distribution might attempt to ensure load balancing by placing the first element of a list on the first node, the second on the second, etc. then wrapping back to first. In parallel systems in general, there is a trade-off between speedup achieved by spreading computation across an increasingly large number of processing nodes, and the overheads involved in scheduling increasingly small parcels of work. Because of this, and because of communication overheads, in order to cope with a finer grain of parallelism it is necessary to have the capability to aggregate a number of contiguous elements of a list on a single processing node.

3.3.2 A Suitable Distribution

The desiderata above almost dictate the method of data distribution outlined here, which is used throughout the experiments detailed in later chapters. It should be borne in mind that performance in some circumstances is quite dependent on the way that data is laid out; this point will be discussed further in the relevant context.

The detail of the data distribution used is as follows:

- Let the machine consist of n processing nodes, each with a distinct memory space, which we number 0 through $n-1$. It is assumed that the topology of the machine is a superset of a ring (one-dimensional toroid), i.e. that node 0 is topologically adjacent to node 1, 1 to 2, etc. and $n-1$ to 0.

- A contiguous region of memory on each processor is reserved for the storage of list items. It is convenient, though not essential, for this region to be at the same memory address on each processor; we assume for the rest of this description that it is. This memory area will be managed globally, in a similar fashion to the dynamic heap allocation area in a sequential C program

- The location and layout of a list l is given by a fixed size data structure which is treated much like a pointer in a sequential programming language. This descriptor consists of the 4-tuple (p_l, a_l, b_l, n_l), where p_l is the processor node on which the first list element resides, a_l is the memory address at which is resides, b_l is a value we call the *blocking factor*, and n_l is the number of elements. In practice, this will usually require four machine words.

- The first b_l elements are stored consecutively on node p_l starting at address a_l; the next b_l elements are stored on the next node, and so on. After node $n-1$ it wraps back to node 0, and after node $p_l - 1$ it comes back to the starting node p_l and continues on contiguously from where it left off.

- When referring to them individually, we number list elements from 0. The location of an individual element i is described by a 2-tuple (p_i, a_i) giving the processor node and memory address where it is stored. The following calculation is used to obtain this location from (p_l, a_l, b_l, n_l):

$$p_i = (p_l + i/b_l)\%n \tag{3.1}$$

$$a_i = a_l + s_l(i\%b_l + b_l(i/(b_l n))) \tag{3.2}$$

The division operations shown are the integer "*div*", and the % signs represent the "*mod*" (integer remainder) operation. The value s_l is the size of a list element, which is calculated at compile time based on the element data type.

Note that n_l is not used in the above calculation; there is however a reason why we need to keep track of it, while a normal C pointer or array reference does not indicate how large the storage space is. The language implementation will need to know the length of a list in order to structure parallel operations based on it.

This computation of the location of a data item requires 11 integer operations, some of which may be resolved or subject to strength reduction at compile time. This compares with only 2 operations to calculate the address of an element of an array in the sequential C language in a single address space environment; for accessing a parallel data structure spread across many address spaces, this is a reasonable cost, and is small compared with the cost of actually performing a remote access to exchange data with a remote node.

If the programmer writes (non-parallel) code to perform a sequential run through the list data then the average lookup cost is more similar to that for an array; as in the case of an array, the address of the list element is an induction variable, the only difference being that a test is required to determine when the end of a block has been reached, thus it requires 2 operations per iteration rather than 1, and only requires the more extensive computation when crossing a block boundary.

This data distribution strategy encompasses the requisite properties; the layout of a list is defined by small, fixed number of parameters, and using this information alone, the location of any element can be computed in $O(1)$ time.

Clearly, this distribution satisfies all of the requirements detailed in the previous section.

Composability is achieved by making the list descriptor fixed size and by treating it like a pointer; a list of lists just becomes a number of independent lists, one of which happens to be a list of these 3-tuple descriptors. The generality of allowing the list to start on any node leaves scope for reasonable load balancing when many lists are in use.

Repeatability is something to be addressed in the implementation of the BMF operations; this is nonetheless assisted by the fact that it is easy to decide where a list element should be placed as soon as one is created, facilitating concurrent restructuring of lists where it is necessary.

Locality is addressed in exactly the fashion described; recall that the capacity to embed a ring topology in the actual topology of the parallel machine is a basic stipulation of the BMF method.

Immediate reference is ideally catered for; the calculation of an element location requires a total of 11 integer operations, all of which can be expected to be single-cycle instructions on a modern RISC processor; if division is expensive, this can be avoided by ensuring that both b_l and the number of processors are always powers of two, which allows the *div* and *mod* operations to be implemented as shifts and masks respectively.

Aggregation is of course catered for by the blocking factor, which is otherwise a largely arbitrary parameter.

3.3.3 Memory Management

This algorithm for data distribution makes the problem of memory management quite simple; one block of memory is reserved for the list storage heap on each processor, with these blocks having the same size and alignment (and where possible, hardware address) on each node. This allows global memory allocation to be handled by a single thread on any arbitrary processor, which may well reside outwith the pool of processors used to run the user's code and need not share memory space with them.

In a massively parallel environment there is an obvious space - performance tradeoff to be invoked by having multiple aligned memory blocks, each under the exclusive control of a separate memory manager thread.

3.3.4 Tuning Data Distributions

There is scope in any type of parallel program to tune it for performance on a particular machine by altering the detail of the data distribution. In the distribution detailed above, this scope is embodied in two areas: varying the placement of the start points of multiple lists, and choosing the blocking factor. The latter is an obvious target for a compiler directive of the style traditionally used to assist auto-vectorising FORTRAN compilers. This is discussed in chapter 7.

Chapter 4

Semantics

This chapter will introduce and describe a non-deterministic operational semantics for the language and illustrate its use in reasoning about the outcome of programs which have a variety of orderings and side effects.

The execution of a parallel program is defined as a sequence of atomic transitions between instantaneous *program states*. The semantic rules define permissible transitions between states. The key property is that they are non-deterministic; from any one program state, there may be a number of possible transitions, corresponding to the number of active *threads* in the program.

The "meaning" of the program is captured by the set of all possible execution traces, formed by a closure over the rules which define the semantics. The ideal for an argument or proof constructed with these semantics is to demonstrate that, for some key part of a program or algorithm, the significant part of the end result is deterministic, even though the path through the computation need not be.

This leaves a hypothetical implementation free to execute concurrent operations in any order; to leave it free scope as to if, how and where to implement concurrency, it is necessary to apply the following further constraint - unless *all* possible paths of execution terminate, then the result of the program or fragment is *undefined*.

What this additional constraint corresponds to for the programmer is that any program which assumes concurrency across a BMF operation and attempts to exploit it, for example by communicating interim results between the threads invoked by a *map* operation, is likewise undefined, and might deadlock on some valid implementations.

4.1 Scope

The purpose to which the semantics are to be put is to analyse the behaviour of concurrent sections of code, with a "language level" perspective. We therefore limit the scope of our attention in the following ways:

- Traditional C language "hacker's" techniques such as casting pointer types and directly probing memory locations have no place in this complex programming environment, and so they are ruled out in the language definition. Programs are assumed to be well-formed, and we do not make provision for partial failure of the machine or operating environment. The combination of these restrictions allows us to use a simplistic, high-level model of a program state.

- The semantics of I/O operations are simply ignored; apart from the fact that I/O is invariably performed sequentially, and is therefore uninteresting in this context, we can theoretically, without loss of generality, model a filestore as an item in a data space managed by a particular thread which fulfills the role of the operating system kernel.

- We avoid making a detailed consideration of the behaviour of operations, e.g. arithmetic overflow; to do so would bog us down in implementation-level detail. We restrict our attention to examples where we can safely assume that sequential operations are free of this kind of problem.

- We avoid a clumsy hierarchical representation of the threads and data spaces, and instead use a more simple "flat" model. In order to do, we impose the restriction on program segments to be considered with the semantics that all data to be referenced by multiple threads is stored globally, i.e. is part of a list or object, or is statically allocated or referenced by a pointer. This does not reduce the generality of the programs which can in principle be analysed, as converting a non-conforming routine into this form is fairly straight-forward.

- Similarly, we avoid considering the C rules of scope; to encompass them would require an even more cumbersome notation. The main methods for doing so in the examples are to use unique and distinct variable names, and only to declare variables at the top of a function body. In this case, the conversion of a problem piece of code merely requires some relabeling.
- The semantic rules presented below are not an exhaustive enumeration; rather, they are specimen examples which cover the more important points, and give a general view of the less complex rules, especially those dealing with sequential execution. It is not difficult to generate the latter from an intuitive understanding of the sequential behaviour of a language, and to convince oneself of their correctness.

4.2 Nomenclature

4.2.1 Use of Fonts and Labels

We will follow these conventions for fonts:

- Semantic entities, brackets, and labels will be set in $mathematical\ italics$
- Entities which relate directly to fragments of code will be set in `typewriter font`

Labels will be conventionally named to reflect what they represent:

- $expr$ for expressions
- $value$ for a value which cannot be further evaluated
- C for command sequences
- S for thread data spaces
- P for threads

- V for program state data spaces
- M for entire machine states
- Lower case letters such as x to represent pointer values

Where two labels on either side of a transition relation represent the same entity which has changed, the latter one will be primed, e.g. $expr$ and $expr'$. Where two or more labels occur for unrelated entities of the same type, they will be labeled by subscripts, e.g. P_1 and P_n.

4.2.2 Program States

Our model of an instantaneous program state consists of a *global data space* and a collection of *threads*, e.g.:

$$M \quad = \quad (\{y \mapsto 6, z \mapsto \ 'a'\}, [P_0, P_1, P_2]) \tag{4.1}$$

4.2.3 Threads

As in traditional operational semantics, we denote a thread state P by a tupled pair consisting of a command sequence C, and a data space S. A thread terminates with a value, which is the return value from the top level function, unless that function's return type is `void`.

$$(\texttt{x=3; printf("\%d",x);}, \{\texttt{x} \mapsto 6\}) \tag{4.2}$$

The local space is data *private* to that thread, rather than data *associated* with it, e.g. while performing a *map* operation over a BMF list data structure, there will be a thread associated with each element of the list, but the list will be in the shared global data space.

4.2.4 Data Spaces

There are two types of data space, each represented by identical notation. Each thread contains a private data space, while a program state contains a global data space for the shared data.

A data space consists of a mapping from names to values. These names are either variable names from the program code, or tags used to represent the connectivity offered by pointer types; the tags are a way of capturing this property without resorting to a data model at the memory addressing level.

4.2.5 Labeling, Substitution and Ellipsis

Any attempt to formally capture properties of a real programming language, rather than a toy example, is destined to create cumbersome notation and expressions. Our intention is to take whatever steps necessary, including reducing the formality of statements and proofs, to avoid becoming bogged down by notation. For our purposes, formality of notation is secondary to conciseness. We will use these forms of notational shorthand:

- Wherever an entity of any type appears in any context, it may be prefixed by a label, e.g. by writing

$$P = (\texttt{x=3; printf("\%d",x);}, \{\texttt{x} \mapsto 6\}) \tag{4.3}$$

  instead of 4.2 above. In subsequent references to the entity, we may then simply use the label.

- The "overloaded" operator $/$ is used to denote substitution into any data space, code, thread, list or state expression. The expression defined is the result of substituting the item on the left into the item on the right, where labeled items with matching labels are replaced rather than duplicated, but non-matching items are added.

In addition to labels as defined above, data space names are considered as labels for the purposes of substitution, e.g. $\{x \mapsto 6\}\ /\ \{y \mapsto 3\}$ represents $\{x \mapsto 6, y \mapsto 3\}$ but $\{x \mapsto 6\}\ /\ \{x \mapsto 3\}$ represents $\{x \mapsto 6\}$

Substitution is often used to indicate there may be a context on *both* sides of a transition relation, which remains unchanged, and the content of which is not relevant, e.g.

$$(\texttt{x=3}, \{\texttt{x} \mapsto 6\}\ /\ V) \ \rightarrow\ (\texttt{3}, \{\texttt{x} \mapsto 3\}\ /\ V) \tag{4.4}$$

Substitution of only one part of a tupled entity (thread or program state) is used where the meaning is clear.

- The ellipsis ... is used informally to indicate that part of an expression has been omitted for brevity

These shortened forms will only be used where they enhance readability without obscuring meaning.

4.2.6 Tags

We use a number of types of tags in the semantics. These obey the following conventions:

- They are always given in *italics*

  represent states; $expr$, $expr'$, $expr_1$ etc. represent expressions; C, C_1 etc. represent commands; $value$, $value_1$ etc. represent values (a value is an expression which cannot be further evaluated or simplified, and which does not depend on the state).

4.2.7 Transitions

We define four classes of binary transition relations:

- $\rightarrow$ represents a transition which occurs in a *single step* (i.e. is inherently atomic) and which occurs locally within a thread, without reference to the global state. It is overloaded to apply to expression evaluation as well as statement execution in the usual manner. This relation is a partial function, and so this type of transition is deterministic. This essentially captures the sequential semantics of the language.

- $\rightarrow^*$ is defined as the reflexive-transitive closure of $\rightarrow$, i.e. a deterministic multi-step transition. This operator is useful because it represents intuitively atomic operations in threads, i.e. sequential behaviour. It is equivalent to the τ-action found in CCS [33], and somewhat analogous to the concept of a "basic block" in the analysis performed by an optimising compiler for sequential code.

- $\Rightarrow$ is a relation on program states and represents an *atomic* transition in which a thread may refer to the global state. In a loose sense, it is a superset of the $\rightarrow$ relation, but for the fact they operate on different types (rule 4.7)

- $\Rightarrow^*$ is defined as the reflexive-transitive closure of $\Rightarrow$ and is in some sense a superset of each of the other three. This relation on program states encompasses all possible partial and complete execution paths of a program or fragment.

Unlike some semantics, these do not distinguish between *expression evaluation* and *statement execution* — they are both treated by rewrite rules in the same manner. This is in keeping with the general style of C derived programming languages.

We can express some of the properties connecting them formally by the following induction rules:

$$\frac{}{P \rightarrow^* P} \tag{4.5}$$

$$\frac{\begin{array}{rcl} P & \rightarrow^* & P' \\ P' & \rightarrow & P'' \end{array}}{P \rightarrow^* P''} \tag{4.6}$$

$$\frac{P \rightarrow P'}{(V,[P]) \Rightarrow (V,[P'])} \tag{4.7}$$

$$\frac{(V,[P]) \Rightarrow (V',[P'])}{(V,[P_0,...,P,...,P_n]) \Rightarrow (V',[P_0,...,P',...,P_n])} \tag{4.8}$$

$$\frac{}{M \Rightarrow^* M} \tag{4.9}$$

$$\frac{\begin{array}{rcl} M & \Rightarrow^* & M' \\ M' & \Rightarrow & M'' \end{array}}{M \Rightarrow^* M''} \tag{4.10}$$

4.3 Sequential Behaviour of Threads

The intention is that the sequential semantics of a single thread expressed by $\rightarrow$ and $\rightarrow^*$ will be as closely as possible identical to those of the original C language. The definition of $\rightarrow^*$ is implicit in our everyday understanding of C, and so we do not propose to define much of the sequential semantics explicitly, but to assume their existence and allude to them as required. Where an explicit definition specific to our purpose is appropriate, in particular to define atomicity (section 4.7) we will provide it.

$$\frac{S(x) = s}{(\texttt{y=x}, S) \rightarrow (\texttt{y=s}, S)} \tag{4.11}$$

$$\frac{(expr_1, S) \rightarrow (expr_1', S')}{(expr_1\texttt{+}expr_2, S) \rightarrow (expr_1'\texttt{+}expr_2, S')} \tag{4.12}$$

$$\frac{}{(value_1\texttt{+}value_2, S) \rightarrow (value, S)} \tag{4.13}$$

where $value$ is the result of an addition computation between $value_1$ and $value_2$. It should be noted that this addition will not always conform to the rules of arithmetic. Similar rules apply for each of the diadic operators `+ - / * | &`

$$\frac{}{(value;, S) \;\rightarrow\; (skip, S)} \tag{4.14}$$

$$\frac{(expr_1, S) \;\rightarrow\; (expr_1', S')}{(expr_1\texttt{,}\;\; expr_2, S) \;\rightarrow\; (expr_1'\texttt{,}\;\; expr_2, S')} \tag{4.15}$$

We define $TRUE$ to be "any value which when cast to an integer type is non-zero" and $FALSE$ to be "any value which when cast to an integer type is zero" as per the C language behaviour.

The triadic `?:` operator operates like a functional `if`:

$$\frac{(expr, S) \;\rightarrow\; (expr', S')}{(expr \;\texttt{?}\;\; expr_1 \;\texttt{:}\;\; expr_2, S) \;\rightarrow\; (expr' \;\texttt{?}\;\; expr_1 \;\texttt{:}\;\; expr_2, S')} \tag{4.16}$$

$$\frac{}{(TRUE \;\texttt{?}\;\; expr_1 \;\texttt{:}\;\; expr_2, S) \;\rightarrow\; (expr_1, S)} \tag{4.17}$$

$$\frac{}{(FALSE \;\texttt{?}\;\; expr_1 \;\texttt{:}\;\; expr_2, S) \;\rightarrow\; (expr_2, S)} \tag{4.18}$$

$$\frac{(expr, S) \;\rightarrow\; (expr', S')}{(\texttt{if}\;\;\texttt{(}expr\texttt{)}\;\; C, S) \;\rightarrow\; (\texttt{if}\;\;\texttt{(}expr'\texttt{)}\;\; C, S')} \tag{4.19}$$

$$(\texttt{if}\;\;\texttt{(}TRUE\texttt{)}\;\; C, S) \;\rightarrow\; (C, S) \tag{4.20}$$

$$(\texttt{if}\;\;\texttt{(}FALSE\texttt{)}\;\; C, S) \;\rightarrow\; (skip, S) \tag{4.21}$$

$$\frac{(expr, S) \;\rightarrow\; (expr', S')}{(\texttt{if}\;\;\texttt{(}expr\texttt{)}\;\; C_1\texttt{;}\;\; \texttt{else}\;\; C_2, S) \;\rightarrow\; (\texttt{if}\;\;\texttt{(}expr'\texttt{)}\;\; C_1\texttt{;}\;\; \texttt{else}\;\; C_2, S')} \tag{4.22}$$

$$\frac{}{(\texttt{if (}TRUE\texttt{) } C_1\texttt{; else } C_2, S) \ \rightarrow \ (C_1, S)} \tag{4.23}$$

$$\frac{}{(\texttt{if (}FALSE\texttt{) } C_1\texttt{; else } C_2, S) \ \rightarrow \ (C_2, S)} \tag{4.24}$$

$$\frac{(expr, S) \ \rightarrow \ (expr', S')}{(\texttt{while (}expr\texttt{) } C, S) \ \rightarrow \ (\texttt{while (}expr'\texttt{) } C, S')} \tag{4.25}$$

$$\frac{}{(\texttt{while (}FALSE\texttt{) } C, S) \ \rightarrow \ (skip, S)} \tag{4.26}$$

$$\frac{(expr, S) \ \rightarrow \ (TRUE, S')}{(\texttt{while (}expr\texttt{) } C, S) \ \rightarrow \ (C\texttt{; while (}expr\texttt{) } C, S')} \tag{4.27}$$

Since the `for` statement is merely syntactic sugar for a special case of the `while` statement we can define its semantics accordingly.

$$\frac{(expr_1\texttt{; while (}expr_2\texttt{) } \{C\texttt{; } expr_3\}, S) \ \rightarrow^* \ (skip, S')}{(\texttt{for(}expr_1\texttt{; } expr_2\texttt{; } expr_3\texttt{) } C, S) \ \rightarrow^* \ (skip, S')} \tag{4.28}$$

$$\frac{(expr, S) \ \rightarrow \ (expr', S')}{(\texttt{function(}expr\texttt{,}expr_2\texttt{,...)}, S) \ \rightarrow \ (\texttt{function(}expr'\texttt{,}expr_2\texttt{,...)}, S')} \tag{4.29}$$

$$\frac{(expr, S) \ \rightarrow \ (expr', S)}{\begin{array}{l}(\texttt{function(}value_1\texttt{,...,}value_n\texttt{,}expr\texttt{,...)}, S) \ \rightarrow \\ \qquad (\texttt{function(}value_1\texttt{,...,}value_n\texttt{,}expr'\texttt{,...)}, S')\end{array}} \tag{4.30}$$

$$\frac{(body, \{arg_1 \mapsto value_1, ...\} \ / \ S) \ \rightarrow^* \ (\texttt{return}value_r, S_B \ / \ \{arg_1 \mapsto value'_1, ...\} \ / \ S')}{(\texttt{function(}value_1, ...\texttt{)}, S) \ \rightarrow^* \ (r, S')} \tag{4.31}$$

In rule (4.31) above, S_B in the local state associated with the function body block (the automatic variables), and we assume the definition of function is of the form:

type `function` (*typearg*$_1$`,` ...)

body;

$$\frac{\begin{array}{c}(C_1, S) \rightarrow^* (skip, S_1) \\ (C_2, S_1) \rightarrow^* (skip, S')\end{array}}{(C_1; C_2, S) \rightarrow^* (skip, S')} \tag{4.32}$$

4.4 Concurrency

In our system there can be many threads across many processors executing concurrently. We view each thread as being composed of a sequence of atomic transitions, i.e. literally those operations which always proceed to completion unaffected by other changes in the environment, and can be loosely thought of as corresponding to a single machine instruction; this is distinct from any more coarsely grained concept of atomicity which we may choose to offer to the user, and implement through mechanisms such as semaphores.

We consider the possible transition sequences for the program as a whole to be those which can be composed by the arbitrary interleaving of the thread transitions. We can thus expand the single transition operator $\Rightarrow$ for program states as in rule (4.8)

It is important to note that because of the non-determinism, this rule only gives us possibilities, not consequences. We might also have

$$\frac{(V, [P_0]) \;\Rightarrow\; (V'', [P_0'])}{(V, [P_0, ..., P, ..., P_n]) \;\Rightarrow\; (V'', [P_0', ..., P, ..., P_n])} \tag{4.33}$$

4.5 Shared Data

We also define semantics for access to the shared data (denoted V in rule 4.8 above). As with the thread state spaces, we interpret V as a partial function from variable names to values.

$$\frac{V(x) = v}{(V, [..., (\mathtt{y} = \mathtt{x}, S), ...]) \;\Rightarrow\; (V, [..., (\mathtt{y} = \mathtt{v}, S), ...])} \tag{4.34}$$

$$\frac{}{(V, [..., (\mathtt{x=6}, S), ...]) \;\Rightarrow\; (\{x \mapsto 6\} \;/\; V, [..., (skip, S), ...])} \tag{4.35}$$

where in neither case does x exist in S

It should be noted that the shared data V above does not correspond to the data which would be globally held in a real parallel implementation, and similarly that the state spaces within threads in the semantics do not correspond to data which would be held locally on processor nodes in a parallel implementation. Data distribution in a real implementation is a complex issue, with great potential for influencing the efficiency and performance of different user programs.

4.6 BMF Operations

As noted above, the execution of a BMF operation proceeds by creating a thread on an appropriate processor for each sub-operation, and collecting up the result when the sub-threads terminate. We define these operations in terms of operators $\Rightarrow^*$ and $\Rightarrow$. Again we must keep in mind that these transitions are not consequent, and that the complete set of possible transitions is formed by rule induction over them.

4.6.1 The *map* operator

$$\frac{\begin{array}{r}[P_i = (\mathtt{f(l,0)}, \{\}), ..., P_{i+n} = (\mathtt{f(l,n)}, \{\})] \;/\; M \\ \Rightarrow^*(V', [P_i = (l'_0, W_i), ..., P_{i+n} = (l'_{i+n}, W_{i+n}]) \;/\; M'\end{array}}{[P_k = (\mathtt{map\ f\ l}, W_k)] \;/\; M \Rightarrow^*(V', [P_k = (l', W_k)]) \;/\; M'} \tag{4.36}$$

where $l = [l_0, l_1, ..., l_n]$, $l' = [l'_0, l'_1, ..., l'_n]$ and neither M nor M' contains any threads in the range P_i through P_{i+n} nor P_k.

4.6.2 The *reduce* operator

We use the non-deterministic property of our transition operators to make a simple definition of the action of *reduce*. Note the full implication of this in rule (4.38) which allows the reductions to proceed in any order, concurrently.

$$\frac{}{(\texttt{reduce op [}l_0\texttt{]}, W) \;\rightarrow\; (l_0, W)} \tag{4.37}$$

$$\frac{\begin{array}{c}[P_i = (\texttt{op(}l_i\texttt{,}l_{i+1}\texttt{)}, \{\})] \;/\; M \Rightarrow [P_i = (l_i', W_i)] \;/\; M' \\ [P_k = (\texttt{reduce op } l', W_k)] \;/\; M' \Rightarrow [P_k = (x, W_k')] \;/\; M''\end{array}}{[P_k = (\texttt{reduce op } l, W_k)] \;/\; M \;\Rightarrow\; [P_k = (x, W_k')] \;/\; M''} \tag{4.38}$$

where $l = [l_0, ..., l_{i-1}, l_i, l_{i+1}, l_{i+2} ..., l_n]$ and $l' = [l_0, ..., l_{i-1}, l_i', l_{i+2}, ..., l_n]$

4.6.3 The *foldl* and *foldr* operators

These operations are also known as *directed reduce* operations, as they perform a similar reduction, but maintain left (in the case of *foldl*) or right associativity for the operator *op*. These are inherently less efficient for parallel computation than a non-directed reduce as they impose a sequential ordering on the computations, but are semantically more precise.

If an operator *op* is commutative and has no side effects which are sensitive to the order of execution, then *foldl*, *foldr* and *reduce* are equivalent when used with it.

$$\frac{}{(\texttt{foldl op [}l_0\texttt{]}, W) \;\rightarrow\; (l_0, W)} \tag{4.39}$$

$$\frac{[P_k = (\texttt{foldl op [}l_0\texttt{,...,}l_n\texttt{]}, W)] \;/\; M \;\Rightarrow\; [P_k = (x, W')] \;/\; M'}{[P_k = (\texttt{foldl op [}l_0\texttt{,...,}l_{n+1}\texttt{]}, W)] \;/\; M \;\Rightarrow\; [P_k = (\texttt{op(}x\texttt{,}l_{n+1}\texttt{)}, W')] \;/\; M'} \tag{4.40}$$

$$\frac{}{(\texttt{foldr op [}l_0\texttt{]}, W) \;\rightarrow\; (l_0, W)} \tag{4.41}$$

$$\frac{[P_k = (\texttt{foldr op [}l_1\texttt{,}\ldots\texttt{,}l_n\texttt{]}, W)] \ / \ M \quad \Rightarrow \quad [P_k = (x, W')] \ / \ M'}{[P_k = (\texttt{foldr op [}l_0\texttt{,}\ldots\texttt{,}l_n\texttt{]}, W)] \ / \ M \ \Rightarrow \ [P_k = (\texttt{op(}l_0\texttt{,}x\texttt{)}, W')] \ / \ M'} \tag{4.42}$$

4.6.4 The *inits* and *tails* operators

In a purist BMF style, where the underlying algebra is, for example, a simple arithmetic system, these data replication operations are almost essential for certain computations. In our system they are less crucial, but are retained to allow computations to be expressed in a natural BMF style, which among other things provides data distribution cues to the implementation.

$$\frac{}{(V, [(\texttt{inits [}l_0\texttt{,}\ldots\texttt{,}l_n\texttt{]}, S)]) \ \Rightarrow \ (V, [(\texttt{[[],[}l_0\texttt{],[}l_0\texttt{,}l_1\texttt{],}\ldots\texttt{,[}l_0\texttt{,}\ldots\texttt{,}l_n\texttt{]]}, S)])} \tag{4.43}$$

$$\frac{}{(V, [(\texttt{tails [}l_0\texttt{,}\ldots\texttt{,}l_n\texttt{]}, S)]) \ \Rightarrow \ (V, [(\texttt{[[],[}l_n\texttt{],[}l_n\texttt{,}l_{n-1}\texttt{],}\ldots\texttt{,[}l_n\texttt{,}\ldots\texttt{,}l_0\texttt{]]}, S)])} \tag{4.44}$$

4.6.5 Other operations

Given a suitably rich underlying algebra, all BMF computations can be written in terms of one of a number of carefully chosen subsets of the operations; one such subset consists of just the two operations *map* and *reduce*. However, for efficiency it is necessary have available as primitives a number of operations which otherwise might have been constructed by composition. Choosing what BMF operations to include in or leave out of a particular working set is an open problem, and an area of ongoing research. We therefore reserve the right to bring in other operations to our system as the need arises; the semantic rules for their behaviour can be simply added to this set without affecting existing ones, and the style of the above definitions can and would be used to guide the construction of their semantics.

4.7 Sequential Semantics and Atomicity

Although we hope to take the sequential C semantics largely for granted, it is essential to define at least which types of sequential operations will be implemented atomically. The operations of principal interest here are concerned with access to shared or potentially shared memory; even if an implementation could permit a thread to interfere with e.g. an expression evaluation in another, we would disregard a program in which this occurred as being flawed (buggy).

Keeping in mind that our semantics only gives possibilities (i.e. that the relations $\rightarrow$ and $\Rightarrow$ are surjective but not functional), we define the granularity of our atomicity by giving rules for the $\rightarrow$ relation; the unwritten assumption is that transitions which do not fit these rules are not atomic.

The atomicity behaviour we will give for this initial semantics will correspond to the simple case where the implementation makes no special provision for atomicity, with the exception that it guarantees atomic read and writes for the base data types (`int`, `char`, `double`, etc.)

We might assume, where relevant, that the processor nodes have a conventional hardware environment that will allow reasonable operations to proceed in a single step; however, we are not concerned with minor details and will take steps to ensure that the validity of experimental and theoretical results is not obfuscated by questions about the implementation of different C data types on different hardware architectures.

So, for data of any base type we expect to have

$$\frac{S(x) = s}{(\texttt{y=x}, S) \;\rightarrow\; (\texttt{y=}s, S)} \tag{4.45}$$

$$\frac{}{(\texttt{y=s}, S) \;\rightarrow\; (skip, \{y \mapsto s\} \;/\; S)} \tag{4.46}$$

where in both cases s denotes a value, not an expression.

As a particular counter-example, the evaluation operation

$$\frac{V(x) = s}{(V, [(\texttt{x++}, S)]) \Rightarrow^* (\{x \mapsto s'\} \ / \ V, [(s', S)])} \tag{4.47}$$

is *not* atomic since the post-increment side effect involves a read operation and a write operation.

4.8 Proving Properties of Programs

The practical purpose to which we can put these semantics is to illustrate proofs about the behaviour of programs, in particular, to show that a program with a non-deterministic execution and side effects will produce a deterministic result.

Due to the inclusive rather than exclusive nature of the semantic rules, to prove that *all* possible sequences of execution starting in a given state lead to the same result, it is necessary to use an exhaustive case analysis, which amounts to running all possible executions of the program. The scope of this effort can however be reduced by considering individual elements of data structures and parameterising the proofs, and by proof techniques such as induction.

4.8.1 A Worked Example

As an example we consider the following program:

```
#include <stdio.h>
#define k 3

int list l[k];

void f(int BMFlist l, BMFindex i)

{
   int y=l[(i+1) % k];

   l[i] = (l[i]==y) ? l[i]+y : y;
}
```

```
int main()

{
   int i;

   for (i=0; i<k; i++) l[i]=6;
   map f l;
   for (i=0; i<k; i++) printf("%d\n",l[i]);

   return EXIT_SUCCESS;
}
```

This short program has the following behaviour: it sets each element of a list of integers to the same value, 6, then considering the list as a circular data structure, it concurrently executes a function on each element which compares it with its neighbour on one side, and adds it to the neighbour if they are equal or simply copies the neighbour if they are not. This "algorithm" involves access to data from other threads and side effects, but after a moment's thought, we can see that the output is in fact deterministic - all elements of the list are set to 12.

We now outline a proof of this deterministic outcome, for $k \geq 1$

We stipulate without proof that

$$\begin{gathered}(\{\}, [P = (\texttt{for (i=0; i<k; i++) l[i]=6;}\ \texttt{C}, \{\})]) \Rightarrow^* \\ (\{l \mapsto [6, ..., 6]\}, [P = (skip; C, \{i \mapsto k\})])\end{gathered} \tag{4.48}$$

This sequence of transitions is deterministic and consequential, since the program segment contains no concurrent operations.

If we consider the list as a circular, with l_{i+1} to the "right" of l_i and l_0 to the right of l_{k-1} then the expression $i + 1\%k$ denotes the index of the element to the right of element i. We refer to this as $r(i)$

Let $g(x, y)$ denote the expression `(x==y) ? x+y : y` We note that

$$g(6, 6) \rightarrow^* 12 \tag{4.49}$$

$$g(6,12) \rightarrow^* 12 \tag{4.50}$$

Consider the thread P_i operating on element l_i We have that

$$\frac{}{(V,[...,P_i = (\texttt{f(l,i)},\{\}),...] \;\Rightarrow\; (V,[...,P_i = (\texttt{y=}l_{r(i)}\;\; ...,\{\}),...])} \tag{4.51}$$

Clearly, its "neighbour thread" $P_{r(i)}$ has either written a new value for $l_{r(i)}$ prior to this read access, or it has not. If it has not, then the value of $l_{r(i)}$ will be 6. If it has, there will be a new value. Each thread performs its remote read access before it updates its local l_i, so clearly some P_i which is the first thread to perform the read must get a value of 6 for $l_{r(i)}$. We know that $g(6,6) \rightarrow^* 12$, and so the result written to l_i will be 12. Subsequent reads will also read a 6, until a thread reads an $l_{r(i)}$ which has already been updated, in which case it will read a 12. But $g(6,12) \rightarrow^* 12$ also, and so $g(l_i, l_{r(i)}) \rightarrow^* 12$ for all i.

We then have that

$$(V,[...,P_i = (\texttt{l[i]=12},S),...]) \;\Rightarrow\; (l_i \mapsto 12 \;/\; V,[...,P_i = (skip,S),...]) \tag{4.52}$$

for all i and so

$$\frac{\begin{array}{c}(\{l \mapsto [6,...,6]\},[P_0 = (\texttt{f(l,0)},\{\}),...]) \\ \Rightarrow\;\; (\{l \mapsto [12,...,12]\},[P_0 = (skip,S_0),...])\end{array}}{(\{l \mapsto [6,...6]\},[P = (\texttt{map f l},S)]) \;\Rightarrow\; (\{l \mapsto [12,...,12]\},[P = (skip,S')])} \tag{4.53}$$

In this proof we have relied heavily on arguments based on secondary techniques and argument rather than simply presenting a plethora of rule inductions. This is essential to control the combinatorial explosion of possible sequences that occurs from the non-determinism. This system of semantics is suitable for semi-automated theorem proving, allowing a more rigorous and straight-forward approach to be taken and leaving the task of verifying the bulk of the combinatorial case analysis to a machine.

4.9 A Tractable Analysis - Establishing Compositionality

As the above trivial example shows, the combinatorial explosion of possible states makes full analysis of a program of any size intractable; further more, even supposing that analysis were to be carried out, it is necessary to invoke congruence properties of the underlying operations to prove that all possible end results of the computation are equivalent.

The property of sequential operation semantics which we have lost by introducing parallelism is compositionality; in the case of two operation composed sequentially, it is sufficient to analyse them independently and then combine the results in order to determine the effect of the whole, i.e.

$$\frac{\begin{array}{c} P \rightarrow^{*} P' \\ P' \rightarrow P'' \end{array}}{P \rightarrow^{*} P''} \tag{4.54}$$

In the case of two program fragments composed in parallel, there is no equivalent, as they might potentially interact with each other. What is needed is to establish a rule, preferably one which can be evaluated at the syntactic level, which allows two fragments of code to be composed in parallel, knowing that they will not interact. This would allow us to e.g. establish the semantics of a map operation by merely examining the base algebra function, and scaling the result.

4.9.1 The Composition Rule

The following rule defines which program sequences may safely be composed in parallel:

- "Two or more program fragments may be executed concurrently if and only if any data item which is written by any one fragment is neither read nor written by any other"

4.9.2 Semantic Implications of the Composition Rule

In formal terms, we can say that, if the Composition Rule holds for two code fragments P_1 and P_2, then

$$\frac{\begin{array}{c}(V,[P_1]) \Rightarrow^* (V_1',[P_1']) \\ (V,[P_2]) \Rightarrow^* (V_2',[P_2'])\end{array}}{(V,[P_1,P_2]) \Rightarrow^* (V_1' \ / \ V_2',[P_1',P_2'])} \tag{4.55}$$

where it should be noted that

$$(V_1' \ / \ V_2') = (V_2' \ / \ V_1') \tag{4.56}$$

and also noted that if both P_1 and P_2 are deterministic, then so is $[P_1, P_2]$

The generalisation to more than two threads retains these properties.

4.9.3 Implications for the Programmer

What this means in the context of BMF operation is that any base algebra function which writes to a data item in non-local storage (including any list item) must only be used in operations like *map*, *reduce* or *filter* if it can be shown that that no other copy of the same function will read or write the same data item.

A sensible rule to stick to is that base algebra functions should only write their "own" element of the lists or lists across which a BMF operation is invoked, and should not read other elements of a list which they are writing to. In these cases, satisfaction of the Composition Rule can be determined algorithmically as discussed in 4.9.4 below.

A program which follows this rule will have the highly desirable property that the execution sequence as well as the outcome is totally deterministic.

4.9.4 Syntactic Checking for Composability

In many sensible programs, composability can be determined by syntactic analysis and simple index analysis of accesses to list or array elements; the analysis required is similar to the type used to determine if FORTRAN loops can be vectorised, or that used by an optimising sequential compiler to identify loop induction variables. In these cases, the analysis can be performed algorithmically, e.g. by a compiler.

An algorithm which can identify a significant proportion (but by no means all) of the safe cases for composing a function with itself in a *map* operation is as follows:

1. Identify all write accesses to non-local variables, be they global single variables, lists or arrays. Initially, mark all such variables as non-safe.

2. Identify any read accesses to these variables which are written. Ignore those variables which are only read and never written.

3. For each of the array or list variables, check all references to that variable. If (a) all such references can be determined to be semantically equivalent expressions, i.e. to evaluate to the same index, and (b) are dependent on the index within the *map* operation in a way which guarantees an injective relation from function instances to array/list elements, then this variable can be marked as safe.

   The simplest determination of this is when the *map* index *is* the index used in all the references, but simple linear functions of it could be checked with suitable algorithms.

4. For each of the remaining non-local variables, whether they are single variables, arrays or lists, mark it as safe if it can be determined that the control flow to all references will only be satisfied for one, single value of the *map* index.

5. If all variables have been marked as safe, then a *map* operation over this

function is safe. If not, it may be safe, but we cannot prove it with this simple algorithm.

Consider the following two implementations of a simple, idealised one-dimensional Jacobian relaxation, one single-buffered and one double-buffered - the single buffered one is shown first:

```
/* Single buffered version */

#include <stdio.h>

#define SIZE 100
#define MIDDLE (SIZE/2)
#define COLD -50.0
#define HOT 200.0
#define TOL 0.000001

float list t[SIZE];
float prev,diff;

void zero(float list a, int i)
{
  a[i]=0.0;
}

void f(float list a, int i)
{
  if (i == MIDDLE) prev = a[i];

  if (i == 0) {
    a[i]=(a[i] + COLD) / 2.0;
  }
  else {
    if (i == SIZE-1) {
      a[i]=(a[i] + HOT) / 2.0;
    }
    else {
      a[i]=(a[i-1] + a[i] + a[i+1]) / 3.0;
    }
  }

  if (i == MIDDLE) diff = abs(a[i] - prev);

}
```

```
void iterate(void)
{
  map f t;
}

int main(int argc, char **argv)
{
  int i;

  map zero t;

  /* force at least SIZE iterations so tolerance check fails the
     first time */

  for (i=0 ; i<SIZE ; i++) iterate();

  while (diff > TOL) iterate();

  printf("Result: %f\n",t[MIDDLE]);
}
```

This single-buffered implementation has an obvious (to us) race condition - let us follow the steps of the algorithm to analyse the `map f t;` statement:

1. We have 3 variables referenced in `f` - `prev`, `a` (i.e. `t` and `diff`
2. We have read references to `t` and `prev`
3. We have references to `t[i]`, `t[i-1]` and `t[i+1]` whose indices (unsurprisingly) cannot be determined to be equal
4. All the references to `prev` occur for a single constant value of the index, and so are found safe. Likewise for `diff`
5. We have not proven the function to be safe, so we assume it is unsafe (which it happens to be)

Now consider the same program, but modified to use a double buffered implementation:

```
/* Double buffered version */

#include <stdio.h>
```

```
#define SIZE 100
#define MIDDLE (SIZE/2)
#define COLD -50.0
#define HOT 200.0
#define TOL 0.000001

float list t1[SIZE];
float list t2[SIZE];
float prev,diff;

void zero(float list a, int i)
{
  a[i]=0.0;
}

void f(float list a, int i, float list b)
{
  if (i == MIDDLE) prev = a[i];

  if (i == 0) {
    b[i]=(a[i] + COLD) / 2.0;
  }
  else {
    if (i == SIZE-1) {
      b[i]=(a[i] + HOT) / 2.0;
    }
    else {
      b[i]=(a[i-1] + a[i] + a[i+1]) / 3.0;
    }
  }

  if (i == MIDDLE) diff = abs(b[i] - prev);

}

void iterate(void)
{
  map f(t2) t1; /* in f, a==t1 and b==t2 for this line */
  map f(t1) t2;
}

int main(int argc, char **argv)
{
  int i;

  map zero t1;
```

```
  /* force at least SIZE iterations so tolerance check fails the
     first time */

  for (i=0 ; i<(SIZE/2) ; i++) iterate();

  while (diff > TOL) iterate();

  printf("Result: %f\n",t2[MIDDLE]);
}
```

The analysis this time is slightly different - we perform it only for the first of the two statements in `iterate`:

1. We have 3 write references, `prev`, `b` (i.e. `t2`) and `diff`
2. We have read references to `t1`, `t2` and `prev` - we ignore `t1`
3. All of the references (read and write) to `t2` are to `t2[i]` so we declare`t2` to be safe
4. As before, `prev` and `diff` are safe
5. All write variables have been marked safe, so we declare this composition under *map* to be safe

The same analysis applies to the second statement of `iterate`, and a trivial one takes care of `zero`, and so we can find the whole program to be deterministic.

Similar algorithms can be developed for *reduce*, *filter*, *zip*, etc.

Note the fact that we needed to know the instantiation of the arguments `a` and `b` at compile time and ensure that they were separate in order to make this determination. Higher level determination of composability, when base algebra functions invoke concurrent BMF operations within themselves, could take place in one of two ways:

- The analysis could proceed top down, with recursive invocations of the algorithms corresponding to each BMF operation. The arguments to each BMF operation can either be determined to be syntactically separate, or their separation unknown. If it is unknown, we face the possibility that two arguments of the same type could represent the same instance, and must analyse assuming this to be the case. Because there is no pointer manipulation allowed in the language, we need not consider the case where there is an overlap.

- To perform a bottom up, modular analysis, we need to identify arguments which could potentially reference the same data structure, analyse for each possible combination, and identify those which are safe, and store this information as part of the analysis of the function. We must also include information on which type of reference is made to each argument, and to any global variables. This can then be incorporated into the analysis of the calling function.

As with an optimising or vectorising compiler, the analysis can be extended by adding code to check for more and more cases, but can never be complete. In a practical situation, a compiler might flag a warning when an analysis of an operation fails to determine that it is safe, and offer the option to override this with `pragma` directives or options, much as vectorising compilers allow the programmer to use directives to assert the safety of vectorising a loop where the compiler cannot determine its safety algorithmically.

Chapter 5

Performance Studies

This chapter investigates the performance of the various BMF operations, both singly and in context, and how that performance is affected by factors such as list size, data distribution and the granularity and uniformity of the user supplied base algebra code. We aim to show that, while the ultimate performance is less than that which could be achieved by hand crafted low level code, it is sufficient to justify the use of this approach when the overall costs are considered.

5.1 A Basic Implementation of the BMF Paradigm

In order to properly quantify the performance of the BMF operations, it was necessary to produce a workable implementation of the BMF language under study which would run in a plausibly realistic fashion on real systems. In a real world situation, the correct way to implement the language to achieve maximal performance on a particular system would be to produce a fully targeted compiler which could directly utilise the communication hardware of that system.

For the purpose of a few experiments to gather relative performance data, the effort of producing a full compiler is not justified, and there is the constraint that the resulting compiler would only be suitable for one hardware platform. The approach which was taken instead was to try to produce something fairly portable, based upon a lower level standard and using generic compilers already available.

The system which was implemented provides the BMF operations in a library call manner within a standard C language environment, using a fixed number of threads of control, and message passing for communication and parallelism. This necessitated a number of compromises which are outlined in section 5.1.1.

5.1.1 Living Without Dynamic Threads

A straightforward implementation of the prototype language defined in section 3.2 and one which would offer maximal concurrency would be to create a lightweight thread for each base algebra function invocation within each BMF operations. With suitably efficient thread handling this might prove quite effective.

Unfortunately, most parallel computers offer fairly primitive run time systems, which often lack features such as dynamic thread creation and virtual memory. The application programmer is supposed to target their code for a specific platform, and tune it to gain maximum efficiency. This objective is somewhat at odds with our one of creating portable parallel code!

The Cray T3D system used for the performance tests is typical of such systems, and only offers a single thread of control on each processor.

The compromise which was implemented in the prototype run time system was to use what is in effect a re-entrant message handler and scheduler, allowing all of the components of a BMF calculation on each node to run within a single thread of control.

The principal constraint which this places upon BMF programs is due to the fact that it imposes an order of execution upon invocations of base algebra functions which might otherwise be concurrent. Any program which attempts communication between these operations could therefore potentially deadlock this implementation; however, such a program would violate the proposed guidelines for good structured programming in section 4.9.

5.1.2 MPI

MPI [35] is a standard interface which can be used to write portable parallel programs in a message passing paradigm. An MPI program is written in a sequential, imperative programming language, with calls to an MPI subroutine library. The initial standard defines interfaces for C and FORTRAN 77.

Parallelism is achieved by running one copy of this program on each processing node; the first thing that each of these processes typically does is to determine its own identity, and then branch to the appropriate part of the code, thus creating a task farm or whatever work sharing paradigm the programmer has implemented.

Communication between the processes is achieved by matched pairs of send and receive calls to the MPI subroutines, which support an exhaustive variety of communication patterns.

As an aside, it is worth noting that the MPI-2 draft standard expands on this and allows dynamic creation of threads, which would be ideally suited to this application, but this standard was not yet widely implemented at the time of these experiments.

The initial intention with the prototype implementation was to use a subset of MPI to support simple message passing between the threads of control on each node. However, initial performance tests with MPI on the Cray T3D system being used revealed that there were occasional serious anomalies and inconsistencies in the performance achieved for fine-grained operations.

5.1.3 Cray Virtual Shared Memory Library

An alternative version of the implementation using the Cray low-level virtual shared memory library [12] was developed. Since the functions offered by this library map very directly onto the T3D hardware, the performance of the resulting implementation was much more regular and predictable.

Communication between processing nodes was achieved using a simple first-in first-out rotating buffer scheme, implemented using shared memory writes and atomic spin-locks.

5.1.4 Implementation Details

The implementation of the functionality of the BMF operations using plain C library functions is a good deal less aesthetic than the syntax presented in section 3.2, and a fully detailed description of it is not of great interest. A brief outline of some details is merited, to offer a subjective insight into its inner workings.

5.1.4.1 The `list` Data Type

The layout of the parallel list data structure faithfully follows the one proposed in section 3.3. The actual C type used to implement a list handle is:

```
struct BMFlist {
  void *base;
  int esize;
  int n;
  int firstnode;
  int block;
};
```

Note the extra item, which is the element size - the type system of the C language is not sufficiently rich to allow the BMF operations and list handling functions to be polymorphic, so the solution adopted was to encapsulate list and base algebra functions using the `void` type.

This has the unfortunate side effect of making the system completely non-type-safe, but this is in keeping with its rôle as a prototype.

5.1.4.2 Implementation of Base Algebra Functions

The invocation of a base algebra function on a remote processor requires something akin to a remote procedure call. In order to implement this, function pointers for these functions are stored in an array on each node, so that they can be referenced by number.

Again, due to constraints of the type system, it is not possible for the base algebra functions to have a range of return types, so they are passed additional arguments giving the location of storage for any return values, and the return type as far as the compiler is concerned is `void`.

Thus the following declaration of a base algebra function for use with *map*

```
int inc(int list input, int index)
```

becomes

```
void inc(BMFlist *input, int index, BMFlist *output)
```

5.1.4.3 Implementation of BMF Operations

In this implementation, the programmer's calling routine must invoke a BMF operation using a library call, and pass the reference number of the required base algebra function and handles for the list(s) as arguments.

This BMF library function issues the messages required to start off the parallel operations, and then invokes the re-entrant scheduler as a subroutine. The scheduler will check for incoming messages which require work to be done on the local node, which will conceivably include one of the ones it has just issued. It will only return to the BMF library function which called it once these have been completed and once all remote nodes have replied to the outstanding requests.

The BMF library routine then finishes up by deleting any temporary data structures no longer needed and exiting, returning control to the user function which called it.

The other half of a remote invocation occurs when the scheduler on a node receives a message asking it to perform its share of a BMF operation. The scheduler invokes an agent subroutine which in turn makes calls to the appropriate base algebra operations, one after the other. These may in turn invoke other BMF operations. The agent function replies to the node on which the call originated using an appropriate message and then exits.

In this way, there may be at any one time several operations queued at a particular node, and several copies of the scheduler on the stack. It is perfectly possible for two nodes to issue two different requests simultaneously, and for their base algebra functions to be interleaved. If the guidelines in 4.9 are followed then this is not an issue, as there should be no data dependency between them.

5.1.4.4 Overall Structure

The operating system on the Cray T3D starts a parallel program by loading identical copies of the program code on each node, and starting them off (i.e. in C, calling the `main` function) simultaneously.

To achieve the desired behaviour in the prototype BMF system, it is arranged that the `main` function is actually one of the BMF library functions. Each copy determines which node it is running on, and if it is node 0, it invokes a single copy of the user's main program, and otherwise it invokes the re-entrant scheduler which idles until request messages begin to arrive.

5.1.4.5 Overheads

The one major and inescapable consequence of this library style of implementation is that it introduces the overhead of several layers of subroutine calls for even a fairly simple operation. In particular, this pushes up the minimum practical granularity of a base algebra function quite considerably.

5.2 Performance of Basic Operations

The first thing to quantify when examining the performance of this BMF based approach is the performance of individual operations taken in isolation. This is of course highly dependent on the architecture of the parallel machine they are being run on, and also relates strongly to the choices made in implementation and their associated overheads, for instance the method of data distribution defined in section 3.3.

The BMF operations can be grouped into categories, according to the communication pattern generated by the data movement implicit in the operations themselves. Of course, we cannot legislate for patterns of data references invoked by the underlying functions created by the programmer; if the program has been developed fully in the spirit of the approach, then the communications arising from such references should not fundamentally alter the asymptotic cost of the operations.

Among the fundamental BMF operations, we can identify four principal categories:

- The *map* operation stands alone since it is "flat" in the sense that it involves no communication other than the overhead of invoking it. This is intuitively optimally parallel. There is potential within the data distribution model we have proposed for obtaining almost similar performance from the *zip* operator; the data for one of the two lists is first migrated in an implementation-dependent optimal fashion, to align the lists, and then the operation can proceed in the same communication-free manner as *map*.
- The *reduce* operator has the next largest potential for regular concurrency. The tree of reduction operations can be implemented in a series of waves, bottom up, to yield a single value. We should not aim for a completely balanced tree, but rather perform reductions on blocks of list elements within a processing node first, followed by reduction between nodes. This latter reduction offers scope for communications other than "nearest-neighbour" and is one of the areas which a relatively dense architecture may be exploited. A commutative variant of *prefix* could be implemented with a similar strategy and level of concurrency.
- The majority of the remaining active operations, such as *foldl* and *foldr* involve a sequential element, and we are cornered by Amdahl's law. The only scope for concurrency in the general case is in the overheads; avoiding these is a matter which must be resolved at the algorithm design stage.

- The final category is operations principally intended for data migration. This includes not only pure migration and replication operators like *inits* and *tails* but also *filter* which does contain an active underlying function. The performance of these is highly implementation specific, and depends entirely on the connectivity available in the machine and how the implementation makes use of it. This category opens a number of performance tuning implications; for example, *inits* and *tails* can be trivially implemented by local copying, but this leaves the distribution of data across the machine unbalanced. Expending communication overhead during these operations to achieve a more even distribution may yield future performance benefits.

A performance issue which is not addressed in [43], and which is of stronger significance in an imperative programming paradigm than in a functional one, concerns the possibility of there being highly variable costs and data referencing behaviours in the underlying functions. If this situation does occur, it may well be that attempting to distribute computation according to the layout of the data is not an applicable strategy. In this case it may be more suitable to rely on randomisation and a task farm approach to achieve better load balancing.

5.2.1 Timing and Interpretation of Results

All timings were taken against real (i.e. "wallclock") time. This is the only appropriate measure to use for real performance, since the concept of "CPU time" as measured by an operating system on a multi-user system is an arbitrary function of its scheduler. Since the Cray T3D system dedicates a physical partition of the machine to each parallel program, and since the interference between these partitions is usually quite small, the wallclock times are in fact very consistent and repeatable.

The timing library call used derives its values from the hardware CPU clock ticks on the T3D system, and appears to have a resolution of within 100us and a stability which is a fraction of 1 percent.

For purposes of interpretation, it was decided that it was enitrely correct to perform a number of runs of each experiment, and to take the minimum time, rather than the average. The justification for this is that the experimental runs are deterministic in their nature, as there is no randomisation in the parallel interconnect, and therefore any significant variation in timing can only be due to interference from another partition. This interference can occur when communication passes through other partitions and back, taking advantage of the toroidal topology of the machine, or when another program performs I/O which goes though the interconnect of the local partition.

5.2.2 Fully Concurrent Operations - *map*

The sub-operations invoked by map are semantically independent, in the sense that the semantics makes no guarantees about order of execution, thus they can be performed totally concurrently. However, because we are working in an imperative domain, and we make available the expressive power for a sub-operation to reference global data, it is possible to construct pathological cases where a map operation effectively becomes sequential; equally, it is possible to envision perfectly reasonable operations which deliver far from optimal speedup. This is one of the many situations where it is necessary to design an algorithm in the BMF style from the top down to ensure good performance.

5.2.2.1 Test Program for *map*

For the purposes of testing, we have constructed the following BMF program, and implemented its equivalent using the prototype library; the base algebra function consists of a loop which consumes a controlled amount of processor time, allowing a variety of granularity to be simulated. Note the operations which write back to the input list, thus ruling out the possibility of an optimising compiler removing the loop!

It can be seen by inspection of the source code (or indeed, the object code) that this loop requires four operations per iteration - an addition, an increment, a test and a branch. This corresponds to four machine instructions on a typical modern RISC processor.

All subsequent references to granularity (k) are in terms of these "four operation" units.

```
void f(int list l, int i)

{
  int i,j,k;

  j=l[i];
  k=0;
  for (i=0; i<j; i++) k += i;
  l[i]=k;
  l[i]=j;
}

int main()

{
  int i;
  int list l;

  /* length, iterations, and delay  parsed
     from command line arguments */

  newlist(l,length);
  for (i=0; i<length; i++) l[i]=delay;

  /* start timing */

  for (i=0; i<iterations; i++) map f l;

  /* stop timing */
}
```

The program allows for multiple iterations of the *map* operation to permit the timing to be scaled for more accurate readings, and to allow for elimination of systematic error due to timing startup. In typical runs, 100 iterations were used and the result divided.

In the results below, k is used to refer to the granularity of the base algebra function, l is the length of the list and n is the number of processor nodes.

Timings, where given, are in microseconds. Unless stated otherwise, the blocking factor of the list storage is l/n, which is usually optimal in simple cases.

5.2.2.2 Scalability

The first simple property to consider is scalability; for a baseline we will take a list of 1024 elements, and a granularity of 100, and try the operation on different numbers of processors. The results are shown in figures 5.1 and 5.2 as execution times and relative speedups respectively.

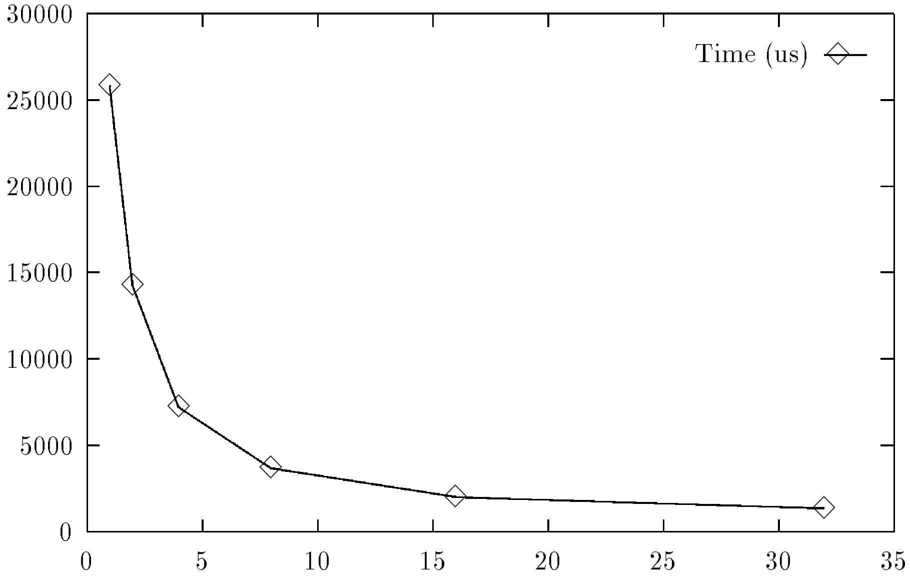

Figure 5.1: Execution time for *map* v. number of processors n ($l = 1024$, $k = 100$)

As can be seen from these graphs, the apparent speedup is quite respectable even for this relatively small amount of data and computation. A note of caution must be sounded - the amount of actual computation going on is larger than readily apparent, as discussed in section 5.2.2.3. The results for the absolute running time are therefore pessimistic compared to what might be achieved in a "production" implementation, and those for the speedup are optimistic. This is quantified to an extent by the granularity tests in section 5.2.2.4 below.

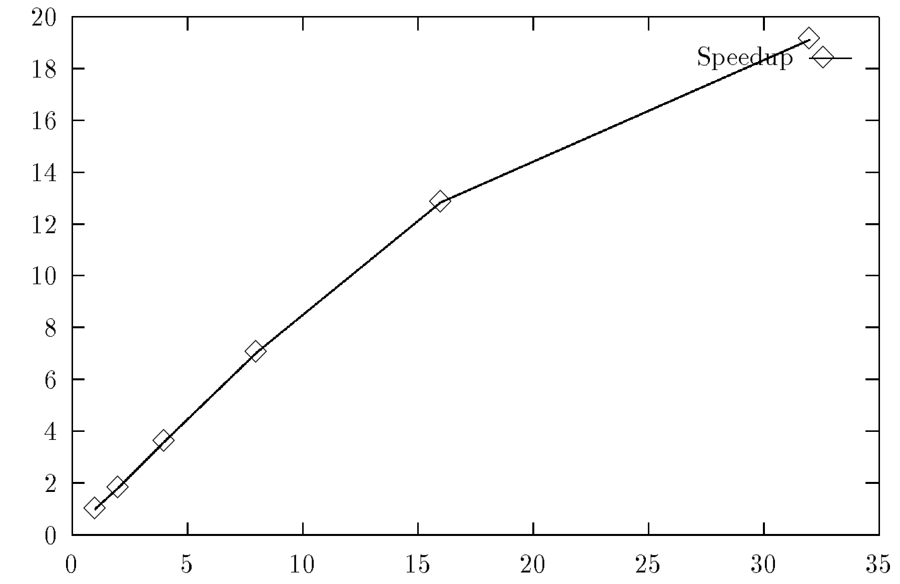

Figure 5.2: Relative speedup v. number of processors n - ($l = 1024$, $k = 100$)

5.2.2.3 Overhead Tests

To attain some measure of the baseline overhead, we can measure the execution time when there is little or no data compared with the time for a significant amount. Figure 5.3 shows the execution time for a variety of list sizes, on a fixed number of processors ($n = 8$) and with a fixed granularity ($k = 100$).

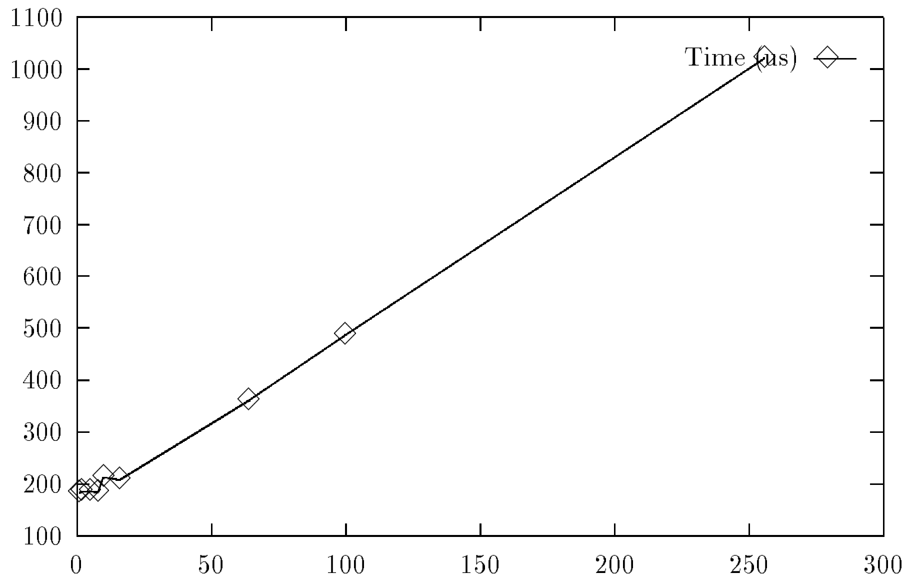

Figure 5.3: Execution time v. length of list l - ($n = 8$, $k = 100$)

Two points should be noted from this data; firstly, there is no difference at all between the time taken for a 1 item list and that for an 8 item (i.e. 1 item per processor) list, since the computation is regular in structure.

Secondly, the time taken for 256 elements (32 per node) is approximately 5.5 times greater than that for 1 to 8 elements (1 per node); we can therefore quantify the total overhead time at about 7 times the cost of processing a single element, or equivalent to a granularity factor k of about 700.

5.2.2.4 Granularity Tests

We now attempt to quantify the granularity required to attain reasonable performance with this implementation; to do so we will first look at how the execution times grow with the granularity for a fixed list size and fixed number of processors. Graph 5.4 shows this for the case $l = 256$ and $n = 8$.

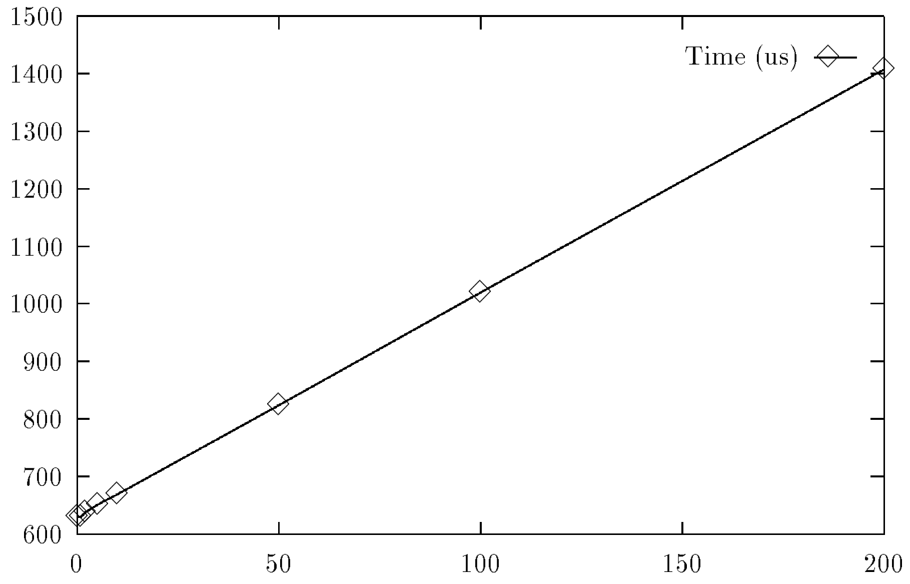

Figure 5.4: Execution time v. granularity k ($l = 256$, $n = 8$)

As we would expect, the execution time rises more or less precisely linearly with granularity, with a baseline of around 650us - this baseline does include the invocation of the user function, so gives a true representation of all overheads including the significant one of the procedure calls being made due to the nature of the implementation.

We then consider the granularity $k_{1/2}$, which we define as the granularity required

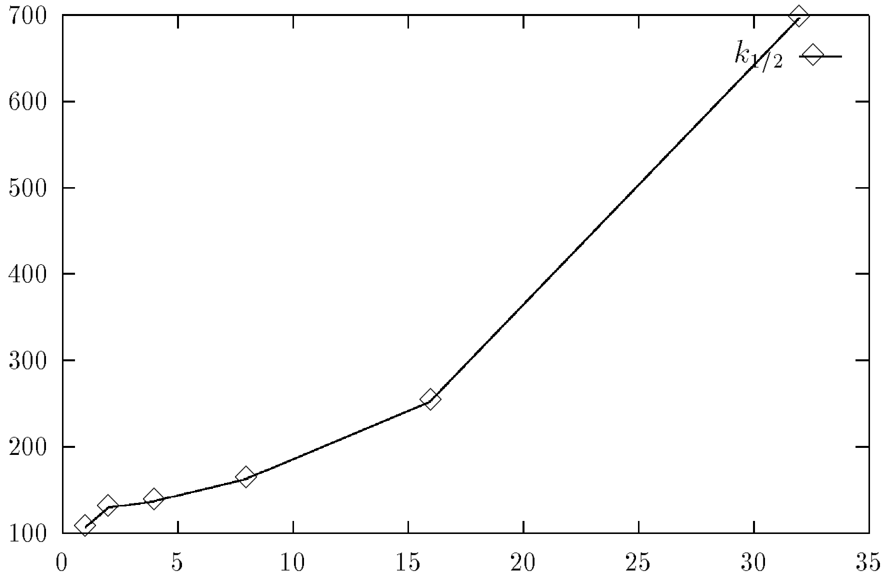

Figure 5.5: Half-efficiency granularity $k_{1/2}$ v. n ($l = 256$)

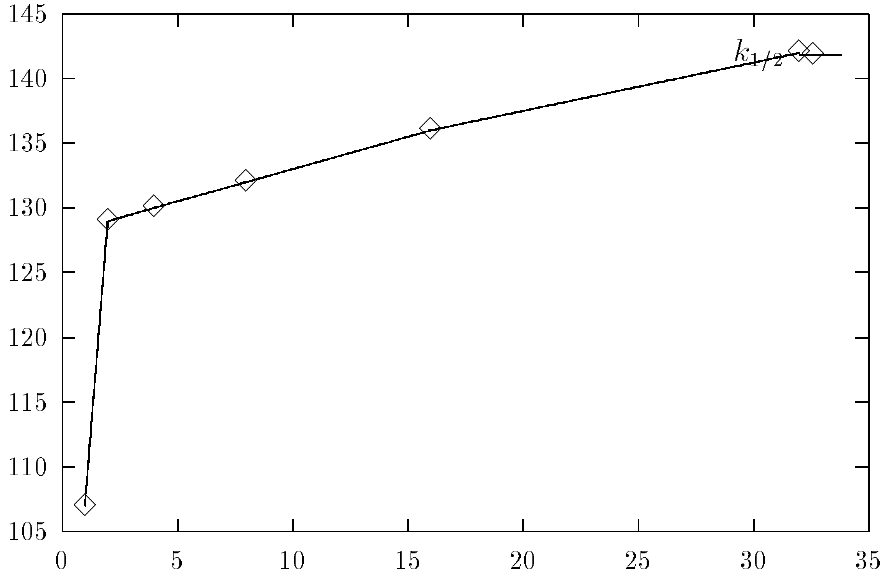

Figure 5.6: Half-efficiency granularity $k_{1/2}$ v. n ($l = 256n$)

to reach the situation where 50 percent of the time taken is actual computation. An approximation to this was measured by determining the value of k required to make the total execution time double that taken when $k = 0$. Graph 5.5 show this for a range of machine sizes n with a fixed list size $l = 256$; graph 5.6 shows how it scales when the number of elements per node l/n is held constant instead.

For the relatively small list size in graph 5.5 the granularity required rises markedly with the number of processors, as the overheads of distributing begin to dominate the increasingly small per-processor workload.

In the case of graph 5.6 where the workload per processor is constant, the global overheads are not increasing by very much as the number of processors rise; the only additional cost to the total runtime is the sequential issue of requests and collation of replies by the processor running the main program code. The difference in this cost between 2 processors and 32 processors is only a "k-value" of 13, i.e. corresponding to a sequential cost to distribute work of less than 450 machine instructions per task.

The seemingly anomalously low figure for the case $n = 1$ is explained by the fact that no actual communication is taking place, and execution occurs locally and sequentially.

5.2.2.5 Effect of the Blocking Factor

Thus far, we have ignored the impact of the blocking factor, which was introduced in the data distribution to allow aggregation of contiguous list items to improve locality of reference. Clearly, on a pure *map* operation like the one under test, this makes no difference; what we are measuring is a small difference in overheads. This is shown in graph 5.7

There is a small but significant additional overhead introduced by "odd" (non power of two) blocking factors.

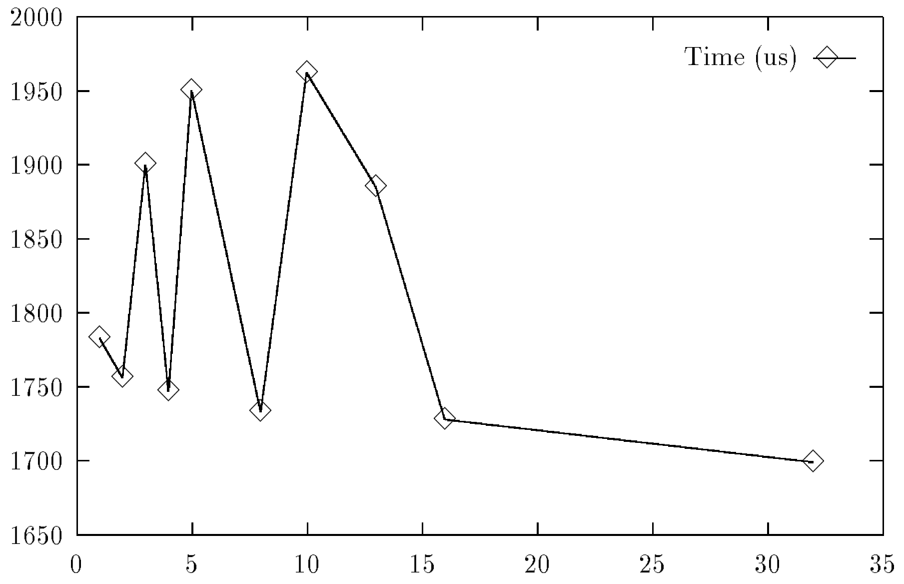

Figure 5.7: Variation in execution time with blocking factor ($l = 1024$, $n = 32$, $k = 100$)

5.2.3 Tree Structured Operations

The most commonly used implementation for efficient execution of the *reduce* operator can be thought of as having a tree structure. The sub-operation is first invoked on concurrently on pairs of adjacent list elements, and then on pairs of the results, and so on until a single element is returned. This can be shown graphically as a binary tree, with sub-operations as internal nodes and the original list elements as the leaves. The execution can be performed asynchronously, with results used as they are combined.

Using the data distribution we have proposed, the prototype implementation of *reduce* takes place in two distinct phases; the first performs a reduction on the contiguous blocks of data held on individual nodes, and the second combines these results.

We will test the *reduce* operator using the following program:

```
int f(int list l, int i1, int i2)

{
  return l[i1]+l[i2];
}
```

```
int main()

{
  int i;
  int list l;
  int result;

  /* length and iterations  parsed
     from command line arguments */

  newlist(l,length);
  for (i=0; i<length; i++) l[i]=i;

  /* start timing */

  for (i=0; i<iterations; i++) result=reduce f l;

  /* stop timing */
}
```

Graph 5.8 shows the execution time for this operation for a fixed list size of 1024, as the number of processors increases. Graph 5.9 shows the same data in the form of a relative speedup.

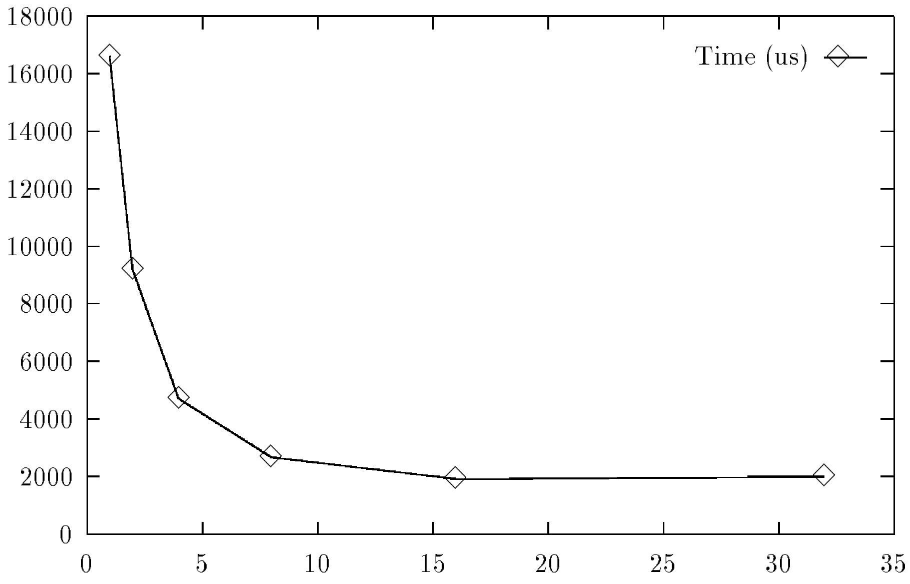

Figure 5.8: Execution time for *reduce* v. n ($l = 1024$)

Although there is a significant speedup as the number of processors increases, the gains are not as large as for a "fully" parallel operation like map, and at 32

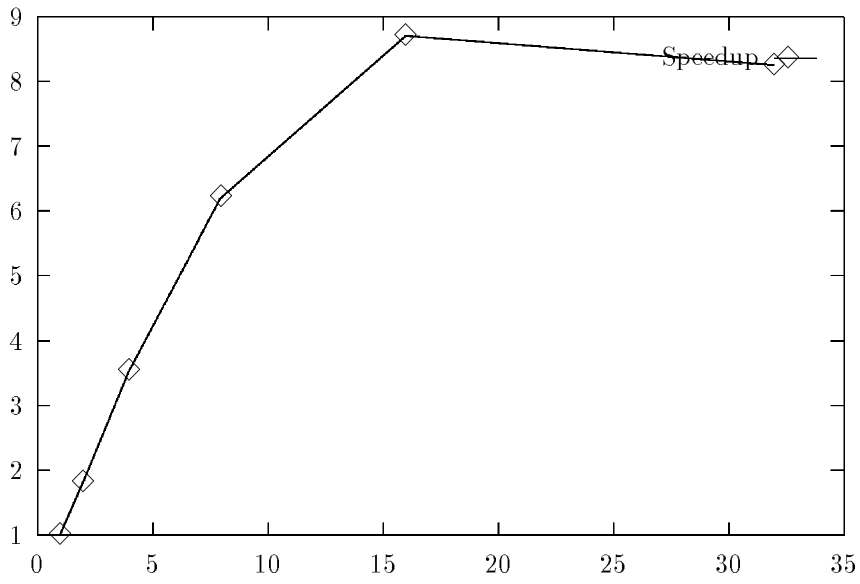

Figure 5.9: Speedup for *reduce* v. n ($l = 1024$)

processors we see the speedup begin to fall again as the overheads outweigh the gains.

5.2.4 Linear Operations

The *fold* operations are semantically similar to reduce, but make the additional constraint that the order of evaluation is guaranteed. What this means is that the operation changes from one which has a relatively high potential for concurrency to one which is totally sequential. The one BMF advantage we do retain is locality of reference; in common with the *reduce* operator, in the general case the majority of data references occur to elements stored on the same node, and all inter-node communication is between topologically adjacent processing nodes.

For a direct comparison we use the same example computation as above , but with a *foldl* operation instead of *reduce*.

The graphs 5.10 and 5.11 show the execution time and “speedup” for the equivalent operations to those tested with *reduce* above.

As we would have expected, the cost in total time for performing the *fold* operation actually *increases* with the number of processors, though admittedly only one

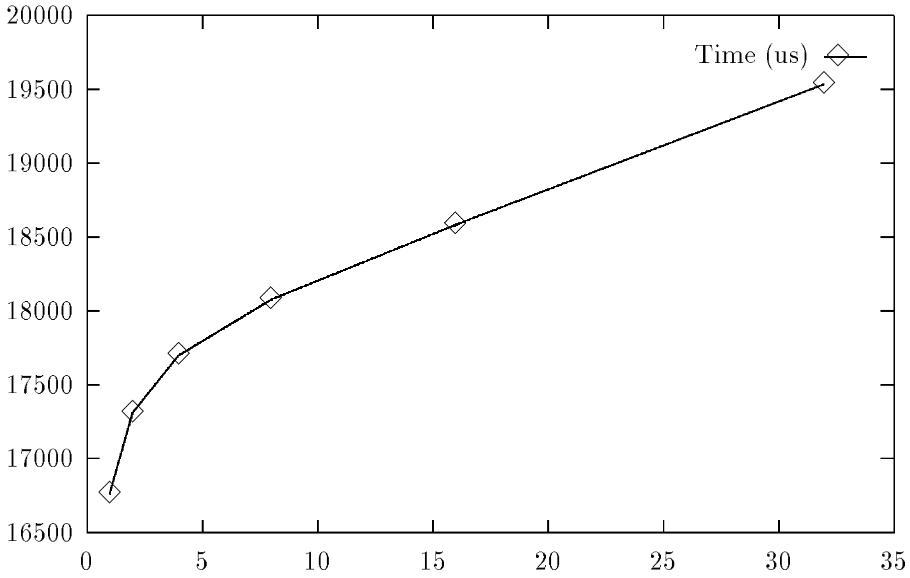

Figure 5.10: Execution time for *foldl* v. n ($l = 1024$)

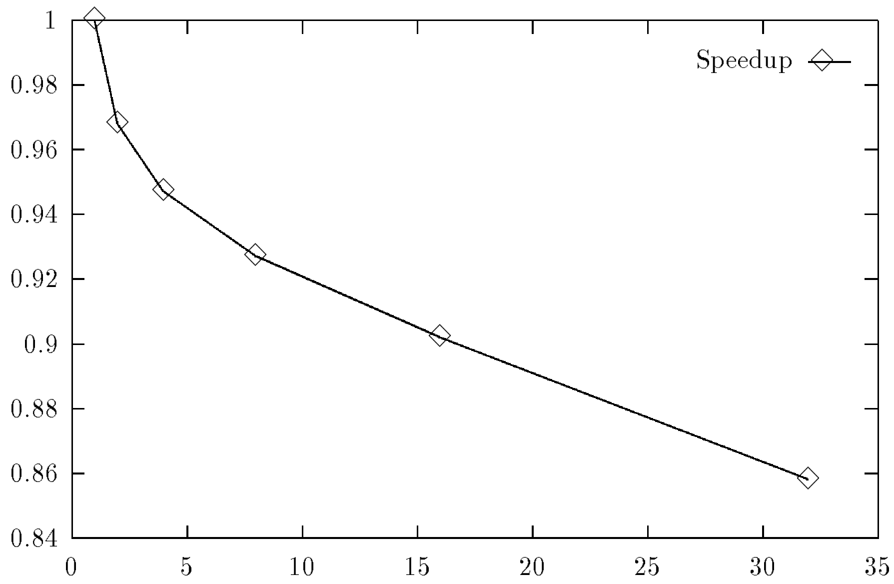

Figure 5.11: “Speedup” for *foldl* v. n ($l = 1024$)

processor is being used at a time and so there are resources which could be used if other operations were proceeding concurrently. Note the fact that even on a single processor this operation is slightly slower than *reduce* - there is no underlying reason for this so we may surmise it is an artifact of the implementation.

The moral of the story here is that a fold operation is sequential and hence very slow, and can have a serious impact on the performance of an otherwise parallel program. The way to tackle this is to try to eliminate the need for sequential operations like fold at the algorithm design stage, even if this does necessitate a slightly more complex data structure and more complex base algebra functions.

5.2.5 Data Movement and Replication Operations

The BMF operations which correspond to restructuring or substructuring of lists translate to data movement and copying in the imperative domain. A key way in which they differ from the active operators is that they can be composed of operations with no user supplied code and simple semantics. Because their behaviour is effectively hidden from the user, they allow considerable scope for performance tuning in the implementation.

Since in general they will produce new copies of lists, they can indirectly exert considerable influence the performance of the user program by the particular data distribution they produce (recall that the prototype distribution we are using has three parameters - section 3.3) and its impact on future load balancing.

The operation we will consider here is one of the more complex ones, namely *filter*. To implement this requires several phases:

- The user supplied base algebra function is invoked for each element of the list, in a manner akin to *map*. This yields truth values for each element which correspond to whether it should be kept or discarded, which are held in temporary workspace.
- Next, the numbers of positive results for each block of the original list are counted. This too is fully parallel.

- These are then collated to determine the length of the new list. Rather than using a *reduce* structured operation, this is done using a *foldl* calculation which simultaneously calculates the corresponding start index in the new list for each block

- The final phase is the redistribution of the data. This is again fully parallelised, with the computation aligned on processors with the old list data. The amount of communication this generates, and its locality, is a complex function of the distribution of items being copied from the old list. In the Cray T3D implementation, this is achieved with point-to-point writes direct into the target memory, which is quite robust on this relatively dense topology. On other architectures a more sophisticated implementation might be called for.

For the initial test of the *filter* operation, we use the following program, which generates a list of base type `int` and applies a filter function which selects two thirds of the elements in a repeated "no, yes, yes" pattern:

```
#define TRUE 1
#defin FALSE 0

int f(int list l, int index)

{
  if (l[index] % 3) {
    return TRUE;
  } else {
    return FALSE;
  };
}

int main()

{
  int i;
  int list l1,l2;

  /* length and iterations  parsed
     from command line arguments */

  newlist(l1,length);
```

```
  for (i=0; i<length; i++) l1[i]=i;

  /* start timing */

  for (i=0; i<iterations; i++) l2=filter f l1;

  /* stop timing */
}
```

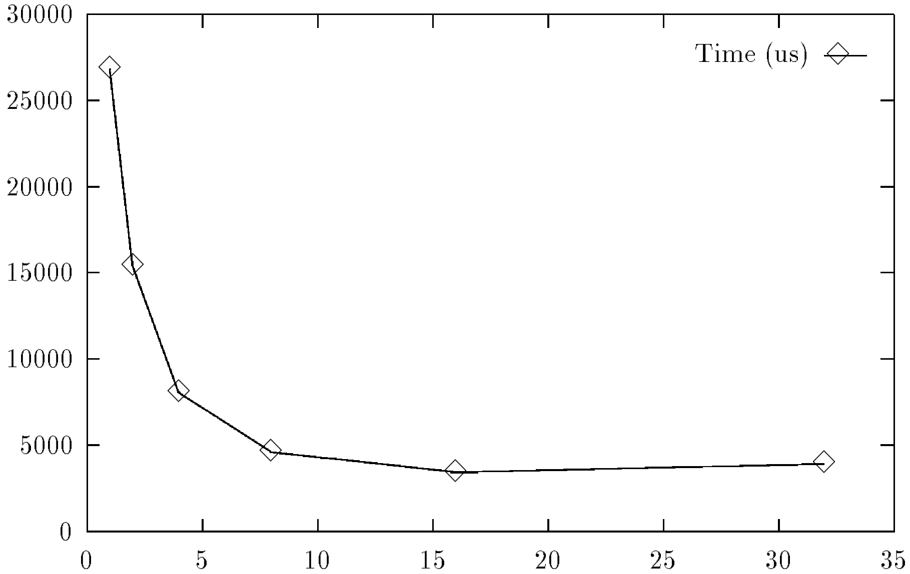

Figure 5.12: Execution time for *filter* v. n ($l = 1024$)

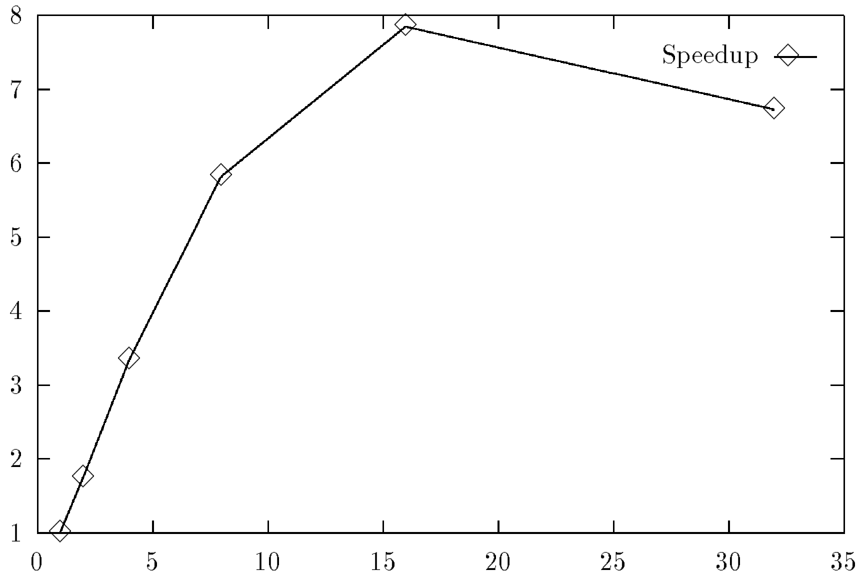

Figure 5.13: Speedup for *filter* v. n ($l = 1024$)

Graphs 5.12 and 5.13 present the performance of this *filter* operation on a 1024 element list over a range of numbers of processors.

We see a similar profile to the *reduce* operation, with a reasonable speedup which tails off as the number of processors reaches 32. This operation is composed of four stages with *map*, *reduce*, *fold* and random patterns, so this represents a reasonable average.

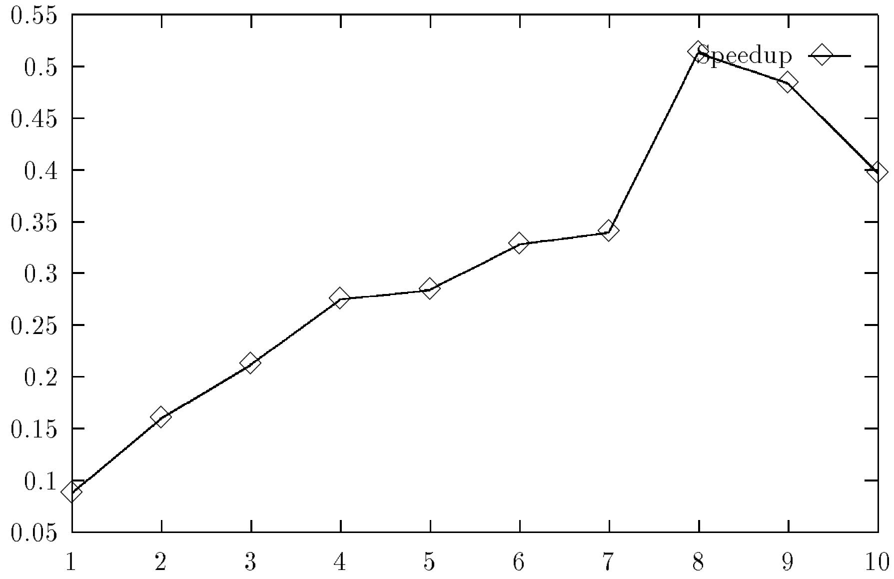

Figure 5.14: Speedup for *filter* v. blocking factor ($l = 1024$, $n = 32$)

The effect of the blocking factor on *filter* is more complex than on other operations, since it has a chaotic relationship to the data redistribution pattern at the end of the operation. Graph 5.14 shows relative speedup on 32 processors for a variety of low blocking factors, compared to the performance default blocking factor of 32. Note the relatively good performance, in context, when the blocking factor is 8, as this is well suited to the power of two topology of the T3D architecture, while the performance tails off again at 9 and 10 which are not powers of two. The fact that performance is better with a blocking factor of 9 than one of 10 can be attributed in part to the fact that the artificial data set we have used has a pattern which repeats in multiples of 3, but also depends on it being closer to 8.

In general, the relationship of performance to blocking factor for non-uniform operations such as *filter* is complex almost to the point of being chaotic; on the T3D and similar architectures, powers of two give the best and most consistent performance.

5.3 Summary

There is clear potential for very reasonable performance to be delivered by parallel programs written in this BMF style; the key factor to this is the scalability. The results for the *foldl* operation emphasise the importance of improving the algorithm at the design stage to try to eliminate these and other sequential operations which can inhibit concurrency.

What a library based implementation cannot show is the potential for fine grained concurrency; this necessitates the removal of the large number of procedure call overheads inherent in the implementation method.

Chapter 6

A Case Study

In this chapter, we will use a small but realistic "real world" algorithm to assess the usefulness and expressiveness of the language proposed in section 3.2.

6.1 Desiderata for an Example Algorithm

As discussed in the early part of this thesis, there is no universal model for parallel computation. Analogously, there can be no single parallel computing paradigm or language which can comfortably express the full range of possible parallel algorithms - even in the realm of sequential computing, where there *is* a universal model of computation, there are several paradigms (imperative, functional, declarative) and dozens of programming languages.

Clearly, it is possible to shoe-horn almost any algorithm from any programming paradigm into almost any language, but to do so is not useful for the purposes of understanding the capabilities of that language.

We can identify the following desirable characteristics which would make an algorithm amenable to implementation in this language:

- Perhaps obviously, it must have scope for concurrent execution. There is no point in trying to express parallelism only to fall afoul of Amdahl's Law [17] at the first hurdle

- Equally, there is little point in choosing a problem from the category previously referred to as *embarrassingly parallel* problems; these are problems which readily admit of a solution which has a potential concurrency of several thousand or even million fold, with no need for exchange of data between sub-problems; typical examples are computationally intensive rendering (ray-tracing) and brute force cryptanalysis. While these problems permit solutions which make excellent use of the resources offered by a parallel machine or loosely coupled network, they are not subjects of interest for studying parallelism. In this paradigm, they would largely boil down to a single *map* or *reduce* operation

- The characteristic data storage needs of the problem must fit what we have available; in this case, the BMF list data type which we have constructed is essentially a parallel equivalent of the one-dimensional array data types common in imperative programming languages, and shares most of its properties with them. An algorithm which uses one dimensional arrays, or arrays of arrays, is therefore most suitable; one which relies on true multi-dimensional properties or which uses a highly dynamic pointer based data structure would be unsuitable.

- It is also desirable, though not essential, that the algorithm solves a relevant, real-world problem, of a class of problems of sufficient complexity to justify the use of parallelism to solve them.

A class of algorithms which closely fits the above criteria is that of solutions to structural engineering and other problems by finite element analysis.

6.2 The Finite Element Method

Consider the situation where there is a piece of structure, made of metal or some other material, which is fixed in place at certain points, and subject to forces applied at others. This might be a supporting joist in a building, or a suspension component on a motor vehicle. It is necessary for the design engineer to know how reliably the structure will withstand the loads being placed upon it, in order to optimise its design for strength, weight and fabrication cost.

The suitability and durability of the structure's design can be predicted with a high degree of certainty provided that information such as the distribution of stresses within its material and amount of deformation it will undergo can be determined.

In all but the simplest of cases, it is impossible to algebraically derive an exact analytical solution which will yield this information. The way forward in this case is to use a numerical approximation - a finite element discretization.

The type of finite element solution which we will consider here is one of the simplest, but yet still widely used; a linear, elastic solution.

6.2.1 A Linear Elastic FE Solution

An approximation to the geometry of the structure is modeled using a continuum of elements of simple, polygonal or polyhedral shape and finite size; the key point is that the equations for stress or other values *can* be solved for each of these finite elements.

Each element has a certain number of node points associated with it; typically, these are located at vertices, at the midpoint of edges and occasionally in the middle of faces or solid element bodies. At each node, there are a number of degrees of freedom, depending on the element formulation.

The elements are coupled together to form the global structure at these node points, where they coincide geometrically. The entire structure is thus approximated as stiffness properties coupling a finite number of degrees of freedom.

The solution proceeds in three principal phases, often known as *data generation*, *reduction* and *retracking*.

The steps in the data generation phase are:

1. For each element in the structure, an element stiffness matrix is generated

which expresses the "coupling" it provides between the degrees of freedom at its node points.

2. The structure is assembled by combining these element matrices to form a global stiffness matrix. This large, square matrix contains one row (and one column) for each degree of freedom in the global structure.

3. The loading forces at each node point are assembled into a *load vector*

4. The fixed point constraints, conventionally known as *boundary conditions* are applied by effectively removing these degrees of freedom from the problem

We now have the stiffness equations for an approximation to the structure expressed in a matrix form as a set of simultaneous equations where each free variable corresponds to a degree of freedom in the global structure. The reduction phase is therefore nothing more than a conventional solution of this system of equations using a technique such as Gaussian elimination.

The retracking phase recovers the results by back substitution; this typically yields displacement vectors for each node (i.e. a value for each degree of freedom), reaction forces corresponding to the constraint points, and the stress field in each element.

Using a carefully discretised finite element model, with a high degree of refinement (i.e. a large number of small elements) it is possible to obtain numerically approximated results which are within a few percent of the actual values. Since in most engineering applications it is typical to leave a factor of safety (margin of error) of 100 percent or more, these results are more than sufficiently accurate for most engineering purposes.

6.2.2 Relevance

The applicability of this discretization technique is not limited to problems of structural engineering, but can be applied to almost any continuum problem, and equivalent applications exist in fields as diverse as geology and electromagnetics.

While theoretical physics and molecular biology tend to get more scientific attention, finite element solutions and other similar techniques are undoubtedly still the "bread and butter" of the supercomputer industry.

Moreover, the parallelisation of production FE solvers is far from trivial, and is not generally very successful; the current state of the art in commercial products [24] for a linear solution is to achieve a speedup factor of around 3.5 for the reduction and retrack phase of a large problem on 8 processors, on a *shared* memory system, with little or no gain when using larger numbers of processors. Parallel solutions for distributed memory are not generally attempted, nor is parallelism of the data generation phase, which can require significant computation for large problems.

While many of the reasons for the comparative lack of parallelism in commercial FE products are historical engineering rather than fundamental science, it does not negate the point that a simple way of expressing parallel algorithms could be of real benefit in this application area.

6.3 Applying BMF to an FE Solution

For the purposes of illustration, let us consider one of the simpler variants of a linear FE solution, a two dimensional membrane problem. This will be composed using a single type of triangular element, with three nodes, one at each vertex, and two degrees of freedom per node. The algorithm for this solution is taken from Chapter 2 of Cheung and Yeo [51].

The following sections present code fragments which implement the data generation and reduction phases of the algorithm; for simplicity, rather than presenting code for reading the data in from a formatted ASCII input file, as is the universal case with FE solvers, we will assume that this task has already been completed and that the data is contained in a number of BMF list structures.

6.3.1 Overall Structure of the Program

The bulk of the main program merely consists of a series of invocations of parallel BMF operations:

```
int main(int argc, char **argv)

{
  int i;

  /* Data input from ASCII file omitted for clarity; we assume
     that the lists coords, lnods, aload and nfix have been
     initialized with the correct data */

  smatrices = map cesm(coords) lnods;

  map gsmrow(lnods,smatrices) ss;

  map calcload(aload) sl;
  map boundary(ss) nfix;

  for (i=0; i<(lengthof(ss)-1); i++) map reduce_row(i) ss;

  /* Retrack would now be performed */
}
```

6.3.2 Data Declarations

The declarations for the main data structures are:

```
typedef struct {
  float x,y;
} node;

typedef struct {
  int n1,n2,n3;
} element;

typedef struct {
   float x,y;
} load;
```

```
typedef struct {
  int xfix,yfix;
} constraint;

typedef struct {
  float coeff[6][6];
} elstiff;

node list coords[MAXNODES];
element list lnods [MAXELEMS];
load list aload[MAXNODES];
constraint list nfix[MAXNODES];

elstiff list smatrices[MAXELEMS];

float list list ss[2*MAXNODES][2*MAXNODES];

float ym,pr,t;

float ec1,ec2,ec12;
```

A number of types and data structures have been defined; the BMF lists used to store the main data items have been given the same names as the equivalent arrays in the example FORTRAN code given in [51]. These types and data structures are:

- `node` - a C structure type to represent a node definition, giving its (x, y) coordinates
- `element` - a C type for an element, defining it in terms of the three corner nodes
- `load` - a type for the load vector at a single node
- `constraint` - a type for constraint flags; for each DOF, the value `0` indicates free, and the value `1` means fixed
- `elstiff` - a C struture containing a 6x6 matrix, which is represented using a conventional C two dimensional array; this is used to store an element stiffness matrix

- `coords` - a BMF list of `node` types, holding the coordinates for the model
- `lnods` - the element definition for the model
- `aload` - the load vector for the model
- `nfix` - the constraint vector for the model
- `smatrices` - a list containing the element stiffness matrices
- `ss` - the global stiffness matrix, represented by a list of lists of `float`
- `ym`, `pr`, `t` - material properties (Young's modulus, Poisson's ratio, thickness)
- `ec1`, `ec2`, `ec12` - coefficients calculated from the above

6.3.3 Element Stiffness Matrix Function

Now we define the base algebra function to compute an element stiffness matrix; this is used within a *map* operation to perform these operations in parallel. This is one slight divergence from the sequence of operations in [51] as the implementation given there creates each element stiffness matrix one at a time and assembles them to form the global stiffness matrix on the fly.

Once again, we have preserved the variable names used in the original.

```
elstiff cesm(element list e, int index, node list n)

{
   elstiff ematrix;

   int nic1,nic2,nic3;
   float x1,y1,x2,y2,x3,y3;
   float area;
   float bi,ci,bj,cj,bm,cm;
   int row,col;

   nic1=e[index].n1
   nic2=e[index].n2;
   nic3=e[index].n3;

   x1=n[nic1].x;
```

```
y1=n[nic1].y;
x2=n[nic2].x;
y2=n[nic2].y;
x3=n[nic3].x;
y3=n[nic3].y;

area=(x2*y3-x3*y2-x1*(y3-y2)+y1*(x3-x2))/2.0;

bi=y2-y1;
ci=x3-x2;
bj=y3-y1
cj=x1-x3
bm=y1-y2
cm=x2-x1

ematrix.c[0][0]=ec1*bi*bi + ec12*ci*ci
ematrix.c[1][0]=(ec1*ec2+ec12)*bi*ci
ematrix.c[1][1]=ec1*ci*ci + ec12*bi*bi
ematrix.c[2][0]=ec1*bi*bj + ec12*ci*cj
ematrix.c[2][1]=ec1*ec2*bj*ci + ec12*bi*cj
ematrix.c[2][2]=ec1*bj*bj + ec12*cj*cj
ematrix.c[3][0]=ec1*ec2*bi*cj + ec12*bj*ci
ematrix.c[3][1]=ec1*ci*cj + ec12*bi*bj
ematrix.c[3][2]=(ec1*ec2+ec12)*bj*cj
ematrix.c[3][3]=ec1*cj*cj + ec12*bj*bj
ematrix.c[4][0]=ec1*bi*bm + ec12*ci*cm
ematrix.c[4][1]=ec1*ec2*bm*ci + ec12*bi*cm
ematrix.c[4][2]=ec1*bj*bm + ec12*cj*cm
ematrix.c[4][3]=ec1*ec2*bm*cj + ec12*bj*cm
ematrix.c[4][4]=ec1*bm*bm + ec12*cm*cm
ematrix.c[5][0]=ec1*ec2*bi*cm + ec12*bm*ci
ematrix.c[5][1]=ec1*ci*cm + ec12*bi*bm
ematrix.c[5][2]=ec1*ec2*bj*cm + ec12*bm*cj
ematrix.c[5][3]=ec1*cj*cm + ec12*bj*bm
ematrix.c[5][4]=(ec1*ec2+ec12)*bm*cm
ematrix.c[5][5]=ec1*cm*cm + ec12*bm*bm

for (row=0; row<5; row++) {
  for (col=row+1; col<6; col++) {
    ematrix.c[row][col]=ematrix.c[col][row];
  };
};

for (row=0; row<5; row++) {
  for (col=row+1; col<6; col++) {
    ematrix.c[row][col]=ematrix.c[row][col] * t / (4.0 * area);
  };
```

```
    };

    return ematrix;
}
```

The detail of the body of this function is not of particular interest; the key features are that it performs a reasonable number of operations, thus making for significant granularity, and that there are no data interactions between them. It is thus highly suited to being parallelised in this way.

The expression `map cesm(coords) lnods` can now be used to build a list containing the stiffness matrices.

6.3.4 Global Stiffness Matrix Assembly

The assembly of the global stiffness matrix is little more intricate; there are data dependencies caused by the fact that more than one element contributes to each degree of freedom in the global model. There are a number of approaches to this problem, any one of which might be suitable depending on the circumstances:

- One simple way is to sequentialize the whole assembly process, mimicking the sequential implementation in [51]. While this avoids any race conditions, it is clearly not meeting the objective of producing a parallel program.
- It might be possible to use some kind of access control mechanism for the entries in the global stiffness matrix; however, we have not made this option available in the language, and so this will not be considered further here.
- The canonical way of doing this in the BMF paradigm is to let the data drive the parallelism; in this case, we structure the parallelism based on the result data, i.e. the global stiffness matrix.

There are two levels of potential parallelism in the generation of the stiffness matrix, which are to generate the rows in parallel, and to generate the columns in parallel. It is unlikely to be efficient to exploit both, as in a realistic situation there would not be a sufficient number of processors to exploit this much concurrency, and the overheads in doing so would swamp any advantage. For simplicity, we present the algorithm with only one layer of parallelism

```
void gsmrow(float list list gsm, int index, element list e,
            elstiff list es)

{
  float list myrow;
  int mynode,myfreedom,i;

  myrow = gsm[index];
  mynode = index / 2;    /* two freedoms per node */
  myfreedom = index % 2;

  for (i=0; i<lengthof(myrow); i++) myrow[i]=0.0;

  for (i=0; i<lengthof(e); i++) {
    if (e[i].n1 == mynode) {
      myrow[2*e[i].n1]   += es[i].c[myfreedom][0];
      myrow[2*e[i].n1+1] += es[i].c[myfreedom][1];
      myrow[2*e[i].n2]   += es[i].c[myfreedom][2];
      myrow[2*e[i].n2+1] += es[i].c[myfreedom][3];
      myrow[2*e[i].n3]   += es[i].c[myfreedom][4];
      myrow[2*e[i].n3+1] += es[i].c[myfreedom][5];
    }
    else {
      if (e[i].n2 == mynode) {
        myrow[2*e[i].n1]   += es[i].c[2+myfreedom][0];
        myrow[2*e[i].n1+1] += es[i].c[2+myfreedom][1];
        myrow[2*e[i].n2]   += es[i].c[2+myfreedom][2];
        myrow[2*e[i].n2+1] += es[i].c[2+myfreedom][3];
        myrow[2*e[i].n3]   += es[i].c[2+myfreedom][4];
        myrow[2*e[i].n3+1] += es[i].c[2+myfreedom][5];
      }
      else {
        if (e[i].n3 == mynode) {
          myrow[2*e[i].n1]   += es[i].c[4+myfreedom][0];
          myrow[2*e[i].n1+1] += es[i].c[4+myfreedom][1];
          myrow[2*e[i].n2]   += es[i].c[4+myfreedom][2];
          myrow[2*e[i].n2+1] += es[i].c[4+myfreedom][3];
          myrow[2*e[i].n3]   += es[i].c[4+myfreedom][4];
          myrow[2*e[i].n3+1] += es[i].c[4+myfreedom][5];
        }
      }
    }
```

```
  }
}
```

This algorithm makes a single pass down the list of elements, looking to see if any of them is connected to the degree of freedom corresponding to the row of the matrix being generated. If so, the contribution is extracted and added.

Note that this works in place on the existing matrix, so the base algebra function and the corresponding *map* have a `void` return type.

This is invoked with the statement `map gsmrow(lnods,smatrices) ss;`

6.3.5 Loads and Boundary Conditions

Application of the load and boundary conditions are both achieved by trivial *map* operations which work well in parallel, and have base algebra operations of $O(1)$ cost. We will not illustrate them here.

6.3.6 The Reduction

The Gaussian elimination algorithm also admits of two levels of concurrency; for each pivot, we can potentially parallelise down the remaining rows of the matrix which are to be operated on, and also across these rows. Here we will express both.

The inner and outer base algebra functions are, respectively:

```
void reduce_element(float list r1, float list r2, int index,
                    float factor, int pivot_index)

{
  if (index > pivot_index) r2[index] += r1[index] * factor;
}


void reduce_row(float list list ss, int index,
                int pivot_index)
```

```
{
  float factor;
  float list myrow,pivot_row;

  if (index > pivot_index) {

    myrow=ss[index];
    pivot_row=ss[pivot_index];

    factor= - myrow[pivot_index] / pivot_row[pivot_index];

    zip reduce_element(factor,pivot_index) pivot_row myrow;

    sl[myrow] += sl[pivot_row] * factor;

  };
}
```

The code fragment for the main outer loop of the elimination which is used to call this function is simply:

```
for (i=0; i<(lengthof(ss)-1); i++) map reduce_row(i) ss;
```

6.4 Ease of Expression

One of the claims which we made on behalf of the BMF paradigm was that it would make the expression of parallel algorithms easy and convenient. This is inevitably largely a subjective judgement, but there are one or two points we can note:

- The main steps of the algorithm have been picked out as clear individual or compound BMF operations
- There has been no need to introduce any explicit concurrency management or communication management
- The code is in many ways simpler than its FORTRAN equivalent, though admittedly it has taken advantage of type constructs and subroutines which

are not used in FORTRAN

One obvious negative point is the need to use subroutine notation with extra arguments to express even the simplest base algebra operations, which might be more conveniently expressed in an in-line, functional manner. This perhaps indicates a need for additional functional syntax in the language.

6.5 Performance

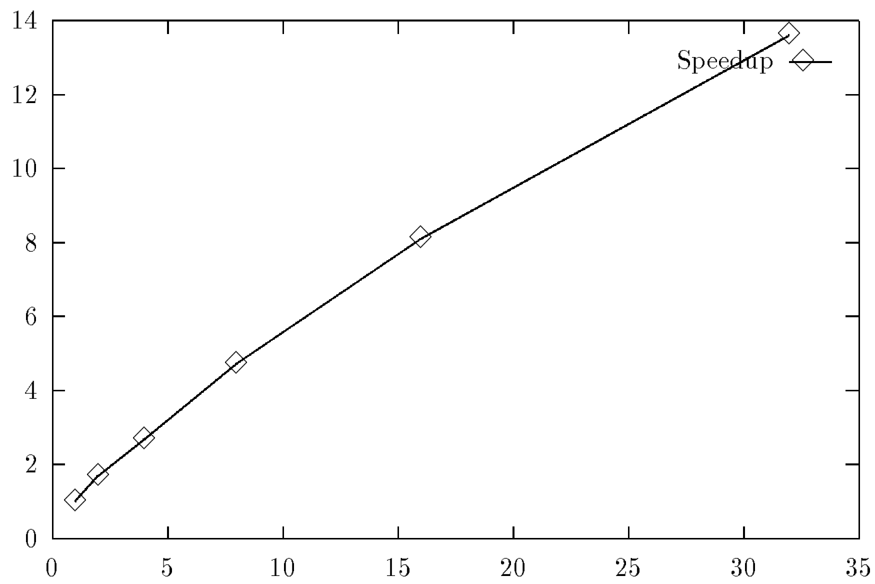

Figure 6.1: Relative Speedup Against Number of Processors for BMF FE Solver

The implementation of the algorithm as presented is in some areas a bit wasteful of resources, notably in the assembly of the stiffness matrix. This is a typical tradeoff in parallelism of increasing waste of machine resources against reducing total solution time for the problem.

It should be noted, however, that it is the norm for a program on a parallel system to be allocated a fixed partition of processing nodes for the entire duration of its execution. This being the case, a more efficient but less parallel algorithm would result in the bulk of these processing nodes lying idle during this phase of the program. In any case, there are obvious alternative implementations which can shift this equilibrium depending on the target platform.

The algorithm above was translated to run using the prototype library implementation of chapter 5. Clearly, with the relatively small granularity of the base algebra operations, a library-style implementation which does not offer the scope for inlining them is subject to a large amount of procedure call overheads and so the absolute level of performance is not particularly relevant. What is of interest, however, is the scalability, which is shown for a typical data set in figure 6.1

6.6 Comparison with Scalar Performance

The traditional acid test for any parallel program is the performance it delivers when compared to a purely sequential equivalent running on a single node of the parallel machine. The library based implementation we have used here does not make a very a good showing - the performance of a sequential C implementation of the same algorithm (i.e. with *map* and *reduce* operations replaced by loops) on the same scale as fig. 6.1 is 50.3 - i.e. the "net speedup" on 32 processors is actually a slowdown of approximately 70 percent.

While this does show that the library implementation is untenable if we wish to use BMF operations with such fine granularity, it does not contradict the "real world" implementation we have proposed which would require a compiler to in-line the remote accesses which trigger the BMF operations. A preliminary study (previously reported in [13]) indicated that, using an inlining paradigm of this type, realistic speedup factors (i.e. significantly in excess of unity) could be obtained even with this level of granularity.

6.7 Summary

In this case study, we have shown how a real world algorithm can be implemented in a BMF paradigm without radical alterations to its structure, and while realising significant potential for concurrency. For the particular application area of finite element analysis, the "top-down" approach of expressing the algorithm in the BMF style offers potential for better exploitation of its concurrency than the current industrial practice of "bottom-up" parallelisation of code in a sequential language like FORTRAN.

Chapter 7

Possible Enancements

The concepts presented in the preceding chapters are something of a beginning; there is vista of related topics which could be explored, many of which would need to be investigated before this type of structured parallelism could be considered an established paradigm for developing parallel software.

In this chapter, we will discuss a number of ideas which have not been covered within the scope of the current project, but which are a natural extension of it.

7.1 BMF and C++

In the early part of this thesis, we started by proposing a quite general paradigm for parallelism, based on using the operations of the Bird-Meertens Formalism as parallel constructs in an otherwise conventional, imperative programming language. We then went on to propose a language which is an instance of this paradigm, based on BMF lists and C.

It must be stressed that both of these choices were somewhat arbitrary, almost wholly orthogonal to the main thrust of the investigation, and made on the basis of purely pragmatic, technical considerations.

The BMF theory of lists is not only the simplest in concept, but it is by far the best understood and best developed of the BMF-style data types, and in many ways the most practical. In particular, its suitability for parallelism has long been established. It was the obvious and only choice of data structure for this project.

The C programming language is the most widely supported one, and one of the most flexible for prototype systems. It has a rich type system, allowing a variety of programming styles to be implemented easily, and it allows the programmer easy access to low level operations. It also offers unrivaled performance. Once again, it seems an obvious choice.

It is a widely held view that, if C is good, then C++ [48] is better. C++ offers all of the benefits of C, while also providing built in mechanisms to support object-oriented programming. Significantly, unlike many other object oriented languages, C++ offers these extra features with no performance penalty; the magic all happens at compile time.

Since we are proposing the structured parallelism paradigm as a method of software engineering, it would clearly be highly desirable to have object oriented programming to add to our tool box, so it would seem to be a clear cut decision to discard the C component in our prototype language in favour of C++.

The original intention in this project was in fact to use C++ as the sequential language on which to base a prototype example; the following points however mitigated against it:

- the C++ language is quite a lot larger and more complex than C, with nearly double the number of reserved words
- the overloading feature is implemented using fairly complex interactions between the compiler and linker (see section 7.1.3), which would have unnecessarily complicated the process of producing a prototype implementation
- most importantly, the use of C as a basis for the prototype language does not detract from the argument being made; indeed, the choice of base language is largely irrelevant to the concepts under investigation

We can substantiate this point of view by considering the extra features which C++ possesses over C, and how they would impact on our prototype language and its implementation.

7.1.1 Object-Orientation

Object-oriented programming is the *raison d'être* of C++, and is the source of most of its additional functionality. It should be noted in passing that is perfectly possible to write object-oriented code in "plain" C; the MIT Project Athena implementation of the X11 window system protocol is a good example of this put into practice.

Objects in C++ are implemented using an identical mechanism to "`struct`" structures in C. The invocation of a normal method is resolved at compile time to a simple function call; a virtual method, which allows different derived classes to supply different implementations, is implemented by nothing more nor less than a function pointer.

As a simple extension to the concept of being able to nest BMF data structures arbitrarily with sequential ones, it would be perfectly proper for an object to have a BMF list as a member.

Equally, one might envisage a list whose elements were objects; this does raise one minor issue, which is the automated invocation of constructor and destructor functions for the objects in the list. This can be easily and neatly dealt with by defining the constructor (destructor) for a list of objects to be an implicit BMF *map* operation of the object constructor (destructor) over the list.

One further implementation issue concerns the handling of list elements which are members of different derived classes. Clearly, these may be of differing size, while having list elements of fixed and uniform size is critical to maintaining the requisite properties of our data distribution strategy, so it would be impossible to store them within the list structure itself. The solution is elegantly simple and conventional; a list of objects would be implemented as a list of pointers to the object data structures, each of which would itself reside on the heap on the corresponding processor node. This is in fact how objects are handled in many sequential C++ implementations.

7.1.2 Implicit Arguments

A feature of C++ which in fact translates nicely into the BMF paradigm is the passing of objects as implicit arguments with method calls. Within the scope of the body of a C++ method, the reserved word `this` can be used as a pointer to the object upon which the method has been invoked, exactly as if it had been passed as an extra, hidden argument during the function call. Indeed, this is how it is often implemented by C++ compilers.

An analogous concept to the `this` argument works well for a base algebra function which is called within a BMF operation, whether the function is an object method or not. As well as using `this` to refer to the list element upon which the function is being invoked, it would be natural to have implicit variables giving access to a handle for the entire list, and giving the index of the requisite element(s) within the list, e.g. `thislist` and `thisindex` for unary operations, with perhaps `leftindex` and `rightindex` for binary ones.

7.1.3 Overloading

The C++ language allows overloading of function names, and also of certain in-line operators. The choice of which instance of a function to invoke is decided by the number and type of the arguments. The different instances of an over-loaded function are distinguished within the compiled object code by encoding the argument types within the entry point names for functions.

In the majority of cases the compiler can determine at compile time which instance of a function is needed; the exception is the case of a virtual method, which is resolved using a function pointer. Consider a BMF operation with one of these overloaded functions as the base algebra operator: if the first case applied, then the exact instance to be called would be known at compile time, and could be passed in the messages sent out by the node initiating the parallel operation; if the function involved was a virtual method, this would be resolved by examining the function pointer contained within each object on each node.

The C++ language also allows the programmer to overload unary and infix operators which most languages would regard as intrinsic, by defining functions or methods which implement them for user defined data types. This can lead to very elegant and natural syntax; the canonical example is a user defined type for complex numbers which overloads all of the arithmetic operators.

It would be perfectly natural for a programmer to do this with list data types as well; one can imagine the `+` operator overloaded as a list concatenation, etc.

None of these overloading behaviours would present any difficulty in the BMF paradigm; they could be implemented in the same way as in sequential C++ systems.

7.1.4 Call By Reference

C only offers one semantic option for parameter passing, which is call by value. Other behaviours can be mimicked by using the flexibility of the type system; in particular, call by reference is achieved by passing pointers or array handles, and is so widespread within the standard libraries that it can be considered more or less intrinsic to the language.

C++ departs from this simplicity by offering a true call by reference syntax; one again, it's all done with pointers, and this extra bit of syntax disappears in a puff of logic at compile time.

7.1.5 Summary

Under detailed scrutiny, when viewed from the perspective of object code and implementation detail, the extra features in C++ largely boil down to what might be called "syntactic sugar", were it not for the fact that it is a somewhat perjorative term. Ultimately, any programming language is just syntactic sugar obfuscating the details which exist at the binary code level, but that does not negate its importance. These features of C++ are both real and useful.

The positive aspect of them being mere syntactic sugar is that they do not have any significant impact on the semantics or implementation of the paradigm we have proposed. We can justifiably argue that the option of seizing C++ and plugging it into the existing framework is tenable.

7.2 Alternative BMF Data Types

The scope of the investigation presented thus far has been restricted to the BMF theory of lists, for valid reasons discussed in section 7.1 above. The BMF concept lends itself readily to other data structures; an analogous theory of trees [18] is fairly well defined, while more exotic types such as one to define chemical molecules [43] have been proposed.

It would be perfectly reasonable to include these data types and their operators in a structured parallelism paradigm. One note of caution; these other data types do not necessarily possess the elegant suitability for parallelism offered by the BMF theory of lists.

7.2.1 The Theory of Arrays

One type in particular worthy of mention in this context is the theory of arrays [32].

It is worth noting again at this point that names of these data types are somewhat misleading when it comes to considering their properties in a parallel implementation; in most sequential programming languages, the nearest equivalent to BMF lists as used in this project is called an "array", and the syntax offered for so-called multi-dimensional arrays is mere syntactic sugar for a single dimensional one or an array of arrays.

In contrast, the theory of arrays presented by Miller represents a true multi-dimensional data type; each dimension of an array has equal status. In the theoretical construction, an array is formed by composing two smaller arrays together along a single dimension, using a constructor; there is a variant of the constructor for each dimension. It is a prerequisite that the two sub-arrays being composed have identical shape, i.e. identical size in every dimension, apart from the one along which they are being composed.

In the common case of two dimensions, these compositions are often informally known as "above" and "beside". This leads to a multi-dimensional analogue of commutativity and distributativity known as "abides with".

If we consider an array of a lower number of dimensions to be a degenerate case of one of a higher number of dimensions, it leads to a fairly natural construction, with the base case of a single element being considered as an array of arbitrary dimensionality, and of size 1 in each dimension.

7.2.2 A Tree-Like Implementation

The difficulties arise when we try to translate this theory of arrays into a concrete data type and implement it. The obvious implementation results in a data structure which would more usually be called a binary tree. There are established methods for distribution of a binary tree across a parallel computer, but the problems arise from the fact that an array which we might like to treat as a seamless block is in fact built from sub-components in an arbitrary fashion. While the implementation is busy papering over the joins, any chance of performance prediction slips through our fingers.

Consider the simple example of a two dimensional array A which might represent a matrix, composed of four sub-arrays tl, tr, bl, and br. We can represent A in two ways:

$A = (tl \text{ beside } tr) \text{ above } (bl \text{ beside } br)$

and

$A = (tl \text{ above } bl) \text{ beside } (tr \text{ above } br)$

provided the constraints on the shapes of the sub-arrays required to make the construction “legal” apply; the simple case where it is possible is where all four have the same shape.

In the tree-like implementation of this data type, it is clearly far from trivial to define and implement something as simple as a natural test for equality, and impossible to make it efficient. Array slicing and striding are equally complicated, and even the simple task of locating a single element spans multiple processors and has a potential $O(n)$ cost if the tree is not balanced.

7.2.3 An Alternative Implementation

An alternative way of implementing true multi-dimensional arrays would be to use a multi-dimensional analogue of the data distribution technique we have used for BMF lists as detailed in section 3.3. The storage for an array would be defined by its starting point, and a number of blocking factors, one for each dimension. Mapping of the dimensions onto the interconnect topology of a parallel machine would be hard-coded in the implementation of the language.

This implementation gains most of the desirable properties of the one for lists; in particular, we can slice and stride arrays at a sensible cost, and locate elements using information directly available at any single node.

The implementation effectively discards the theoretical constructors (above, beside, etc.) which are used to reason theoretically about the data structure, in return for performance.

One minor aesthetic disadvantage is that the process of mapping the array onto the machine topology introduces a performance asymmetry between the various dimensions. In essence though, this is analogous to the performance artifacts caused by cached and interleaved memory architectures on modern sequential computers when using conventional programming languages like C or FORTRAN. Additionally, this problem is more or less negated in many common, practical cases, where the array would typically be two- or three-dimensional, and the machine topology would be a three-dimensional torus.

Implementations similar to this are not uncommon in other parallel paradigms, such as data parallel variants of FORTRAN.

7.3 Support for Concurrent Data Access

Closely associated with the idea of threads of a BMF operation communicating is the idea of them accessing shared data. Once again, this goes against the "natural" style of BMF programming as we have defined it.

To support shared data access in a rigorous fashion, it would be necessary to have some form of atomic operation which could be used to represent acquiring a lock over a data object; the simplest form of this is an atomic spin-lock or semaphore, which could quite easily be added to the implementation. More complex access control algorithms such as monitors and range locks could then be implemented at the programmer's level, e.g. in the form of libraries.

This would have a similar impact on the semantics as explicit communication, for the same reasons.

7.4 Performance Enhancements at the Language Level

There are one or two changes that could be made to the syntax of the language to allow opportunity for modest performance enhancements under the programmer's control.

7.4.1 Combining BMF Operations

If a program is written in a very natural BMF style, there are many circumstances where the list which is the output from one BMF operation is fed straight into the next; in the functional domain, the operations are considered to be joined by function composition.

In some cases, a small performance benefit might be realised by merging the two operations into one, in the sense that parts of the second operation would be triggered to start as soon as the data for them became available, while parts of the first operation are still ongoing. The advantage would have to be weighed against the cost of implementing the asynchronous scheduling involved.

In general, it is unlikely that it would be possible to develop a compiler algorithm sophisticated enough to decide when it would be safe to merge two operations in this fashion, far less one which could determine when it would be of benefit to do so. Hence it would be necessary to make the choice available to the programmer, either as a compiler directive (of the kind associated with the `pragma` directive in C) or as an additional BMF operation in the language.

The composition of operations which is the obvious candidate for this treatment, given that it occurs naturally in many algorithms, is a *map* followed by a *reduce*.

7.4.2 Early Returns

In certain applications, the result of a base algebra function may become available part way through its execution, with the remaining operations constituting side effects. It would be desirable in these circumstances to offer the programmer the option to post the return value as early as possible, while execution of the function continued. The benefit of this would be greatest in operations such as *reduce* or *fold* where the time savings would be cumulative.

An implementation which did not support sufficient concurrency to benefit from the programmer's use of this statement would be free to simply store the return value generated by the expression in the `earlyreturn` statement and use it once the end of the function execution was reached.

In a language based on C this could conveniently be done by adding the statement type `earlyreturn` *value*; - an example of the benefits of this can be seen in [26].

7.4.3 Tuning the List Shape

The data distribution proposed in section 3.3 is parameterised by two significant items which affect performance but do not have any effect on the semantics of the operations being implemented; these are the node on which a list starts and the blocking factor, i.e. the number of consecutive elements of the list stored on each consecutive node.

The prototype implementation in chapter 5 includes a library call for allocating new list storage (analogous to the C `malloc` call) and this allows the calling routine to influence (and indeed choose) these two parameters, while offering sensible defaults; in the default case, the list starts at node 0, and the blocking factor is chosen so that it spreads the list across all the nodes without wrapping around to have multiple blocks per node. This also happens to be the largest block size consistent with minimal memory wastage.

In the more natural syntax of the C based prototype language, where lists would be created and destroyed as automatic variables, it would not be desirable to embed these parameters in the middle of a variable or type declaration. Once again, we would probably resort to `pragma` directives.

7.5 Implementation Performance Tuning

While the prototype implementation used was suitable for comparative performance measurements, there is a clear gap between the absolute performance achieved and what could be achieved in a production implementation with the proper optimisations.

7.5.1 Direct Compilation

The first and most obvious cost in the prototype which could be done away with is the large number of function calls resulting from the implementation in library form. This could be mitigated significantly by inlining the code used to implement the BMF operations; while a production implementation would boast a native compiler for the language, a simple but unaesthetic solution would be to convert functions into pre-processor macros.

7.5.2 Aggregation of Small Operations

If an algorithm is coded in a very pure BMF style, with base algebra functions which are as simple as a single arithmetic operation, then in most practical circumstances implementing these operations in parallel would actually be slower than doing them sequentially on a single processor.

An obvious optimisation in this circumstance is to degenerate the list involved into a simple array stored on a single processor, i.e. a list whose blocking factor is the same as its length. Since the storage would be both local and contiguous, elements of this list could then be accessed using the simpler conventions of address arithmetic used to implement arrays in sequential C. BMF operations over the list could be inlined as simple loops acting on the array, and then subject to conventional sequential loop optimisations such as unrolling and vectorisation.

7.5.3 Distributed Memory Allocation

In the prototype implementation, all list storage is allocated from a single heap space, controlled by a single spin-lock. This constitutes a potential lack of scalability. On a closely coupled parallel system like a Cray T3x this is unlikely to present a problem with any reasonable number of processors, but on a network of workstations with network latency communication involved it could present a bottleneck. There are a number of approaches to tackling this problem:

The simplest, but potentially the most wasteful of resources, would be to have one or more processing nodes dedicated to the task of memory management. In a network environment this might be quite reasonable, as an extra node could be introduced to handle this task, and it could be a system of the same power or capacity as the ones actually running the parallel program. On a parallel machine with a highly symmetric interconnect topology, it would be undesirable as it would not only waste a node but also destroy the symmetry of the mapping of data onto the machine topology.

Another alternative is to allow several, or even all nodes to act as memory managers of distinct heaps. Although potentially wasteful of memory storage, this approach at least has the advantage that it scales to large numbers of nodes.

A sophisticated enhancement would be to allow two-dimensional management of the memory space, whereby two or more lists could share the same address range provided that they only spanned mutually exclusive subranges of nodes. This would be convenient in allowing very large blocking factors to be used (for purposes such as the aggregation described in section 7.5.2) without wasting too much memory.

7.5.4 Broadcast Communication

Most of the routines for BMF operations in the prototype implementation perform what is effectively a broadcast to initiate one or more phases of computation on remote nodes. On the Cray T3D there is no faster way of doing this than by using the shared memory writes which have been implemented, but on loosely coupled systems where the interconnect has a high latency or low bandwidth, it would be worth implementing a properly structured broadcast communication.

7.6 Parallel I/O

A significant bottleneck which prevents scaling to large numbers of nodes in many parallel systems is the process of input and output to and from entities outside the parallel system (mass storage, networks, etc.) as it is almost invariably done sequentially. It is not uncommon for all I/O to go through a single node. True parallel I/O is still something of an arcane art form, for which it is difficult to lay down generic standards.

One simple form of parallel I/O which would be easy to introduce into this paradigm would be to write a list data structure out to a disk array or other form of parallel storage. For maximum efficiency, this would depend upon having a striped file system which would allow variable stripe sizes, thus mimicking the way lists are stored. Even with something less than this ideal, the typical differential between interconnect and disk performance would give sufficient scope for significant performance gains.

7.7 Implementation on a Shared Memory System

Shared memory, symmetric multi-processor computers are taking an increasing share of the market for high-end server applications, and are now boasting enough processors to be an important consideration to the parallelism community. Systems with 32 or 64 processors sharing a single flat memory space are now common, with 256 processor systems on the near horizon.

The main implication for implementing a BMF language on these systems is that it is no longer necessary to use the complex scheme for addressing lists that we have employed; normal indexed array addressing will be sufficient. The problem of communication reduces to one of cache coherency; with well written programs as per 4.9 it would only be only necessary to perform cache synchronisations at the end of BMF operations, mitigating the inherent potential for memory conflict in these systems.

There are a number of architectures of parallel computers which simulate the availability of shared memory in hardware; a common technique is to have a global virtual memory system, where a given page can be either on disk or in the memory physically attached to one processor. A typical example is the Silicon Graphics Origin2000 [40].

While these systems offer the simplicity of shared memory to the programmer at a lower cost than true hardware shared memory implementations, they come with a caveat; as with virtual memory, naïve use of this shared memory abstraction leads to poor performance. In the case of a BMF implementation a lot of this would be within the control of the language run time system which could take account of the peculiarities of this memory architecture.

7.8 Summary

This chapter has raised a number of issues, for any of which a full investigation would be a project in itself. We can however draw some immediate observations about them:

- The use of C++ as a vehicle rather than C would enrich the scope of the example language without imposing any performance penalty; there are a few minor semantic issues but none seem insurmountable.
- It seems that the second suggested implementation of BMF-style arrays, using the same techniques as we have applied to lists, would produce a promising data structure suitable for a range of real-world whose structure it mirrors; Computational Fluid Dynamics is an obvious target field.
- Adding explicit communication and support for data sharing would break down the Composition Rule given in section 4.9, and would not be necessary to express at least a useful subset of algorithms. It would probably detract more from the paradigm than it could add. Parallel software more suited to a message passing paradigm is best developed using one.
- The main weakness of the example language and the existing prototype implementation is performance; clearly, the scope is there, and any performance enhancements possible would be welcomed with open arms.

Chapter 8

Conclusions

It was already well established that the Bird-Meertens Formalism has significant potential for concurrency, and is a suitable paradigm for refining and expressing parallel algorithms in the functional domain. The properties of the Formalism offer a sound theoretical basis on which to build the structure of an algorithm.

The contribution of this thesis is to show that it is possible to use BMF to construct a practical paradigm for expressing algorithms in an imperative domain, while retaining access to most of the benefits of the Formalism.

A programming language which exemplifies this paradigm has been outlined, based on the very practical and robust sequential language C. We have shown how to construct an operational semantics for this type of parallel language, and moreover, how to apply simple syntactic constraints which will make a program deterministic and the semantics tractable without sacrificing the scope for parallelism. The level of tractability confidence offered by these simple semantics is an option which is simply not available to a programmer using a message passing paradigm.

A prototype implementation for the example programming language has been implemented, which though not exploiting the full potential for performance, allowed a clear demonstration of the scalability and other potential in the paradigm.

The implementation of a simple but non-trivial algorithm has been demonstrated, which has shown that it is possible to capture much of the potential concurrency inherent in a problem without sacrificing simplicity.

Finally, it has been pointed out that the present work is only the beginning of this concept, and that there is a great deal of potential yet to be explored.

Developing parallel software is an extremely complex endeavour, and compared to the hardware engineers, computer scientists perhaps have a bit of catching up to do. For parallel computing to succeed as a mainstream technology, a wide diversity of techniques for implementing parallel algorithms will be needed - structured parallelism clearly merits a place among them.

Bibliography

[1] Gul A. Agha. "Concurrent Object-Oriented Programming". *Communications of the ACM*, 33(9):125–141, Sep 1990.

[2] B. W. Kernighan and D. M. Ritchie. *"The C Programming Language"*. Prentice Hall, 1984.

[3] Beowulf Project. "Beowulf Project at CESDIS". Technical report, Beowulf Website, `http://www.beowulf.org/`, June 1998.

[4] R. S. Bird. "Algebraic Identities for Program Calculation". *The Computer Journal*, 32(2):122–126, 1989.

[5] G. E. Blelloch. "Scans as Primitive Parallel Operations". *Proceedings of the International Conference on Parallel Processing*, pages 355–362, Aug 1987.

[6] Guy E. Blelloch and Gary W. Sabot. "Compiling Collection-Oriented Languages onto Massively Parallel Computers". *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[7] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby Yang. "Distributed pC++: Basic Ideas for an Object Parallel Language". Obtained electronically from Indiana University Department of Computer Science, 1992.

[8] Wentong Cai and David B. Skillicorn. "Calculating Recurrences Using the Bird-Meertens Formalism". Obtained electronically from the Department of Computing and Information Science, Queen's University at Kingston, Mar 1992.

[9] Rosemary Candlin, Joe Philips, Peter Fisk, and Neil Skilling. "Studying the Performance Properties of Concurrent Programs by Simulation Experiments on Synthetic Programs". Available from the Department of Computer Science, University of Edinburgh, May 1992.

[10] M. Cole. "Parallel Programming, List Homomorphisms and the Maximum Sum Problem". Technical Report CSR-25-93, Dept. of Computer Science, University of Edinburgh, May 1993.

[11] Murray Cole. "Algorithmic Skeletons: Structures Management of Parallel Computation". MIT Research Monographs in Parallel and Distributed Computing (Pitman), 1989.

[12] Cray Research Inc. *"Cray MPP Application Programmer's Library Reference Manual"*, volume SR-2165. CRI, Eagan MN, USA., 1994.

[13] D. Crooke. *"Robustness and Performance in Structured Parallelism"*, volume Abstract Machine Models for Parallel and Dsitributed Computing. IOS Press, M. Kara, J.R. Davy, D.Goodeve and J.Nash (eds.), 1996.

[14] Marco Danuletto, Roberto DiMeglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. "A Methodology for the Development and the Support of Massively Parallel Programs". *Future Generation Computer Systems*, 8:205–220, 1992.

[15] 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". PARLE93, 1993.

[16] S. Fortune and J. Wyllie. "Parallelism in Random Access Machines". *Proceedings of 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[17] G. M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.

[18] J. Gibbons. *"Algebras for Tree Algorithms"*. PhD thesis, Programming Research Group, University of Oxford, 1991. D.Phil Thesis.

[19] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.

[20] Soonhoi Ha and Edward Lee. "Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iteration". *IEEE Transactions on Computers*, 40(11):1225–1238, Nov 1991.

[21] R. Harper, D. MacQueen, and R. Milner. "Standard ML". Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, Mar 1986.

[22] Todd H. Heywood. *"A Practical Heirarchical Model of Parallel Computation"*. PhD thesis, Syracuse University School of Computer and Information Science, Suite 4-116, Center for Science and Technology, Syracuse NY, USA 13244-4100, Nov 1991.

[23] Todd H. Heywood and Sanjay Ranka. "A Practical Heirarchical Model of Parallel Computation". *Journal of Parallel and Distributed Computing*, 16:212–232, 233–249, 1992.

[24] Hibbett, Karlsson and Sorsensen Inc. *"ABAQUS/Standard Reference Manual - Version 5.7"*. , 1998.

[25] John L. Gustafson. "Re-evaluating Amdahl's Law". Technical report, NASA Ames Laboratory, `http://www.scl.ameslab.gov/Publications/AmdahlsLaw/Amdahls.html`, June 1996.

[26] C. B. Jones. "Process-Algebraic Foundations for an Object-Based Design Notation". Technical Report UMCS-93-10-1, Universtiy of Manchester Dept. of Comp. Sci., 1993.

[27] Laxmikant V. Kale and Sanjeev Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based On C++". Obtained electronically

from the Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.

[28] K. G. Kumar and D. B. Skillicorn. "Data Parallel Geometric Operations on Lists". Obtained electronically from the Department of Computing and Information Science, Queen's University at Kingston, 1993.

[29] Richard E. Ladner and Michael J. Fischer. "Parallel Prefix Computation". *Journal of the ACM*, 27(4):831–838, Oct 1980.

[30] James R. Larus, Brad Richards, and Guhan Viswanathan. "C** : A Large-Grain, Object-Oriented, Data-Parallel Programming Language". Technical Report 1126, University of Wisconsin-Madison, 1210 West Dayton St., Madison WI, USA 53706, Nov 1992.

[31] William F. McColl and Simon J. Knee. "The BSP Model - A Foundation for Architecture Independent Parallel Computation". Obtained electronically from the authors at Oxford University, Oct 1992.

[32] R. Miller. *"Two Approaches to Architecture Independent Computation"*. PhD thesis, Programming Research Group, Oxford University, 1994.

[33] R. Milner. *"A Calculus of Communicating Systems"*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[34] R. Milner. *"The Polyadic π-calculus: A Tutorial"*, volume Logic and Algebra of Specification. Springer-Verlag, 1992.

[35] MPI Consortium. "The Message Passing Interface Standard". Technical report, MPI Consosrtium Web Pages, `http://www.mcs.anl.gov/mpi/`, June 1998.

[36] J. M. Nash, P. M. Dew, M. E. Dyer, and J. R. Davy. Parallel Algorithm Design on the WPRAM Model. In J. R. Davy and P. M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 83–102. Oxford University Press, 1995.

[37] A. G. Ranade. *"Fluent Parallel Computation"*. PhD thesis, Yale University, 1989.

[38] Margaret Reid-Miller. List ranking and list scan on the Cray C-90. *Journal of Computer and System Sciences*, 53(3), December 1996.

[39] Marina Roesler and Walter Burkhard. "Resolution of Deadlocks in Object-Oriented Distributed Systems". *IEEE Transactions on Computers*, 38(8):1212–1224, Aug 1989.

[40] Silicon Graphics Inc. "Origin 2000 systems". Technical report, SGI Web Site, `http://www.sgi.com/origin/2000/`, 1998.

[41] D. B. Skillicorn. "Categorical Data Types". Obtained electronically from the Department of Computing and Information Science, Queen's University at Kingston, Dec 1992.

[42] D. B. Skillicorn and W. Cai. "A Cost Calculus for Parallel Functional Programming". *Journal of Parallel and Distributed Computing*, 28:65–83, July 1995.

[43] David B. Skillicorn. "Architecture-Independent Parallel Computation". *Computer*, pages 38–50, Dec 1990.

[44] David B. Skillicorn. "Models for Practical Parallel Computation". *International Journal of Parallel Programming*, 20(2):133–158, Apr 1991.

[45] David B. Skillicorn. "Parallelism and the Bird-Meertens Formalism". Obtained electronically from the Department of Computing and Information Science, Queen's University at Kingston, Apr 1992.

[46] J. M. Spivey. "A Categorical Approach to the Theory of Lists". *Springer-Verlag LNCS 375 : Mathematical Program Construction*, pages 399–408, Jun 1989.

[47] C. P. Stirling. "A Generalization of Owicki-Gries' Hoare Logic for a Concurrent While Language". *Theoretical Computer Science*, 58:347–359, 1988.

[48] Bjarne Stroustrup. *"The C++ Programming Language"*. McGraw Hill, 1990.

[49] Tera Computer Company. "Tera Multi-Threaded Architecture (MTA)". Technical report, Tera Web Site, `http://www.tera.com/`, 1998.

[50] Leslie G. Valiant. "A Bridging Model for Parallel Computation". *Communications of the ACM*, 33(8):103–111, Aug 1990.

[51] Y. K. Cheung and M. F. Yeo. *"A Practical Introduction to Finite Element Analysis"*. Pitman, 1979.