Type Oriented Parallel Programming

Nicholas Brown

A Thesis presented for the degree of
Doctor of Philosophy

Software Engineering Group
School of Engineering and Computing Sciences
University of Durham
England

January 2010

Dedicated to

My Parents

Type Oriented Parallel Programming

Nick Brown

Submitted for the degree of Doctor of Philosophy
January 2010

Abstract

Context Parallel computing is an important field within the sciences. With the emergence of multi, and soon many, core CPUs this is moving more and more into the domain of general computing. HPC programmers want performance, but at the moment this comes at a cost; parallel languages are either efficient or conceptually simple, but not both.

Aim To develop and evaluate a novel programming paradigm which will address the problem of parallel programming and allow for languages which are both conceptually simple and efficient.

Method A type-based approach, which allows the programmer to control all aspects of parallelism by the use and combination of types has been developed. As a vehicle to present and analyze this new paradigm a parallel language, Mesham, and associated compilation tools have also been created. By using types to express parallelism the programmer can exercise efficient, flexible control in a high level abstract model yet with a sufficiently rich amount of information in the source code upon which the compiler can perform static analysis and optimization.

Results A number of case studies have been implemented in Mesham. Official benchmarks have been performed which demonstrate the paradigm allows one to write code which is comparable, in terms of performance, with existing high performance solutions. Sections of the parallel simulation package, Gadget-2, have been ported into Mesham, where substantial code simplifications have been made.

Conclusions The results obtained indicate that the type-based approach does satisfy the aim of the research described in this thesis. By using this new paradigm the programmer has been able to write parallel code which is both simple and efficient.

Declaration

The work in this thesis is based on research carried out at the Department of Computer Science, the University of Durham, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

I would like to express my thanks to both Dr Yifeng Chen and Professor Malcolm Munro for supervising me during my PhD. Their advice, guidance and encouragement over the past few years has been invaluable and without their help this thesis would simply not exist.

I am also grateful to all the members of the Computer Science department who have made me feel at home during my studies.

I would like to thank my parents for their support and patience over the past few years as well as my girlfriend, Hayley, for all she has done for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Topic Overview

The concept of splitting a problem up, solving these parts in parallel and then combining the individual answers to form a solution has been a popular one for many years. Parallel computing has traditionally been in the domain of the few experts, who have the knowledge and experience to write these highly complex parallel applications. There has been much research directed at this field with most of the attention being aimed at improving the tools and support for parallelism. One area which has not seen a great deal of improvement, although considerable research has been done, is in the actual languages used to create these parallel codes. The difficulty of programming has been the main challenge to parallel computing over the past several decades.

With the emergence of multi, and soon many, core CPUs the field of parallel computing is moving towards the more general, non-expert, programmer. If parallel computing is to continue taking advantage of these new desktop technologies then parallel programming must become more accessible to the non-expert parallel programmer. Existing parallel languages tend to support either simplicity or efficiency (performance), but not both. These objectives, which have been in conflict to this point, must become complementary within a parallel language if the field is to continue to grow and succeed in the wider context.

Those existing parallel languages built for simplicity often rely heavily on implicit parallelism. The programmer may have some control over the parallel aspects of their code, but much of the complexity is taken away, and the compiler will make key decisions regarding parallel issues. This very high level approach, and lack of direction in the source code, makes it difficult for the compiler to optimise and results in inefficient parallelism. Not only this, the programmer is often in a much better position to make certain decisions, but the loss in expressiveness means that this is not possible. A prime example of this is in some implicit parallel languages where variables can either be allocated to a single processor or all processors but nothing in between.

Within the High Performance Computing (HPC) field the pursuit for performance is one of the main objectives and to achieve such currently parallel programmers are commonly writing highly complex codes. For this reason, regardless of newer, simpler languages one of the most popular choices is to use a low-level sequential language combined with a library of parallel functions where all parallelism is explicit. The result is that these programs are difficult to develop, test, debug and modify even to the few experts. Taking this, low level, approach it is often difficult to get the "big picture" of the system as a whole and the programmer can get stuck with specific decisions they made at the start of development which in hindsight may not be the most effective.

To address the issues of simplicity and efficiency this thesis proposes a tradeoff between explicit and implicit parallelism. Type-based parallelization addresses the issue by providing the option to the end programmers to choose between explicit and implicit parallelism. The approach is to design new types governing parallelization. A programmer may choose to use these types, which imposes additional information that can guide the compiler to generate the required parallelization code, or may choose not to use them in which case some default choices will be made. In short these types for parallelization are issued by the programmer to instruct the compiler to perform the expected actions in static analysis and code generation.

Type-based parallelization is different from more traditional procedural calls to parallel libraries or keyword based languages. Programmer-imposed information about parallelization only appears in types at variable declaration and type coercions in expressions and assignments. For example, the assignment between variables may yield local assignment, communications or a combination of them, all depending on the declared or temporarily coerced types in expressions. The compiler helps the programmer to generate those non-interesting but tricky parts of the parallel code.

1.2 Criteria for Success

The 5 criteria for evaluating the success of the research described in this thesis are detailed in this section.

1. **Support code which is simple yet expressive**

   This criterion specifies that parallel code should be conceptually simple to write yet still allow for advanced programmers to enjoy a high degree of control over parallel decisions.

2. **Provide for flexible parallel programming**

   Parallel programmers often wish to get their code working and then fine tune for performance. With many existing languages changing parallel details later down the line can be very time consuming and as such programmers can be stuck with initial, ill informed, decisions.

3. **Be general and none application specific**

   There are a wide variety of parallel applications currently being used. As such it is important to develop an approach which is general and can be applied to not only existing problems but future ones too.

4. **Exhibit a high degree of performance**

   Performance is one of the primary concerns within parallel computing. Any proposed approach must be, at least, as efficient as existing high performance language solutions to stand a chance of adoption.

5. **Must be implementable**

   Arguably there is little point of a paradigm or language if it can not be implemented on a computer. From the specification it must be possible to produce translation tools which work in a timely fashion.

These criteria will be revisited and discussed in the final chapter of this thesis.

1.3 Thesis overview

The remainder of this thesis is as follows.

Chapter 2 surveys the current situation and literature relevant to this project. An overview of the parallel communication and computation models is considered along with an evaluation of existing parallel language solutions. A number of existing parallel codes are surveyed to ascertain how these applications are currently developed and benchmarks used to evaluate the performance of parallel tools. Finally the theories of programming languages are considered because in order to develop a paradigm and associated language it is important to have a background in this field.

Chapter 3 provides an informal definition of the type-based approach and programming language, Mesham, as a whole. Mesham is developed to act as a vehicle, presenting and evaluating the new type paradigm. The purpose of this definition is to give the reader a flavour of the approach itself and how it is used within the domain of parallel programming.

Chapter 4 is concerned with the implementation of the type-based approach and Mesham. In this chapter an overview of the compilation process is provided to give the reader an insight into how efficient target code is generated from the programmer's source code. A number of potential difficulties, and their solutions, found when implementing the compiler are discussed before finally a concrete example of how simple Mesham source code is translated into efficient target code is given.

Chapter 5 overviews a number of case studies implemented in the language. This chapter is specifically aimed towards performance, although some programmability aspects of the code are also considered. A number of experiments performed on Durham University's Hamilton Cluster are presented to assess how the approach performs in relation to existing high performance language solutions.

Chapter 6 presents the port of aspects of the parallel cosmological simulation package, Gadget-2, into Mesham. Extensions to the type library, used to support this work, are specified. In this chapter the issue of programmability is looked at in detail with consideration made towards whether or not the type-based approach has simplified the whole process of parallel programming.

Chapter 7 evaluates the case studies and experiments of chapters five and six. The question of whether the type based approach allows for conceptually simple and high performance parallel programming is considered. Mesham is then evaluated with respect to a number of parallel language criteria developed in Chapter 2. Finally, in order to ascertain where Mesham lies in relation the other parallel languages, a comparison with those languages detailed in Chapter 2 is performed.

Chapter 8 contains the conclusions of this thesis and summaries the research carried out. Further work growing out of this research is considered, which could develop the state of the art.

Chapter 2

Literature Review

2.1 Introduction

In researching for this project there are four broad, distinct, categories which must be considered. These are **Parallel Computation and Models**, **Parallel Paradigms and Languages**, **Theories of Programming Languages** and **Common Parallel Codes**. These are reviewed, in detail, in this chapter to identify the foundations required for the project and the strong and weak points of these fields.

2.2 Parallel Computation and Models

2.2.1 Communication

Key to parallel computing is the idea of communication. There are two general communication models, shared memory and message passing. It is important to consider both these models because of the different advantages and disadvantages which each exhibits.

Shared Memory

In the shared memory model, each process shares the same memory and therefore the same data. In this model communication is implicit. When programming using this model care must be taken to avoid memory conflicts. There are a number of different sub models, such as Parallel Random Access Machine (PRAM) [Fortune1978] whose simplicity to understand has lead to its popularity. Figure 2.1 illustrates how a PRAM would look, with each processor sharing the same memory and by extension the program to execute. However, a pure PRAM machine is impossible to create in reality with a large number of processors due to hardware constraints, so variations to this model are required in practice.

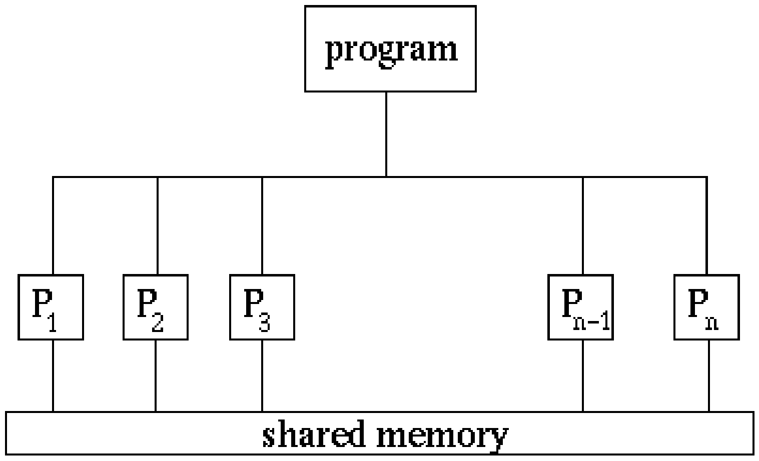

Figure 2.1: Parallel Random Access Machine

"Bulk Synchronous Parallelism (BSP) is a parallel programming model that abstracts from low-level program structures in favour of supersteps. A superstep consists of a set of independent local computations, followed by a global communication phase and a barrier synchronisation." [Skillicorn1999] One of the major advantages to BSP is the fact that the runtime can easily be deduced. The cost of a superstep is the sum of the cost of the longest running local computation, the cost of global communication between the processors and the cost of the barrier synchronisation at the end of the superstep. It is considered that this model is a very convenient view of synchronisation. However, barrier synchronisation does have an associated cost due to the global synchronisation, "the performance of barriers on distributed-memory machines is predictable, although not good." [Hill1999] On the other hand, as [Skillicorn1999] notes, this performance hit might be the case, however with BSP there is no worry of deadlock or livelock and therefore no need for detection tools and their additional associated cost. The benefit of BSP is that it imposes a clearly structured communication model upon the programmer, however extra work is required to perform the more complex operations, such as scattering of data.

Another model following the shared memory model is Logic of Global Synchrony (LOGS) [Chen2004]. LOGS consists of a number of behaviours - an initial state, a final state and a sequence of intermediate states. The intermediate global states are made explicit, although the mechanics of communication and synchronisation are abstracted away. The paper [Chen2004] is an interesting description of this model, although it is quite theoretical. A number of different properties of the model are proven in the paper, such as healthiness, soundness and completeness. In the form described by [Chen2004] it is accessible to specify example parallel problems and prove facets about them.

The study of the shared memory model is relevant due to the convenience and simplicity aspects. However, the major disadvantage of this communication model is that, due to each process sharing the same memory, performance quickly drops as the number of processors is increased. This model is therefore not scalable in terms of performance.

Message Passing

"Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory." [MPI1995] In this model, processors are very distinct from each other, with the only connection being that messages can be passed between them. Unlike the shared memory model, in message passing communication is explicit. Figure 2.2 illustrates a typical message passing parallel system setup, with each processor equipped with its own services such as memory and IO. Additionally, each processor has a separate copy of the program to execute, which has the advantage of being able to tailor it to specific processors for efficiency reasons. A major benefit of this model is that processors can be added or removed on the fly, which is especially important in large, complex parallel systems.

There are two major advantages to message passing, these are efficiency and

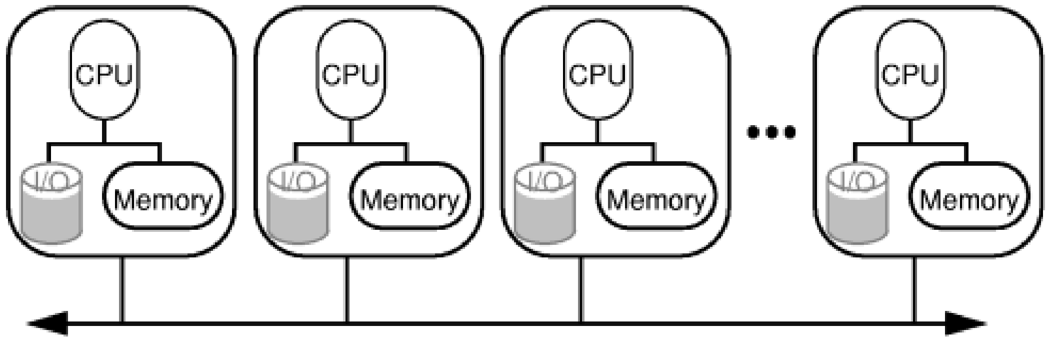

Figure 2.2: Message Passing Communication Model

scalability however it is difficult to write non-elementary message passing programs, especially when these need to be programmed using add on language libraries.

2.2.2 Computation

It is important to understand what is known as Flynn's taxonomy, a classification of computer architectures proposed in the 1960s. This taxonomy gives rise to the concept of Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD). In SPMD, each process executes the same program with its own data, whereas in MPMD each process executes its own program and its own data. It is important to match the appropriate computation model to the problem being solved. Different parallel languages support programming in one or both of these forms. The benefit of SPMD is that only one set of code need be written for all processors, although this can be bloated and lacks support for optimising specific parts for specific architectures. The benefit of MPMD is that it is possible to tailor the code to run efficiently on each processor and keeps the code each processor will execute relevant to that CPU only, however writing code for each processor in a large system is not practical.

Many common parallel languages allow the programmer to write code which will mix these classifications. For instance in many languages the programmer can gain access to the processor's ID number and can write branch statements, where required, in order to issue instructions to specific processors.

Additionally, a parallel program can be written from a data or task parallel point of view. In task parallelism the program is divided up into tasks, each of which is sent to a unique processor to solve at the same time. Commonly, task parallelism can be thought of when processors execute distinct threads, or processes, and at the time of writing it is the popular way in which operating systems will take advantage of multicore processors. In data parallelism each processor will execute the same instructions, but work on different data sets. For instance, with matrix multiplication, one processor may work on one section of the matrices whilst other processors work on other sections, solving the problem in parallel. The actual problem type depends on which form of parallelism is to be employed, however as a generalisation task parallelism is often easier to perform but less effective than data parallelism, which often requires an intimate knowledge of the data and explicit parallel programming.

2.2.3 Problem Classification

When considering both the advantages of and how to parallelise a problem, it is important to appreciate how the problem should be decomposed across multiple processors. There are two extremes of problem classification -embarrassingly parallel problems and tightly coupled problems. Embarrassingly parallel problems are those which require very little or no work to separate them into a parallel form and often there will exist no dependenciess or communication between the processors. There are numerous examples of embarrassingly parallel problems, many of which exist in the graphics world which is the reason why the employment of many core GPUs has become a popular performance boosting choice. The other extreme is that of tightly coupled problems, where it can be very difficult to parallelise the problem and, if achieved, will result in many dependencies between processors. In reality most problems sit somewhere between these two extremes.

There is a common misconception that “throwing” processors at a problem will automatically increase performance regardless of the number of processors or the problem type. This is simply not true because compared with computation, communication is a very expensive operation. There is an optimum number of processors,

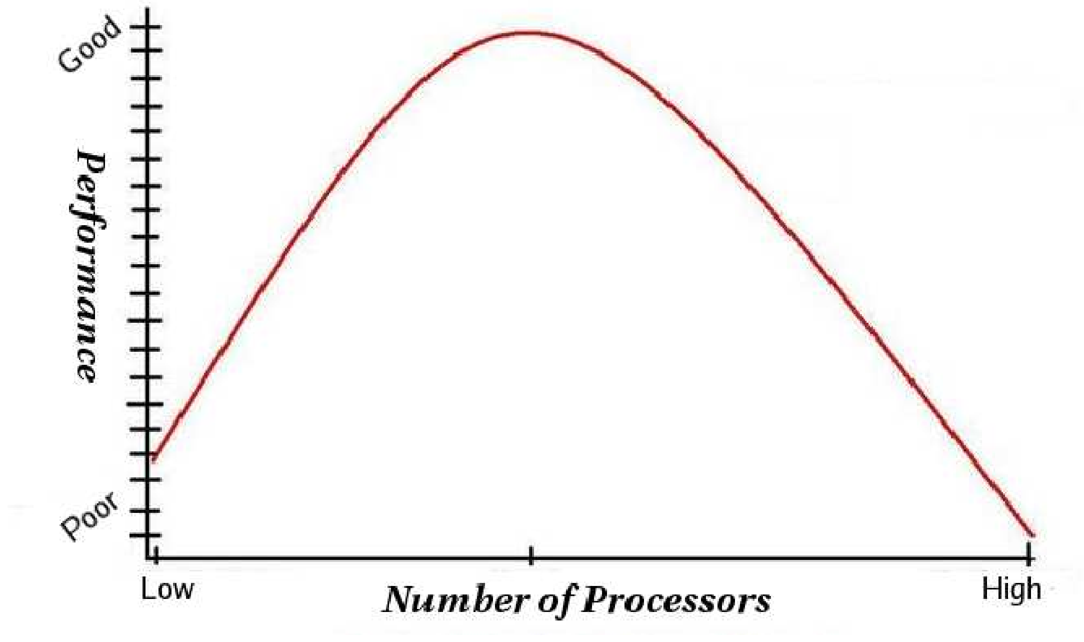

Figure 2.3: Performance vs Number of Processors

after which the cost of communication outweighs the saving in computation made by adding an extra processor and the performance drops. Figure 2.3 illustrates a performance vs processors graph for a typical problem. As the number of processors are increased, firstly performance improves, however, after reaching an optimum point performance will then drop off. In theory a truly embarrassingly parallel problem (with no communication between processors) will not be subject to this rule, and it will be more and more apparent as the problem type approaches that of a tightly coupled problem. The problem type, although a major consideration, is not the only factor at play in shaping the performance curve - other issues include the types of processors, connection latency, bandwidth and workload of the parallel cluster will cause variations to this common bell curve.

2.2.4 Summary

From the literature and their contents reviewed in this section, it is clear that parallel computing is a key field in the sciences, with many applications from climate prediction to drug discovery and engineering design. There have been numerous ways developed in which the programmer can view a parallel system and write parallel codes. It is important to appreciate the disadvantages of some models. For instance both shared memory and message passing communication models have major disadvantages associated with them. It is possible to either write conceptually simple programs or codes which elicit high performance and scalability, but not both. This is a major problem in parallel programming at the moment, the majority of developers choose performance and scalability over simplicity and as such communication aspects of parallel codes can be very complex and difficult to maintain.

2.3 Parallel Paradigms and Languages

In this section a number of different, existing, parallel paradigms and language solutions will be considered. Each of these will be analysed and a code example of matrix multiplication will be demonstrated in each, to give the reader a flavour of each language. As mentioned the aim of this project is to create a parallel programming language. There are many existing parallel languages with different applications, advantages and disadvantages. The study of these languages is very important, as not only do they give ideas as to what is required from such languages, but also ideas what is wrong with such languages so those mistakes can be avoided.

Skillicorn's paper [Skillicorn1998] begins by detailing the problem with parallel computing, then considers six criteria which are important in a parallel model or language. After this, different models and associated languages are considered with reference to the initial criteria defined. This paper is considered a good in depth review, albeit possibly slightly out of date. This survey acts to give a good indication of the models and aspects underlying the field. Found to be particularly interesting was the six evaluation criteria Skillicorn has arrived at. These were the ease of programming, software development methodology, independence of target architecture, easy to understand, guaranteed performance and the existence of costs which can be inferred from the program. The first four of these relate to the need to use the parallel model as a target for software development, whereas the last two address the need to execute the code on real parallel machines ensuring predictable performance.

Ease of programming is exactly that - how easy it is to write code in the model or language. Skillicorn considers that “a great deal of the actual arrangement of the executing computation ought to be implicit and capable of being inferred from its static description, rather than having to be stated explicitly.” Due to the high level of complexity associated with parallel computing, providing an abstract programming model is considered essential in this context. Software development methodology is the existence of *formal* development tools which can be used to prove program properties. In [Skillicorn1998] it is argued that the complexity associated with parallel machines means that the largely popular approach of testing and debugging is not suitable in the long run. Instead a process which involves building software which is correct by construction is required. The third criteria, independence of target architecture, is an obvious requirement. Parallel machines come in all different shapes and sizes, with the programmer unable to write and test their code on all possible targets. Instead the model must allow for code to be written which can be easily moved from one machine to another without any redevelopment or non-trivial modifications.

The fourth criteria laid down is that of ease to understand, where a model must be both easy to understand and to teach. As Skillicorn notes, “If parallel programming models are able to hide the complexities and offer an easy interface they have a greater chance of being accepted and used.” The fifth criteria is that of guaranteed performance. As already discussed in parallel computing performance a major consideration, this requirement states that a model must guarantee performance over a variety of parallel architectures from its design. The sixth and final requirement is that cost measures can easily be ascertained. In parallel programming, like its sequential counterpart, it is essential to be able to determine whether one algorithm is “better” than another. Cost measures exist for a model if it is possible to determine the cost of a program from its text, minimum computer properties (e.g. the number of processors) and the size of the input. Skillicorn goes on to argue that the models must provide predictable costs and that compilers should not perform optimisation of the code due to loosing this transparency.

Although it is appreciated that the criteria are somewhat subjective to the author of the paper, these six act as a good starting point for the design of the language model. One of the very interesting issues raised was, that in order to allow for the programmer to feel like they are in control, the compilation process should be transparent. By Skillicorn's own admission, these six criteria are somewhat contradictory however they do provide an insight into what a parallel programmer requires from a language.

An interesting aspect of [Skillicorn1998] is that in the conclusion the author states that although each model and associated language has its benefits, there are downsides of each and there is no ideal language. Existing abstract models tend to satisfy the software development criteria and the more concrete ones the performance criteria; from this it is obvious that a model with a happy medium is required. The author states that they believe that there will be a model created which satisfies the six criteria and this will lead to greater use of parallel computation. It is this model and associate programming language which is the aim of the project. The fact that Skillicorn overviews numerous languages and models is helpful as it builds up knowledge of what languages are currently available and the form that they take. However as mentioned [Skillicorn1998] is, as it originates from 1998, slightly out of date. In addition to this, as the author considers a wide variety of models and languages, the analysis of individual languages can be quite narrow and often confined to the six criteria considered. Within the last nine years, a number of new languages have appeared, many of which will be considered in this chapter.

2.3.1 Sequential Languages

An initial question is whether parallel languages are essential or not. There are a number of existing compilers which will automatically parallelise sequential code. In the paper [Lou2005], a computational physicist, with considerable technical experience, took a number of popular parallel benchmark codes written in Fortran 77. He then removed all the code for distributed memory, source level optimisations and none portable features, after which modifying it such to take advantage of Fortran

90's language features. Just two factors were considered in the experiment - code size and execution time.

The code size reduction was dramatic, ranging from a 4 to 11 times reduction in size. The paper estimates that on average about 2 times reduction was due to the removal of explicit parallelism. This newly re-factored code was then timed. The new code was, at best, 2 times slower and for one benchmark performed 6 times slower than the original. In the conclusions the authors of [Lou2005] state that "At the cost of a relatively modest performance penalty at run-time, HPC software written in FORTRAN 77 can be improved through perfective maintenance." Whilst this is true it is thought that the "relatively modest performance penalty" of between 2 and 6 times slower would be too great for many parallel programmers to adopt this approach. Maximising performance is hugely important to many of these programmers, so whilst this approach is simple it does lack on the performance side. Additionally, the programmer must rely on the compiler for decisions about important parallel aspects such as computation distribution and communication. This loss in expressiveness is a great disadvantage because many parallel programmers wish for a great deal of control over their code.

2.3.2 Paradigms

LOGS

The LOGS [Chen2004] model has been used as a basis for a parallel language, with a translation tool written [Zhou2005] to convert it into C code using BSP. The LOGS code has introduced into it a number of additional commands. The first one is an early transition, which is a 1-step command that may change the state before the synchronisation point but will maintain a state between the intermediate and final state. Most data parallelism-based computations use early transitions. Secondly, late transitions keeps a state up to the synchronisation point, but may have a different final state from the intermediate state. Late transitions are more aimed towards task parallelism. In addition to this, the *after* command has been introduced which stands for the final state of a program variable, and a *before* command which represents the initial state of a program variable. The concrete language also allows for parallel composition and *par* loops, which is the parallel equivalent of sequential composition and a sequential for loop respectively.

The result of [Zhou2005] is a convenient parallel programming language. A major downside though is that the language is not finished and, although it demonstrates the concepts well, it is not usable by the end programmer. In addition having each variable automatically shared, following the shared memory model, does have its problems as it is often the case that only a small subset of variables are required for communication. Due to the fact that the language has not been finished, there are a number of programming annoyances. In this context the paper [Zhou2005] is considered a very useful resource, as not only does it explain the language in detail, but it also explains the translation process associated with it. However, unfortunately the paper does not detail some of the more advanced issues such as array access, which would have been useful.

Skeleton Functions

Skeleton functions attempt to overcome the limitation of parallel functional languages such as NESL [Blelloch1995]. Numerous skeletons are provided, each is a higher order (functional) form aimed at accomplishing a specific task. The parallel programmer can use and combine these skeletons to form the building blocks of an application, and in order to maintain portability, transformations are provided between the skeletons.

The paper [Aldinuccia2000] explains that parallel systems can be created by the combination of basic skeletons. These skeletons include ones for modelling embarrassingly parallel problems, computations structured by stages (known as pipes) and common data computation patterns such as map, reduce and scan. Each skeleton will take, as a parameter, the computation to model. For instance, the pipe skeleton will take, as a parameter, the stages which might be sequential portions of code or other skeletons. A considerable amount of research has been carried out in this topic and, whilst it has not reached mainstream popularity, there has been some success. However these skeletons still have strong roots in functional programming and, as such, the abstract nature of the model limits programmer expressiveness and places reliance on compiler optimisation.

2.3.3 Languages and Libraries

MPI

The Message Passing Interface (MPI) [MPI1995] is a standard which provides for message passing communication. Before MPI, there were many different message passing standards. This standard, written in 1994, aimed to compose the best features of each and become the defacto. The interface takes the form of a library, aimed to be accessible from at least Fortran, C and C++ and bindings exist for many other languages. There are many different implementations of this standard, aimed at different architectures.

There is much literature on the subject available. Because the standard is aimed, not only at computer scientists, but scientists as a whole, the literature varies greatly from some very detailed to some only providing a general overview. The MPI standard [MPI1995] is the official document detailing the standard. Apart from the description of this standard, some examples are included to illustrate some of the more complex ideas. This resource is a very useful one and, if followed when writting parallel programs, will guarantee that the result will be compatible with all MPI conforming implementations. However, being a standard the document is not always particularly useful as a learning tool, and does not detail implementation in as much detail as required for specific programming. A number of important concepts are also just listed, rather than fully explained and some points require better illustration.

In order to understand and learn MPI a number of other resources have been employed. [Gropp1999] takes the reader from basic MPI to advanced topics. Included, there are numerous code examples illustrating the power and use of different aspects of the MPI standard. One downside to this resource is that even though some of the MPI functionality is covered in great detail, there is a proportion which is hastily covered or not mentioned at all, as if the author did not deem it important. [Gropp1999] also does not mention about the efficiency of different MPI calls, whereas the standard does offer some idea towards optimisation (although in the most case this is very much implementation specific.)

As mentioned above, the MPI standard has numerous implementations. There are architecturally independent implementations such as MPICH and OpenMPI, and vendor MPI versions such as SunMPI for specific classes of machine. As mentioned in [Foster1997] vendor MPI often provides for a more efficient implementation due to some functionality being implemented in hardware. However, from experimentation it has been discovered that the vendor can leave out certain functionality which is not considered to be important, meaning that it can be difficult to deduce exactly what version of the standard is implemented. The paper [MPI1995-2] details the MPICH implementation. This paper provides useful information on how to use the implementation and what is supported. The paper also details the efficiency of this implementation. MPICH is one of the more popular MPI implementation, OpenMPI is also commonly used.

Combining C with MPI is a very popular option, code listing B.1 implements matrix multiplication using this combination. From the code it is clear that this form of parallel coding is difficult and even a simple example requires a relatively large amount of code. As [Gropp2005] notes, MPI became popular because, at the time, it allowed users to get simple parallel codes up and running quickly in a language that they were familiar with. However, as parallel programs have become more complex so has the difficulty in using this option. This resource also details the latency involved in the MPI library, although it is implementation specific. Also shown is a comparison between compiler optimised C MPI code and hand optimised code - there is considerable difference, with the hand coded optimisations being much more efficient. One reason for this performance gap is that, as the parallel system is described in such a low level, there is not rich enough detail to get a high level view of the code. As [Gropp2005] notes, MPI is the wrong model due to the lack of abstractions making parallel programming a difficult task. Interestingly, the author mentions that the aim of a language should not be to make easy programs easier to write, instead it should be to make it possible to create difficult programs.

Additionally C is the most common language to use with MPI; the result being that it is very easy for the programmer to get lost in the small details of their code and loose track of the parallel program as a whole.

BSPLib

"BSPlib is a small communications library for Bulk Synchronous Parallel (BSP) programming which consists of only 20 basic operations." [Hill1998-2] This popular library implements the BSP shared memory model, which has already been reviewed, and is commonly used in conjunction with languages such as C for creating parallel programs. [Hill1998-2] is a paper written about the library, detailing it and explaining, using examples, how to write BSP programs. [Skillicorn1999] provides answers to a number of possible questions about the BSP model. This paper is both interesting and useful, and does address some important concerns that could be raised such as performance aspects. Interestingly, a Fast Fourier Transformation (FFT) example has been created to demonstrate the power of the library.

The code in listing B.2 demonstrates how one would implement a very simple matrix multiplication in BSP. For readability the matrix filling function has been omitted. In this code process 0 will fill the two matrices with data, each process will register, via *bsp_push_reg*, that arrays *matrixa*, *matrixb* and *matrixanswer* are to be globally visible. The BSP *bsp_sync()* call is then made to synchronise all processes. The call *bsp_get* will instruct each process to copy the filled matrices from process 0 into their own arrays, communication is performed to achieve this on the second sync call on line 34. In order to sum each process's copy of the matrix the (extended) BSP collective communications call *bsp_fold* is called. The last sync call on line 48 will perform the communication required for the fold, with the resulting multiplied matrix located in each process's copy of array *result*. This code demonstrates a number of downsides of BSP. Data which is globally visible must be allocated to all processes - in some cases this will result in wasted memory. For instance, in this example, if the result of the matrix multiplication was just required on one processor then memory would be wasted allocating array *result* to all processes. The programmer is also stuck writing code SPMD style, quite often this is not the most convenient form. BSP does provide, as an extension, some limited collective communication commands such as *fold*. By the author's own admission these are based upon the MPI collective calls. However the BSP collective communication library is far more limited than MPI, for instance *fold* will only operate on one data element.

It has already been mentioned that a major issue with BSP, and by extension BSPLib is that of performance. Code written using BSP just does not compete with code written using, for instance, MPI. Although the shared memory model is greatly simpler than the message passing one, when combined with a language such as C the programmer is still stuck in a low-level form of abstraction. Often with parallel computing the programmer needs to take a high-level view of what is happening, having to consider low level issues such as pointers really does stop this. As such, being shared memory does make it somewhat easier to code, however this simplicity is really not enough to warrent such a drop in performance.

Cilk

"Cilk is a multithreaded language for parallel programming that generalises the semantics of C by introducing linguistic constructs for parallel control." [Frigo1998] This extension to C provides the programmer with simple constructs which they can use to create a parallel program. The Cilk extension works from two angles, firstly during compilation the C is analysed statically and calls to the Cilk parallel library are added into the C postsource which is generated (ready for compilation by a normal C compiler.) Secondly, during runtime C code will call the Cilk parallel library which will actually support the parallelism. As [Frigo1998] mentions, Cilk's parallelism is limited to Symmetric Multiprocessors (SMP). "An SMP architecture is simply one where two or more identical processors connect to one another through a shared memory" [Jones2007]. SMPs are scalable to a point however for complex scientific problems, whose solutions are often found by parallel computing, which require many processes possibly distributed over a number of locations this model is completely unworkable. Due to this limitation, there are generally far more processes than processors which have to be queued up.

The code in listing B.3 illustrates how one would program parallel matrix multiplication using this language. The code looks like C code at first glance, with two extra keywords *cilk_spawn* and *cilk_sync*. The first, *cilk_spawn*, instructs the special compiler to place the function following the keyword in a queue for parallel execution whilst the code continues. The second keyword, *cilk_sync*, will pause program execution until all spawned (parallel) functions have concluded executing.

The paper [Frigo1998] focuses consideration on task parallelism rather than data parallelism which is interesting as many parallel languages are targeted towards data parallelism. Process scheduling is a key aspect of Cilk and one which the team have explained in detail in this paper. Cilk follows a “work first principle” where the scheduling overheads produced by the computation are reduced. The notion of critical path length is introduced which is the total execution time on an infinite number of processes and corresponds to the sum of the largest thread execution time along any path. By using this metric the programmer can estimate the runtime of their code.

The scheduling algorithm assumes parallel slackness, where there many more processes than processors. This algorithm maintains a queue on each processor, which holds a list of processes to execute. As the processor works through the list, if a queue becomes empty, then it can “steal” a process from another processor. This scheduling algorithm has a number of considerations attached to it, which the authors discuss their solutions for. Interestingly, the author mentions that there is a tradeoff between portability and efficiency. A problem with this approach, which is evident, is that the scheduler assumes that there is parallel slackness. If this property were not present then efficiency would be lost. The paper mentions that slackness is commonplace and the evaluation covers a number of different Cilk examples with full details as to their different timing results.

The evaluation in [Frigo1998] concludes that the assumptions made are appropriate. However, out of the twelve example programs considered, there were no hardcore scientific parallel programs considered apart from FFT although no details are supplied about the complexity of this program. The paper is considered an interesting, relevant resource. However, only a small section of the implementation is mentioned and it is not clear what other bottlenecks exist. It should be noted that Cilk is designed for a vastly different architecture than the target of this project and so the assumptions and decisions made are most likely not applicable. In the absence of detailed information about the test programs used during evaluation, the results should be viewed with some degree of uncertainty. However, from this it is obvious that SMPs with many cores are a real future for desktop machines and when the time approaches when many processors are placed upon a chip, then the programmer will need to write parallel programs due to the shortcomings of the current threading algorithms. It is not believed that any such parallel languages, Cilk included, provide an adequate model at the present for this. Relating to this point, in the not too distant future, parallel computing will most probably stop being the within the exclusive domain of high performance computing and move much more into the general computing field.

High Performance Fortran

High Performance Fortran (HPF) [HPF1997] is an extension of Fortran 90, with constructs supporting parallel programming. The approach adopted by HPF is to require the programmer to specify data partitioning and allocation, and then have the compiler automatically infer how to distribute computation accordingly. Lastly the compiler will insert communication, as required, to support the parallelism. This implicit model does have its disadvantages, with the programmer having to rely on the compiler's "best guess". Additionally, due to the vast amount of work done by the compiler, often HPF programs are not transparent. The HPF programmer does not have this option of optimising their code, and must rely on the HPF compiler's default solution.

Much work has been done in designing, creating and supporting HPF [Richardson1996], with much literature available on the subject. HPF was the result of a massive standardisation process, although considered ultimately unsuccessful by many, some of the better known HPF projects include parallel programs on the Earth Simulator [Yanagawa2004].

The code in listing B.4 illustrates a parallel matrix multiplication example written in HPF. As can be seen from this example the programmer is determining parallel aspects of the code such as the number of processors, distribution of data and alignment of data via *PROCESSORS*, *DISTRIBUTE* and *ALIGN* keywords respectively, with the compiler taking care of the rest. An interesting aspect of HPF is that all parallel details are provided as comments, so that the program is also perfectly acceptable to a normal Fortran compiler. It is considered that having these, two views of the same program is a very useful attribute because it allows for the code to be easily run serially or in parallel as required.

Co-Array Fortran

Co-array Fortran (CAF) [Numrich1998] is an extension to Fortran 95 for explicit parallel computing. CAF provides extensions to add an optional co-dimension which, when attached to normal objects, makes them co-objects. These co-dimensions allow the programmer to represent indices across processors and hence allow communication, circular braces () represent on a local processor, whilst square braces [] are non-local. The declaration *real :: x(n)[*]* will declare an array x of size n and locate this on all processes.

In this language the program is written SPMD style, with the programmer responsible for distributing computation. The programmer has more control over parallelism in CAF than in HPF and they can explicitly control data partitioning, computation and synchronization. However all communication in CAF is one sided and, as this is dealt with exclusively by the compiler, often messages between processes are short resulting in extra communication overhead. Additionally, there is limited expressiveness as only local or global data is supported. The computational explicitness means that index management by the programmer is required for arrays, which is made even more problematic when data does not divide evenly.

A major drawback of CAF is that, although it does provide some level of expressiveness, the programmer is limited in what they can control. Often the programmer can be in a better position than the compiler to correctly and efficiently decide upon communication, by abstracting this away from the programmer will mean that in some cases performance is sacrificed. Although the programmer is abstracted away from the low level details of communication still they must consider details such as index management, which forces them to work low level rather than allowing for a high level view of the parallel computation. Because the programmer must write code in SPMD style, not only does this affect the programmability, it also limits compiler optimisation (specifically dependence analysis is difficult) because this model is asynchronous.

The code of listing B.5 is an example of matrix multiplication using CAF. The programmer declares the arrays *a,b* and *c* to be co-arrays. As work is done in the code with these arrays then communication is inferred, as required, by the compiler.

ZPL

ZPL [Chamberlain1998], a parallel array programming language, takes advantage of the fact that common HPC applications often involve working with arrays of data with communication in ZPL being inferred by the compiler. In array programming, in order to combine two arrays *A* and *B* into *C*, the statement *C:=A + B* performs the same job as looping through each element as is required in mainstream languages.

As [Deitz2003] shows, array programming and abstracting the programmer away from parallel details does work well in some problem cases. However by tying the language to array programming and making parallel details (such as problem decomposition and communication) implicit, ZPL is not sufficient for use in solving many parallel problems neither in terms of simplicity or efficiency.

Listing B.6 shows example code for matrix multiplication written in ZPL. In the section *config var* the programmer is simply setting up the parallel environment. ZPL has as a fundamental concept the notion of regions. Regions are index sets and rectangular in nature, the regions being declared in listing B.6 are two dimensional. Regions are used both for parallel arrays and to provide indices for array references within the language. For instance, in the *var* section of the code the programmer is using these regions to declare the parallel arrays *A*, *B*, *C*, *Aflood* and *Bflood*. The procedure *Summa* is where the actual work is done, with the programmer using the parallel arrays. There are some strange operators, such as $\gg$ which is the flood operator, replicating a slice of an array's values. Array programming can be seen at work in the statement *C += (Aflood * Bflood)* which will multiply array *Aflood* with array *Bflood* and add the values to array *C*.

From viewing the code in listing B.6 it is obvious that, although the program is not particularly complex, a variety of new concepts must be learnt before the programmer can take advantage of ZPL. Additionally these concepts are hardcoded into the language, making it difficult to modify or remove them at a later date. Once the programmer has learnt these new ideas then writing code in ZPL is not particularly difficult, but there is quite an initial barrier to entry.

NESL

NESL [Blelloch1995] is a functional parallel programming language developed at Carnegie Mellon with the main ideas being nested data parallelism and the provision of a language-based performance model. In nested data-parallel languages [Blelloch1990] any function can be applied over a set of values, including parallel functions. For example, the summation of each row of a matrix could itself execute in parallel using a tree sum. Nested data parallelism is useful especially in order to implement nested loops and divide and conquer algorithms in parallel. Nested data parallelism is an interesting concept and applicable to many problem domains.

Language-based performance models allow the programmer to formally compute the work and depth of the algorithms developed in the language on parallel machines. Work is (simply) defined as the running time over one processor, whilst depth is (again simply) defined as the running time over unlimited processors. Due to the abstract, functional, nature of the language, NESL does allow the programmer to easily find these attributes of their code. Being a functional language means that the programmer writes code by specifying what, and not how, when it comes to problem solving. This means that the programming language is very abstract from the specific considerations of parallel programming and the actual hardware that their code will be executing on. Using this approach, there is much emphasis placed upon the compiler to infer the "best guess" when it comes to dealing with parallel communication and computation. As the programmer has no way to control these aspects, not only does it cause a problem with efficiency in some cases, it also means that dealing with the parallel attributes of code written in NESL is opaque. Whilst it is possible to find the work and depth of a particular algorithm, it is not possible to determine the running time using this abstract model without knowing the intricate details of the compiler.

The problem described with NESL is applicable to all functional parallel programming languages. As the problem of parallel programming has become more and more exposed by the computing community in recent years, many have touted functional programming as being the answer. However, the abstract nature of the functional model means that this is not the "magical" solution some believe that it might be, not at least until large improvements in compiler technology are made to support the programming paradigm.

The matrix multiplication code example (listing B.7) illustrates how NESL is used to solve parallel problems. As can be seen from this code, the programmer is very abstract from what is actually happening which, as already discussed, makes optimisation very difficult. The programmer is entirely dependant on the efficient implementation of language defined functions *sum* and *transpose*.

Titanium

Titanium [Hilfinger2005] is an explicitly parallel version of Java. The advantages to the programmer are that this language is safe, portable and it is very possible to build complex data structures using the OO abstraction. Similarly with CAF, there is a global address space (shared memory) and synchronisation constructs are supported (although the compiler will ensure that synchronisation does not cause deadlock.) Another shared feature with CAF is that the programmer is stuck writing code in SPMD style. Additionally, object orientation can impose a hidden cost and is not transparent.

In the paper [Baker2006], Shafi discusses the performance of porting Gadget-2 into Java. He notes that originally, the Java version was around 3 times slower than the C version. One major slow point was in the communication of objects. In serialising (converting to a byte array) and deserialising (restoring to an object) for communication Java imposes some overhead which had a major impact. To solve this objects had to be replaced by primitives when dealing with communication. Another issue was maintaining memory locality, the Java Virtual Machine (JVM) does not recognise the importance in HPC for related data to be located together, with objects being located in different places of memory. The Java programmer has no control over this and as such Shafi had to replace object arrays with primitive arrays in sensitive parts of the code. Of course there are some differences between Titanium and Java, but Titanium is based upon Java and as such suffers from the same OO overhead issues. From reading the optimisations performed in [Baker2006], one has to wonder if the resulting Java code really is much simpler and more abstract than the C code it was based upon.

Code listing B.8 provides an example of a simple parallel application in Titanium. The arrays in this example are Titanium arrays, supporting parallelism. Titanium arrays are indexed via points such as *(1, ij[1])* on line 3.

2.3.4 Summary

	Imperative	Functional	Library Extension	OO
Message Passing	-	-	MPI	-
Shared Memory	HPF,CAF,ZPL,Cilk	NESL	BSPLib	Titanium

Table 2.1: Overview of Parallel Languages Considered

There are many existing parallel programming languages, following many different programming paradigms and communication models, some of which have been considered in this section. It is appreciated that there are many other parallel languages such as UPC, OpenMP and PVM which for brevity have not been considered in this section. Table 2.1 provides an overview to which programming paradigm and communication model each language belongs. However, for all these different languages, none ideally suit parallel programming. Generally, those which try to simplify parallel programming often impose abstractions which work well only in specific cases and those designed for efficiency require the programmer to consider low level details making programming difficult and error prone. The most common form of parallel programming (C with MPI) is, as mentioned, the completely wrong model. It is too low level and does not provide the necessary abstractions, because the programmer can very easily get lost in the mechanics of communication. By referring back to [Skillicorn1998], it is possible to evaluate existing languages with respect to the six criteria considered in this paper.

	HPF	CAF	ZPL	NESL	Titanium	MPI	BSP	CILK
Easy to Program	Y	Y	Y	N	Y	N	N	Y
Software Dev Methodology	N	N	N	Y	N	N	N	N
Architecture Independent	Y	Y	Y	Y	Y	N	N	N
Easy to Understand	Y	Y	Y	N	Y	N	N	Y
Guaranteed Performance	N	N	N	N	Y	Y	Y	N
Cost Measures	N	N	N	N	N	N	Y	N

Table 2.2: Parallel Languages considered wrt evaluation criteria

Table 2.2 provides an overview of this evaluation, although whether a parallel language or library meets a criteria is contentious in some cases. The paper [Gropp2005] considers whether or not MPI is easy to program or understand; in this case Gropp makes note of the fact that the lack of abstractions do make parallel programming using this a difficult task. Although based upon a somewhat easier communication model than MPI, BSP is still most commonly used from a low-level sequential language such as C. The statements of Gropp [Gropp2005] are true for this model too - the lack of abstractions imposed by the choice of language when using BSP make parallel programming difficult. Although Cilk is based upon C, as [Frigo1998] introduces, code written in this language is sequential C code with only two additional, simple, keywords. Because of this fact the Cilk programmer need not consider in depth about parallel issues, indeed experience with threading will be sufficient, and as such it is easy to write parallel code in this language. For functional languages, such as NESL, the two criteria of "easy to program" and "easy to understand" are addressed by Hinsen; "Functional programming is very different from traditional programming and thus requires a lot of learning and unlearning." [Hinsen2009]

Commonly using the MPI and BSP standards with or basing a language around C causes another problem, namely that of architectural independence. As Hook makes note of in his book "ANSI C and C++ are probably the most unportable languages that were still intended to be portable." [Hook2005] Using C it is far too easy for the programmer to write unportable code, without realising it, thus limiting their MPI, BSP or Cilk application to a specific architecture. The other languages, because of their higher-level nature, are all architecturally independent due to the compiler handling much of the lower-level, and in some case parallel, details.

Considering Skillicorn's 5th criteria, guaranteed performance, the report [Luecke1997] concludes that HPF is not practical when is comes to performance for many common codes. A major contributing factor to this is the use of implicit parallelism, as [Mozafari2008] makes note, ZPL also greatly makes use of implicit parallelism and as such the compiler is responsible for many important parallel decisions with only a limited amount of information to work from. Both these languages suffer as a result of this decision and as such neither meets the guaranteed performance criteria. In terms of Co-Array Fortran, whilst is does afford the programmer more control over parallelism, the compiler is still responsible for a number of important parallel decisions such as communication, without direction from the programmer it is often difficult for a machine to optimise this aspect. Dotsenko, one of the CAF developers at Rice University, makes note "without compile-time optimization of communication, including vectorization and aggregation, we have not yet realized our vision of supporting portable high-performance applications written in a natural style" [Dotsenko2004].

As one can see, no specific language meets all Skillicorn's criteria in table 2.2. There have been many technical advances in the field of high performance computing, however, the programming languages used have lagged behind.

2.4 Theories of Programming Languages

The study of the theories of programming languages is an important one within the context of this project. The theoretical background to programming language design is a large, complicated field which ties together a number of different aspects of Computer Science. Within this section, the theories which are related to the creation of a parallel language are studied along with the literature which introduces them.

[Wirth1974] provides an interesting introduction to the area of programming language design. Although this reference is quite dated, it mentions numerous useful facets. A main trend in the paper is that simplicity is important, not necessarily in reducing the number of features but instead keeping the language simple to understand, which it suggests abstraction is useful for. Also mentioned is that a language should not imply any unexpected features, and that it should be transparent to the programmer. The author then notes that language design really is closely related to compiler creation, "a successful language must grow out of clear ideas of design goals and of simultaneous attempts to define it in terms of abstract structures, and implement it on a computer." [Wirth1974] This is interesting, as it states that in order to create a successful language, a combination of different methodologies are required, from theoretical to practical. Specifically from this paper it should be noted that in order to create a useful language then it is required to know how this is to be used, an efficient reliable compiler is required, as much analysis during compile time (static analysis) should be performed and a complete simple sketch of the language should be created before work on the compiler. When considering a language's simplicity this is not the lack of features but instead transparency, clarity of purpose and integrity of concepts.

It is thought that [Wirth1974] is a useful introduction to language design. Even though this paper is old and some of the examples are slightly out of date, it is easy to see that much of the advice given by the author is timeless and applies to all languages. Even though [Wirth1974] was aimed at sequential language developers, there is no reason why the advice can not be transferred to parallel language designers albeit with possibly more requirements.

2.4.1 Syntax

"Syntax is the way words are put together in a language to form phrases, clauses, or sentences." [Sil1999] The study of syntax is key to programming languages due to the specification of syntax acting as a basis for language description and tool development. It is the norm to specify syntax in terms of a context free grammar. These grammars are powerful enough to specify many of the languages that are in use today.

Commonly used to specify a grammar is Backus-Naur Form (BNF). The book [Reynolds1989] has a short introduction to this using it as a basis for introducing more complex concepts. When studying the theoretical aspects of programming languages, it is often inconvenient to have to worry about all the small syntactic details. Therefore, often an abstract syntax is used, which captures all the important syntactic elements but allows the designer freedom from small details such as parenthesis. [Reynolds1989] covers the topic of syntax, BNF and abstract grammars sufficiently for use theoretically. This resource also mentions that, as good as syntactic descriptions are, often they can either be meaningless or ambiguous, thus requiring a better way of describing a language.

It should be noted that there are a number of tools available which will take a grammar and generate a lexer and parser for it, acting as a good starting point for a compiler. The compiling book [Aho2006] details, amongst other aspects, this step of compiler creation. This resource is very useful and, although these tools will not directly affect the project, it is important to have a good understanding of what is currently available and how they are used. As [Wirth1974] notes, it is important to consider the syntax of a language, because a complicated syntax will often produce a difficult language for a programmer to use.

2.4.2 Semantics

“Semantics refers to the aspects of meaning that are expressed in a language, code, or other form of representation.” [Wang2007] The study of semantics is key to programming language design, as not only do they act as a specification for a language but also as a basis for tool development, they allow for certain facets about a language to be proven and even assist in presentation of the language. There are a number of different semantic models, each with their own advantages, disadvantages and researchers. An important part of this survey was not only to comprehend these different models, but also to understand how the theory is applied in a practical way, which is after all of the most relevance in the context of the project.

Axiomatic Semantics

This model of semantics was first introduced by Hoare in 1969 by the paper [Hoare1969]. Hoare was the first person to model these semantics directly to programming languages, although Floyd first considered this topic in [Floyd1967] referencing to a simple flowchart. Axiomatic semantics is most useful in proving program properties and is still popular to this day.

The whole idea of axiomatic semantics is that a program (Q), or portion of a program has a pre(P) and post(R) condition. This implies that, if the pre condition is true before execution then the post condition will be true after execution has finished. It is written commonly as P Q R. Program properties can be proven by providing a number of axioms to use as a basis for inference. [Hoare1969] has a good example of a formal proof although in order to prove even a simple lemma, twelve steps are required, which begs the question whether or not this is practically suitable for a large program. [Floyd1967] also has a number of different proofs mentioned in the context of the flowchart language. An interesting remark by Hoare in [Hoare1969] is where he states that currently (in 1969) programmers tested their code by running it using different inputs and then modifying it if the desired result is not obtained. Hoare goes on to explain why this is a bad idea and how proving properties will outweigh the cost of testing by this method. However, this method of testing is still very common today and only a small number of programmers actually provide formal proofs for their programs in a small number of cases. It is interesting that, although the benefits of this proof approach to testing is obvious, it is still not as widely used as Hoare foresaw. Skillicorn in [Skillicorn1998] makes note that, as part of his second criteria (software development methodology), parallel programs should be correct by construction. He goes on to say that the familiar process of testing and then debugging is not suited to parallel computing, which echos comments made by Hoare some 30 years previous.

In comparing the two papers, [Hoare1969] is considered to be more relevant to this project due to the fact that it actually references to a programming language instead of a pseudo language described by a flow chart and small fragments of ALGOL as in [Floyd1967]. However, [Floyd1967] does go into detail about program proving and explains in details different axioms and provides examples as to how these are used. [Hoare1969] does mention some commonly used iterative programming axioms, such as the axiom of assignment, and a number of general rules within the context of iterative programming.

In addition to the papers mentioned above, the book [Winskel1993] has a chapter on the topic, bringing together all the different aspects. Unlike Hoare and Floyd, the book considers the semantics in the context of a specific state, which is more realistic to real world languages. This book also mentions proofs in detail and has numerous examples. It is considered that [Winskel1993] is a powerful resource within this context and brings together a number of concepts mentioned in [Hoare1969] and [Floyd1967].

[Winskel1993] also mentions that axiomatic semantics only guarantees partial correctness. Partial correctness is where, if the program terminates then the result is the correct one, however it does not guarantee that there is always termination. Total correctness not only guarantees the correct result, but it also guarantees termination. Total correctness was introduced by Dijkstra in [Dijkstra1975]. In this paper, Dijkstra introduces the idea of a guard, which must be true before execution and satisfies total correctness. The paper goes on to provide a number of examples with respect to common program constructs and also mentions the weakest precondition, which is the weakest condition which satisfied total correctness. The weakest precondition is known as a "predicate transformer" because it associates a precondition to any postcondition and the semantics of a specific program or portion is known sufficiently well when one knows the predicate transformer. Many people cite [Dijkstra1975] as providing a theoretical basis for today's imperative sequential languages. This resource is very helpful and it can be seen why, although only a small imperative language is considered.

When considering these resources, as already mentioned, it is considered that [Hoare1969] is more applicable to "modern" programming languages than [Floyd1967]. [Dijkstra1975] is a useful paper and ties together many of the concepts discussed in the other two. This paper also applies the semantics to real examples and defines many core programming constructs such as the alternative and repetitive constructs. It is believed that [Hoare1969] and [Dijkstra1975] do complement each other well and will both be considered. Concretely, when initially considering the form that the language source code will take, Dijkstra's guarded command language (albeit with some additions) detailed in [Dijkstra1975] has been used extensively. This has allowed for a concise description of the source language and also as a basis from which these semantic ideas can be applied.

Operational Semantics

"Operational Semantics involves giving a precise description of the behaviour of a program or a system, namely, how it may execute or operate" [Prasad2003] This semantic model is practically based, with the underlying idea being that at each point a program has a particular state and as execution progresses the state will change. This state corresponds to the state of the compiler, and so operational semantics is useful as a basis for tool development as it provides an unambiguous language reference. There are a number of different literature resources which have been used to study this model, each has its own positive and negative attributes.

[Prasad2003] introduces, like much of the literature, Structured Operational Semantics. Description of semantics occurs at different levels of abstraction, which is useful for different forms of work on the language and, at some abstraction allows for small details to be ignored and at other levels using the same semantic model allow for these details to be very much considered. The model adheres to the compositionality principal, which says that if the meaning of a set of components can be deduced then so can the phrase they combine to form. Syntax directed inference rules are the standard way of presenting the semantics, with many proof techniques being based on induction of these trees. This model has two broad flavours of abstraction. Firstly, big step semantics which provide a complete execution overview and secondly small step semantics which provide justifications for each step of computation. In order to provide a flexible model, many designers use a mixture of these two abstraction flavours.

Simply, operational semantics are expressed with respect to an environment function which maps variables to values. Big step semantics, as mentioned above specify the normal form (the result of computation) without any specific information as to how it is achieved. Small step semantics specify not only this normal form but also how this can be achieved. Each step in small step semantics reduces a phrase, until it is no longer reducible (normal form.) A part of a deducable phrase which can be reduced further is called a redex. There are a number of different reduction strategies available, [Prasad2003] provides as an example for a language with boolean expressions, compositional evaluation (which evaluates all parts), left sequential evaluation (which first evaluates the left part and only carries on if required) and parallel evaluation (which evaluates parts in parallel.)

When considering a complete language there are numerous program constructs which need to be addressed. A state needs to be provided which relates to the current state of execution (this is simple for a simple language, but when complexities such as scope and procedures are added then the state becomes a more complicated function) and other program constructs such as expressions, numerals, boolean values and commands (which can be seen as mapping one state into another.)

As mentioned above, there are many different literature resources on this subject. A resource that has been relied on heavily is [Prasad2003] which acts as a good reference and provides plenty of examples although many of the related foundation terms which are used are not explained in detail. [Winskel1993] also has a chapter on operational semantics, although it is more concise than [Prasad2003] it does introduce many of the important concepts. [Reynolds1989] supplies a sound introduction to this topic, although at a first read the concepts are theoretically based and require some work to understand. There are two major issues with [Reynolds1989], firstly the author does not differentiate between the different flavours to achieve levels of abstraction, which the other two resources do and secondly the examples rely heavily on language functions already developed using different semantic models, which means that it is very difficult to simply reread a chapter without re familiarisation with the preceding chapters. Out of the three resources mentioned, [Prasad2003] is considered to be the most useful as it describes a complete language with many examples considering issues such as scope, parameters and I/O. At this point it should be noted that, due to the length of inference rules, often operational semantics is not ideal for presentation purposes, which can be seen from the examples provided in the literature.

Denotational Semantics

"Denotational semantics is a technique for defining the meaning of programming languages pinoeered by Christopher Strachey and provided with a mathematical foundation by Dana Scott." [Tennent1976] This model of semantics allows for the description of a language to be at a more abstract level than its operational counterpart, where the denotation of an expression is a partial function on states. [Tennent1976] defines a semantic interpretation function as a mapping between the syntactic construct of the source language into its abstract meaning within the framework of a mathematical model. This interpretation function is the basis for denotational semantics.

After deciding upon the syntax of a language, in order to start specifying denotational semantics, appropriate interpretion functions should be given. For example, a common expression function maps a state to a number and a common command function maps a state to a state. Depending on the language in question, a state is a function which maps each variable into a value (e.g. an integer.) In denotational semantics, the symbols *[[* and *]]* are used to encapsulate syntactic elements to separate them. For example if a state σ*:S* $= Var \rightarrow N$, then for any variable *name*, σ *[[name]]* is equal to the contents of *name*. In order to change a state, σ*[[r/name]]* represents the new state, where that value of *name* is now *r*. In addition, state transition functions are specified, which result in the state after execution of a certain command. A complicating factor is the idea of nontermination. If one wishes to consider nontermination within the context of denotational semantics, then as [Reynolds1989] explains, the symbol $\perp$ is used to denote this and is mapped into a state, thus a state S is now represented by $S\perp$

As [Tennent1976] explains, a problem with denotational semantics was that quite often it is natural to model a language using higher order functions, however these cause problems mathematically due to the fact that they allow general recursive definitions and the fact that this allows self application. [Reynolds1989] also shows that this is a problem in relation to the while command as if it is defined with reference to itself, then the semantics is not syntax directed because the meaning of the phrase is not in terms of exclusively its constituents. Scott has solved this problem by characterising a class of data types, called domains. The idea behind domain theory is that a sequence of better and better approximations in a domain should converge to a limit in the domain. A domain also consists of a bottom, $\perp$ which denotes undetermined information which is an approximation of all elements in the domain and a top $\top$ which represents overdetermined information which all other elements in the domain are approximates of. An example of a domain is $\perp \sqsubseteq 1 \sqsubseteq ..... \sqsubseteq i \sqsubseteq \top$ Building on from domains, is the least fixed point theorem, stated by [Reynolds1989] that if D is a domain and f a continuous function from $D \rightarrow D$ then if *f(x)* $= x$ and *f(y)* $= y$, then $x \sqsubseteq y$. In this context, Yd is the function which maps continuous functions from D to D, into their least fixed points. From this mathematics one can then define recursive definitions, including the while command.

When considering the difference between the operational and denotational models, [Tennent1976] mentions that, because nothing is specified about how the functions are computed or represented in denotational semantics. A description using this model simply requires the implementation to compute the correct result. However, the operational model formalises the language implementation methods so their correctness can be verified. It can therefore be viewed that denotational semantics is more theoretically based around a language than operational semantics. For presentation reasons, denotational semantics can often be preferable over operational due to the more succinct nature of them. However, because denotational semantics does not maintain a notion of state (and this has to be passed to the functions each time) presenting semantics in this way can sometimes be different to the way that the definition of the source language has followed with respect to the compilation tool.

Denotational semantics is very extendable and using the basic building blocks considered it is possible to describe complicated languages. An example of this is both in [Tennent1976] and [Reynolds1989]. [Tennent1976] takes the simple semantics explained and complicates it with the notion of stores, control structures and procedure calls. Although these subjects make the semantics more complicated, it is clear what they mean and how they have progressed from the simple concepts initially discussed. [Reynolds1989] is more complicated on the other hand and, although it covers more detail, the semantics explained takes some work to understand. However [Reynolds1989] covers much more detail than [Tennent1976], discussing amongst other things, proofs, arrays, errors and non termination. [Winskel1993] also has a chapter about denotational semantics. Although this chapter is very similar to the other resources, it presents the material with different a emphasis. Unlike the other two resources, the book does not provide extensive example use of denotational semantics and just limits the consideration to simple imperative languages. Whereas this approach helps initially, it does not provide the detailed information which would be required when semantically modelling the source language.

Algebraic Semantics

Algebraic Semantics is a semantics based upon abstract algebras, being primarily geared towards the formalisation of Abstract Data Types the data and language constructs are provided algebraic specifications. As [Slonneger1994] explains, a type is a sort, a sort being composed of a signature and set of equations (or axioms.) A signature of a specification is a pair, ≪*Sorts, Operations*≫, where *sorts* represents the sorts and *operations* the functionality. Using this notion complex system representation can be constructed following sets, their operations and signatures.

However, representing these semantics via sets is conceptually difficult and a much better way has been developed using a modular approach. In this approach, the semantic specification is provided using a structured framework, and allows for complex modular descriptions to be constructed from a number of much simpler primitive modules. These specifications allow for certain properties to be proven and also act as a good starting point for tool development. However, one minor problem is that due to length of each module's definition and description, in order to present a complex system then a large amount of space is required.

The most useful resource found to use within this context was [Slonneger1994]. The chapter of this book firstly introduces the concepts, illustrates these with the mathematical background and then provides numerous examples to help understanding, as well as using this semantics to define a complete language. However, this resource does not really cover proofs of program properties using this semantics which is an important aspect.

2.4.3 Calculi for Languages

There are two calculi considered in this section which form the basic building blocks of language design formalisation.

Lambda Calculus

"Lambda calculus is a formal system designed to investigate function definition, function application, and recursion." [Zhu2005] This calculus is used to define semantic abstractions, used in type systems and forms the basis of functional programming languages. This notation allows one to focus on the definition of the function and also allows functions to be first class values, meaning that, for example, they can be passed freely to other functions and assigned to variables. This calculus binds a variable into the body of the function, for example $\lambda x.e$ binds x into the body of e. Variables which appear in the body and not in the binder are called free variables. In order to simplify a lambda expression, beta reductions are used, as [Pierce1997] states, the beta reduction rule is $(\lambda x.M)\ N \rightarrow [N/x]M$, which more informally states that an expression can be reduced by replacing a redex with the result of replacing, in the above formalisation, x with N. An expression with no redexes is said to be in its normal form. It is quite possible that there will be a number of different ways to apply reduction to a lambda expression however by the Church-Rosser theorem all reductions on an expression that terminate will produce the same result.

In contrast to reduction, there is also the concept of alpha conversion which allows one to rename bound variables as long as the target name is not an existing free variable. As [Pierce1997] formalises, $\lambda x.M = \lambda.([y/x]M)$ if y is not a free variable. The last elementary concept to consider is substitution, where for example *[M/x] N* denotes the substitution of M for x in N. [Pierce1997] like many texts on this material contain substitution rules. As mentioned, lambda calculus is a powerful abstraction mechanism and functions can provide functions as the result of their computation. In lambda calculus the transformation of a function requiring multiple arguments into a function requiring a single argument is called currying. In this case, the arguments are applied to a function returning the resulting function. An example is provided in [Meunier1997] where the function $mult :: Int \rightarrow Int \rightarrow Int$, *mult 2 6* will result in 12. However to represent this, *mult 2* will result in the curried function *(mult 2)* and then this will be applied to *6* in the form *((mult 2)6).*

[Pierce1997] mentions a number of applications of lambda. For instance, both *true* and *false* can be represented by $\lambda t.\lambda f.t$ for *true* and $\lambda t.\lambda f.f$ for *false*. From these simple boolean representations, expressions such as conditionals and logical expression can be constructed. There also exists a concept known as Church Numerals which allow one to encode integers, for instance $C0 = \lambda z.\lambda s.z$ and $C1 = \lambda z.\lambda s.s\ z$ From church numerals a number of functions can be defined such as arithmetic. Lastly, recursion in lambda calculus can occur however one can not name lambda expression due to their anonymity. In order to get past this, we can rewrite the function so that it takes itself as an argument.

Until this point in the literature review only untyped calculus has been considered. This can be extended to typed lambda calculus where, for instance $\lambda x{:}Int.e$ bound variables are given types. This typed lambda calculus allows us, amongst other things, to consider type systems within the context of the calculus.

There are many texts dealing with lambda calculus. [Pierce1997] is considered an excellent resource, and it demonstrates in detail the beta reductions for functions such as addition for church numerals. [Reynolds1989] also provides a detailed chapter and, in addition to the calculi a link with denotational semantics to interpret applications as the application of functions to arguments is provided. Both these texts provide a useful resource, although it is believed that [Pierce1997] gives more detail to the basic concepts and so helps more as an introductory text.

Pi Calculus

Pi calculus is an extension of lambda calculus. Pi calculus is a process calculi which describes concurrent behaviour. This was first described in [Milner1993] by Milner in 1993. In this calculus, each expression represents a process which runs in parallel with other processes. Channels exists which allow for processes to exchange messages which is the only thing observable about a process's behaviour. A process can read a variable, write a variable, create a fresh (private) channel, compose itself from two sub-processes, replicate itself and also become inert (do nothing.)

Using the simple concepts defined above, complicated parallel systems can be defined. The lambda calculus expression introduced above can be expressed in pi calculus. As [Pierce1997] notes, *true (b) = !b(t,f).*$\bar{t}$ $\prec\succ$ and *false (b) = !b(t,f).*$\bar{f}$ $\prec\succ$ This calculus is useful for formalising concurrent systems and, in the context of the parallel language, can be used to describe parallelism to an extent. [Pierce1997] considers this calculus, although not in particularly great detail and more application examples would be useful. [Milner1993] provides much more information and background to the calculi with some specific application examples. It is believed that, if this calculus is to be used then [Milner1993] will provide a good basis to work from.

2.4.4 Types

The idea of associating types with program variables and expressions is an important one. Not only does this aid the compilation of a language (as it is explicit what each atom represents), it also allows more easily for program errors to be detected. For example, if a variable has been typed to be an integer and the programmer attempts to assign a character to it, then this can be an indication of an error which can be easily picked up by the compiler. [Cardelli1997] is a useful literature resource introducing the field, the context within which it sits and also some practical examples of type systems.

It should be noted that a type system does not necessarily require that a programmer explicitly gives type information. For example, via type inference, the expression *x:=33* will provide enough information to the compiler that x should be an integer. As [Cardelli1997] notes, a language is explicitly typed if types are part of the syntax and implicitly typed otherwise. As mentioned previously, the idea behind a type system is to prevent program errors. A trapped error is one which, when detected forces execution to cease, whereas an untrapped one goes unnoticed. A safe language does not have the possibility of untrapped errors within it, which is an important property of many languages (especially a parallel language, where time really is money!) Untyped languages can enforce safety by extensive runtime checking, however this has a significant performance hit and type checkers of typed languages can often produce more efficient results, with much of the analysis being carried out during compile time (known as static analysis.) A program passing the type checker is known as well typed.

When designing a type system it is important to consider exactly what this system will provide. [Cardelli1997] suggests three basic properties that all systems should provide. Firstly, type systems should be decidedly verifiable meaning that there should exist an algorithm to ensure that a program adheres to its type rules. Secondly a type system should be transparent, where a programmer should easily be able to predict if the typechecker will fail and if so easily discover why. Thirdly, a type system should be enforceable where type declarations should be statically checked as much as possible and then dynamically checked if required.

A good place to start when considering a specific type system for a language is to formalise the system and prove facets such as the system is sound. As [Cardelli1997] mentions, to formalise a type system we first of all describe its syntax. Next the scope rules of the language need to be defined which associate identifiers to their binding locations. After these two initial stages, the type rules of the language can be defined. Notationally, *V:T* means that term V has a type T. Associated with this is also an environment, which stores the types of free variables in the program segment. The type rules of a language follow an inference based proof and define the truth of certain judgements based upon smaller component judgements. For example, if *E:Int* and *P:Int*, then *E+P:Int*. It is the collection of these rules which form the type system. This type system can then be used as a basis for a number of other typing functions, one example being that of a type checker. From type rules, a number of derivations can be deduced. These derivations mean that, from a few relatively simple rules complication compositions can be modelled.

As mentioned previously, there are numerous implicitly typed languages available, where the programmer does not specify the type. In this case, in order to be a typed language type inference is required. Type inference results in a type, after applying a number of inference rules to the program being analysed. The inference rules are based from the typing rules. For example, if there is a type rule saying that *1:Int*, then if an expression *a:=1* were encountered, via inference *a:Int*. Type inference can be very complicated, especially if there is no type information provided by the programmer in the source language what so ever. There are a number of languages which require some information about type, but also allow for much of the information to be implicit. This information is enough to allow the inference system to correctly deduce types, yet gives the programmer freedom from having to explicitly type everything.

[Reynolds1989] provides an extensive number of example type rules for a particular language and then use these to prove a particular type judgement. Mentioned is the fact that whether or not a language is explicitly or implicitly typed often depends on the complexity of the language itself. There is also a notion of subtypes, which allows for a type to be based upon a more simple one, thus allowing for implicit type coercion. The subtype notation is $\leq$. For example, $A \leq B$ means that A is a subtype of B, or a more concrete example $Integer \leq Real$. Both [Reynolds1989] and [Cardelli1997] explain in detail how to intergrate subtypes into the type rules.

[Cardelli1985] is another literature resource which has been considered when surveying types. This paper re enforces many of the notions raised by [Cardelli1997] although there is a large portion considering polymorphic types. "Polymorphic types may be defined as types whose operations are applicable to operands of more than one type." [Cardelli1985] This is contrasted to monomorphic types, which limit to at most one type. There are two general forms of polymorphism introduced by this paper, universal and ad-hoc. Within both these categories there are two subcategories. Figure 2.4 provides an overview to these polymorphic classifications. In

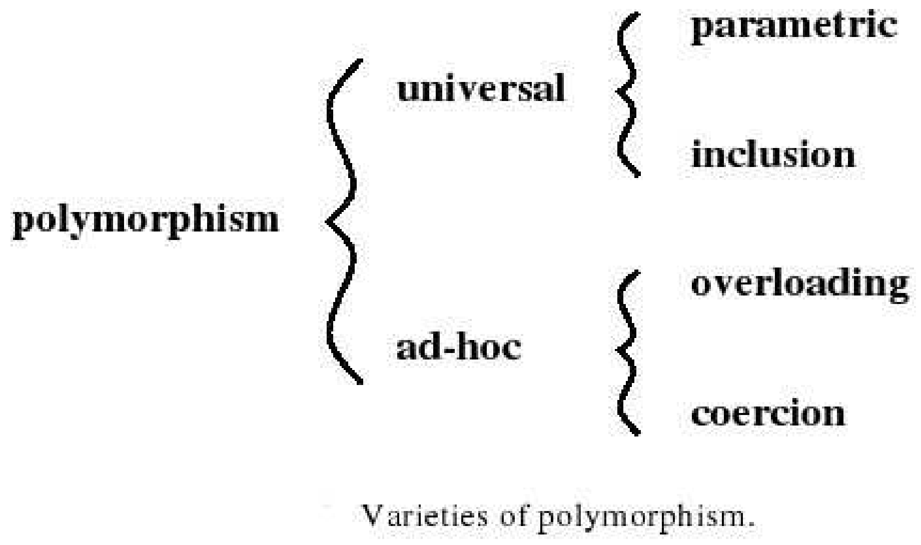

Figure 2.4: Polymorphic types taken from [Cardelli1985]

parametric polymorphism, the purest form, the explicit or implicit type determines the type of argument for all functions. The other form in this category is inclusion polymorphism, where an object belongs to many different classes (which is an example of subtyping.) Within ad-hoc polymorphism, overloading is where a different type denotes a different function and the context decides the function, which can be seen as syntactic sugar. The other form of polymorphism within this category is coercion, which converts the type of an argument into the type expected. Another interesting section is where the author notes that a type is simply a set of elements or values. The universe, U, is the set of everything, with ideals being a subset of U which obey properties associated with the type system. Therefore a type system is simply a collection of these ideals, and the notion of a value having a type is simply membership of that type set.

Within [Cardelli1985] a type expression sublanguage is used to define the set of types. Using this sublanguage, a toy example language known as "fun", a typed λ language to model polymorphism, is used as an example basis. Within this example language, it is noted that lambda calculus is not sufficient to model polymorphism, and universal quantification is required, an example provided by [Cardelli1985] is $\forall[a], fun(x : a) = x$ for the identity function. Existential quantification is used for data abstraction, for example $\exists a.\exists b.a\ x\ b$ denotes the set of all ordered pairs.

[Cardelli1997] is a useful, in depth review of typing. This resource is considered very informative and the examples provided are helpful to illustrate the concepts being explained. However, these examples are not considered within the whole evolution of a language, as [Reynolds1989] does and so it is difficult to see from [Cardelli1997] how this typing formalisation fits in with other theoretical concepts. However, on the other hand [Cardelli1997] does not get lost in the detail relevant to other theoretical concepts. [Cardelli1985] is another informative resource which delves into more detail as to the considerations behind a typing system and provides many useful examples of using lambda calculus to define the system. However, [Cardelli1985] considers in great detail Object Oriented notions and, due to the form of source language, these are not relevant to this project although are interesting in general. When formalising the type system of the source language then it is believed that [Cardelli1997] will be most useful as reference material, although the way [Cardelli1985] describes types as sets and values of a certain type being set membership is helpful to represent these notions.

Plugable Types

Plugable Types [Bracha2004] allow the language designer to separate the language from the type system - the (dynamic) type system is kept separate from the language itself and has no affect on the run time semantics. Using this approach the designer can use zero, one or many type systems in combination with their language. As [Bracha2004] argues, mandatory type systems (as in the majority of mainstream languages) can be considered harmful, as they limit the expressiveness of a language and can make systems brittle when programmers rely on them to be sound and complete. It can be seen, if the type system is taken out of the language itself and provided as a plug-in, how this can result in a much simpler “core” language. Although being completely dynamic will limit the amount of analysis which can be performed during compilation, there is certainly merit in this proposal even if modifications are required to suit it to the domain of parallel computing.

2.4.5 Program Analysis

The process of program analysis is used to detect errors in a program. The aim of analysis is to prevent forbidden errors, which are defined as untrapped errors and a number of trapped errors. A program is well behaved if it does not cause any forbidden errors to occur. A language for which all its legal programs are well behaved is known as strongly checked. In order to achieve the detection of errors, there are two techniques possible, dynamic and static analysis.

Dynamic Analysis

Dynamic analysis is the detection of forbidden errors during program execution. An advantage to this method is that as the program is running, all the program attributes are known and a correct analysis check can be made from these. However, a major downside of dynamic analysis is that it is computationally expensive, which is key within the context of high performance computing. An example of this cost is easily seen in languages such as Java. Java performs numerous runtime checking tasks, such as array bounds checking, conversely many C implementations do not. The result is that Java has a performance hit, even when combined with advanced compiler technology, which C does not incur. However forbidden program errors in Java are less likely and far less significant than in the approach adopted by many C implementations.

It should be concluded that dynamic analysis is a useful tool if used only when there is no possibility to perform the analysis during compile time. Within the context of high performance computing, a forbidden program error can lead to disastrous results such as a large waste of resource or the incorrect result and as such are not to be allowed.

Static Analysis

Static analysis is the detection of forbidden errors during program compilation. The major advantage of this is that, because it only needs to be done once (during program compilation), then there will be no reduction in efficiency. However, during compilation only a subset of the program state is known and so absolutely correct analysis is often not possible or would be too resource and time consuming. This has meant two things, firstly a static analyser must keep track of what it has not been able to check to mark it for dynamic analysis and secondly a number of analysis techniques have been developed to help out with this problem of incomplete information.

One important technique is Abstract Interpretation. "Abstract interpretation theory formalizes the conservative approximation of the semantics of hardware or software computer systems." [Cousot2002] This theory was first introduced in [Cousot1975] which states that the reason for this interpretation is that fact that during compile time type verifications are usually incomplete. A program is evaluated using abstract values, instead of concrete ones. [Cousot1975] introduces an abstraction function which maps concrete values to abstract ones. The converse, a concretisation function is also provided which maps abstract values to concrete ones. The paper considers a number of interesting abstractions. In order to use abstract interpretation within the context of loops a problem of termination arises, which is solved via widening. The symbol for this widening is $\bar{\nabla}$. The different forms of abstract values are explained via examples, which is very useful to promote understanding.

[Cousot1975] also considers the idea of an abstract context. "An abstract context is a set of pairs, *(i,v)* which express that the identifier i has the abstract value v at some program point." [Cousot1975] This not only represents a history of abstract values to being stored, but it also allows one to work with these concepts theoretically. ϱ is defined as the set of abstract contexts. These concepts introduced form the basis of abstract interpretation. A common example of abstract interpretation is within the context of mathematical expressions. For instance, a concrete value expression might be *5 - 2*, instead of caring about the actual operands the analyser might only care about the sign of the operand. Therefore, applying an abstraction function would result in *+ - +*. If the analyser wished to avoid negative resulting numbers, then it would be sufficient to detect errors if, for instance the abstract expression was *- - +*. However, if the expression was *+ - +* then an error would not be detected without further information. This example illustrates the tradeoff involved with abstract interpretation, for efficiency of analysis detail is lost.

Due to this loss of detail, it is quite possible that often the analyser will result in a state where it is undecidable whether or not an error will occur. As [Cousot2005] considers, there is a concept known as false alarms, where, due to the approximations made during abstraction, the analyser believes there to be a problem whereas in reality the concrete values do not cause an error. In this case it is important to minimise the number of false alarms and, if one does occur, it should not result in the failure of compilation - at worse some dynamic analysis code should be added.

There have been numerous papers and published details about abstract interpretation. It is considered that [Cousot1975] which first considers this concept provides a useful introduction and illustrates the power that this technique affords. The examples in this paper help describe the concepts and their application practically. A more theoretical paper is [Cousot1977] which applies this interpretation to a wider variety of issues such as correctness and termination. The paper [Cousot1977] is useful as it takes the whole approach further. [Cousot2005] is an introduction to abstract interpretation and is easy to understand, with illustrative examples. Although this third resource [Cousot2005] does simplify the concepts a great deal, it is believed that simplification is carried too far in some respects and some important issues are lost. It can therefore be concluded that each of the three literature resources considered here have a place and when combined they provide a good knowledge foundation to the interpretation.

Another powerful technique used, which is mentioned in [Cousot1975], is range analysis. In range analysis instead of a specific value being stored during analysis, which is often not achievable, only the range of possible values are stored. Quite often this range is represented by the smallest possible value and the largest possible value. Range analysis for integers was first considered in the paper [Wagner2000] which attempted to use the analysis to detect buffer overruns in C. Range analysis is an example of abstract interpretation, where, instead of the concrete value being stored, an abstract value (the possible range) is instead recorded. Range analysis is used in the LOGS translator [Zhou2005] to perform static analysis. An example of the application of range analysis is in detecting array bounds errors, where, if the range value of the index variable is within the size of the array then an error will not occur. Likewise, if only one range value (i.e. the highest possible value) is outside the bounds then dynamic analysis only needs to be carried out for that case thus saving on dynamic analysis.

When considering the literature available for range analysis, [Zhou2005] is a very useful example of how range analysis is conducted and the source code described in this paper offers a practical description. [Cousot1975] is a useful overview, although it uses range analysis to describe other concepts, and the examples provide illustrations as to the power of this method.

Static analysis also allows for certain optimisations to be made during compilation time. Depending on the values of variables, quite often items such as control and iterative structures can be simplified, as well as the value of an expression computed. It is believed that the rich parallel information available in the source language shall help with static analysis and thus optimisation, producing target code which is very efficient.

2.4.6 Summary

From the concepts considered in this section it is clear that the theory of programming language design is a complicated, extensive field. Due to the shear volume of the field, it is important not to get lost in this theory and ensure that the concepts do directly relate to this project. Many of the concepts discussed will be used as tools to present, define and act as a basis for tool development. In the context of this field, it is not expected that any innovation will occur, as the concepts will be used in order to innovate in the field of parallel computing.

2.5 Popular Parallel Codes

In order to properly identify what must be supported by a parallel language it is important to consider existing parallel applications. There are a great number of parallel code examples to solve “toy” problems. The book [Pacheco1996], provides numerous code examples which are very useful to look through in order to understand exactly how programmers are currently writing parallel code and what is required from a language.

In this section, three applications and one benchmark will be considered which highlight the advantages of parallel computing. The codes are not only important for discussion within their own right, but they will also be targets of porting into the source language.

2.5.1 Gadget-2

The first package to consider is Gadget-2. "GADGET is a freely available code for cosmological N-body/SPH simulations on massively parallel computers with distributed memory. GADGET uses an explicit communication model that is implemented with the standardised MPI communication interface. The code can be run on essentially all supercomputer systems presently in use, including clusters of workstations or individual PCs." [Springel2005] This scientific application is around fifty thousand lines of C code using the MPI library. The fact that it has taken over ten years to create this many lines just shows how difficult parallel programming is at the moment.

Gadget-2 is detailed in [Springel2006]. This literature resource follows the evolution of Gadget-2, from initial design decisions to the current incarnation. There are evaluations and design decisions included about different algorithms and the concepts behind them. These slides are very technical with respect to the physics in places, however, do help to understand the actual code, for instance covered is the Peano-Hilbert curve, which is used for domain decomposition. There are many different types of simulations which are all controlled with a parameter file.

An especially interesting section of [Springel2006] details a simulation which ran on 512 CPUs, required one Terabyte of RAM and took 350 hours of processing time (28 days). If this simulation had been carried out on a sequential machine, the results would have taken over 38 years to produce. These figures not only show the great advantage to parallelism, but also demonstrate that Gadget-2 and MPI are serious resources, designed for serious work. This resource concludes with the evaluation of programming in C, C++ or Fortran, all of which MPI supports. It is concluded from this that C is most suitable for development.

As part of the Gadget-2 presentation, the scalability and limitations of Gadget-2 are mentioned. This is a very relevant section due to some of the conclusions mentioned here affect all parallel, or MPI based programs. In all, Gadget-2 performs well in terms of scalability for large problems, but not so well for small ones. The author details that parallelism is hampered by parallel overhead, communication costs, work imbalance wait times and the existence of serial code. It is suggested that weak scaling (where as the number of CPUs is increased, so is the problem size) is preferable to strong scaling (where the problem size remains fixed regardless.)

The paper [Springel2006] is considered very informative. However, this ninety four page document does cover a lot of different aspects relating to Gadget-2, with each individual topic only reviewed in a high-level manner. Also, as the resource was designed for presentation, most likely there is some explanation and further detail associated with many of the points. In order to get an understanding of how Gadget-2 works (for porting), then it will be important to study the actual code, which [Springel2006] does not provide information for.

As mentioned, apart from Gadget-2 providing useful solutions to some parallel problems, it would also act as a good showcase application for the parallel language being created. If the parallel segments of Gadget-2 can be rewritten, more succinctly, in the language then it will help maintenance and development of the code by scientists. [Baker2006] details a port of Gadget-2 to Java. The aim of this project has been to illustrate that Java is able to support high performance computing. The paper initially reproduces much of the information in [Springel2006], albeit in less detail. In order to successfully port Gadget-2, seventeen thousand lines of C code were reproduced in Java, with MPI calls being replaced by MPJ Express [Carpenter2000], which is a Java messaging system based on MPI. The project group concluded that the Java version was slower by a factor of 2-3, although this was in part due to the immaturity of support libraries such as MPJ Express. [Baker2006] is very interesting, although not directly related to this project it does show that the porting of Gadget-2 is possible and gives some indication as to the work required.

Unfortunately, the work is not yet complete and thus the authors have not released their source code.

A major issue of Gadget-2 is that each substantial simulation requires parts of the package to be recoded. Although written to make this as simple as possible, departments still need expert parallel programmers well versed in C to carry out this work.

2.5.2 Fast Fourier Transformation

"The Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse. FFTs are of great importance to a wide variety of applications, from digital signal processing to solving partial differential equations to algorithms for quickly multiplying large integers." [Lavoie1996] Apart from being a very commonly used scientific and engineering algorithm, it can also act as a benchmark program. Most parallel languages especially BSPlib, Cilk and MPI in the scope of this thesis, have been used to implement FFT and measure its efficiency. To this end, FFT is an important algorithm and as a "real" scientific problem it is a good starting place to indicate what the language needs to support. Due to the frequency upon which FFT calculations occur, there has been much investigation in this area. The Fastest Fourier Transformation in the West library (FFTW) is "a comprehensive collection of fast C routines for computing the discrete Fourier transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size." [Frigo1999]. A useful resource detailing FFTW is [Frigo1998-2]. This paper details the ideas behind FFTW, a brief overview, performance analysis and then conclusions. FFTW comprises of a number of different steps. In order to accomplish computation, the transformation is computed via a number of different small sections of code. These small sections, termed codelets, are highly optimised and compose together to form a complete solution. The codelets have been generated via a specially designed compiler and each one is designed for a specific purpose. In order to determine which codelets to use, the problem, which is described in a specific way by the programmer is passed to the planner. The planner then produces a plan which uses a dynamic programming algorithm to determine the fastest combination of codelets. At this point, the plan is then fed into the executor, which actually calls the required codelets and performs the computation, after this stage the result of the FFT is provided. As concluded in [Frigo1998-2] computer architectures are nowadays so complex that manually optimising software is almost impossible, therefore the FFTW approach produces a high performance implementation available to almost all FFT problems. The performance results section is somewhat sparse, with a diagram detailing evaluation against eleven other FFT algorithms but without much description or detailed analysis. However, what the resource does identify is that only the planner and executor need to be considered as the codelets have been precompiled into C. [Frigo1999] provides detailed information as to the codelet compiler, whereas this is interesting to read, it is not considered relevant to the project, but does illustrate that the source language will require some form of native interface, to call existing C code.

FFTW, in addition to the sequential implementation, has two parallel implementions using MPI and Cilk. The MPI version is the mature parallel code and implements sections of a parallel planner and executor, with other sections using the sequential C code. The Cilk version is much simpler and in an experimental stage. It is considered that the Cilk code is more suited to use to gain an understanding of how FFTW works due to the fact that it is much simpler than the MPI version.

2.5.3 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) are a collection of benchmarks written by NASA's Advanced Supercomputing division (NAS) in order to evaluate parallel supercomputers. The latest version of these, NPB 3, contains 11 distinct benchmarks for which specifications and implementations exist in a variety of languages.

Amongst these benchmarks, the Integer Sort (IS) is the most interesting one in terms of this project. A relatively simple benchmark, both mathematically and programatically, this code will sort integers using a parallel version of bucket sort. The resource [Baily1994] details the specification of this example, formalising the sorting method, initial number generator (generating the same input data for each run so they are directly comparable) and the verification methods to ensure correct sorting has been completed. The specification lists five classes of benchmark data, class S with sixty five thousand numbers, class W with approximately one million numbers, class A with approximately eight million numbers, class B with approximately thirty three million numbers and lastly class C with approximately one hundred and thirty four million numbers. It is the intention that the specific class of experiment can be tailored to the parallel machine, or that multiple classes can be run.

Along with the specification produced by NAS, there is also a freely available, official NASA implementation written in C using MPI. It is this code which NASA suggests using for comparing parallel supercomputers. For this project, it will be possible to use the code to act as a control and benchmark C-MPI, which will be directly comparable with versions written in other languages as long as they run on the same parallel machine. NASA also maintains a database of benchmark results for different machines. Using this, by taking the timings produced by NASA's existing C code it will be possible to get a general idea of how other languages perform by comparing the timings.

2.5.4 Mandelbrot Set

The Mandelbrot Set is a set of points in the complex plane, the boundary of which forms a fractal. Computing the mandelbrot set is embarrassingly parallel, each point can be calculated irrespective of any other point and as such it makes for a very simple, elegant programming example. Many existing parallel languages have an implementation of this problem in order to illustrate basic language concepts, with all computations collected on a single processor and an image such as that in figure 2.5 is produced.

2.5.5 Summary

There are a wide variety of existing parallel codes in daily use. Many of these are aimed at the scientific community to perform very specific jobs. There is often not the expertise in other fields to take advantage of parallel computing. A

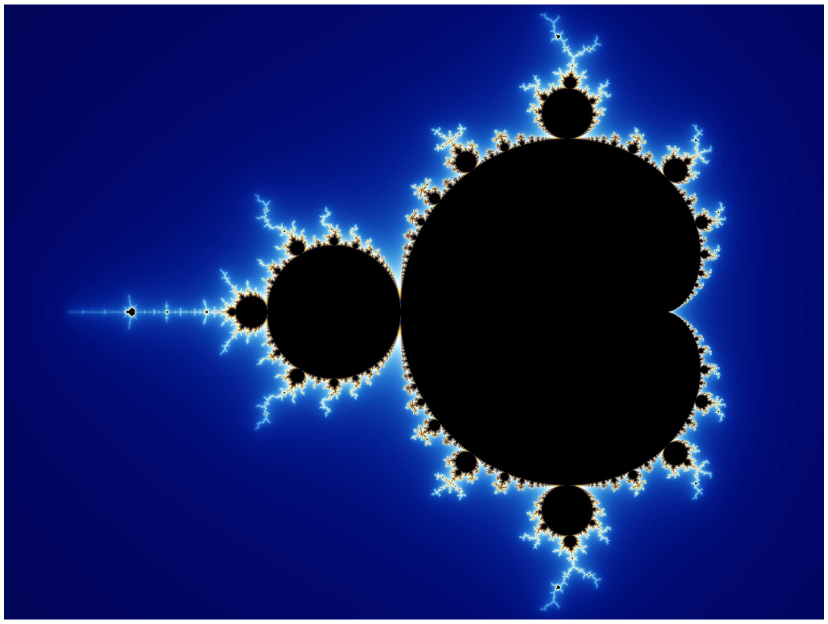

Figure 2.5: Example fractal produced by Mandelbrot Set

parallel program which is 50,000 lines is generally considered to be very large by many programmers whereas that sort of size is normal in sequential codes. Looking at these efficient programs it is often evident that the programmer has done some low level optimisations, for example using pointers, which may achieve some performance increase in the short term but make for a much more difficult to maintain application.

2.6 Conclusions

From reviewing the literature, there is a clear indication that current parallel programming languages and models are not sufficient for modern uses and applications. With the advent of multi core, and soon many core, processors, parallel programming will continue to move from being in the domain of the few experts to that of general computing and programming. This shift has started to force the industry to consider these issues, which until now have been considered too difficult to address. The language design principals and theories which have been considered give a clear message that languages which seem syntactically distinct can often be viewed and categorised into a few models.

Programmers wish to write code which is simple and efficient; in parallel programming these aims are, at the moment contradictory. This is a huge problem, acting as a barrier to many who wish to take advantage of parallelism. By considering these principals and ideas it should be possible to identify ways in which these can be built upon to solve the problem of parallel programming.

Chapter 3

Language Definition

3.1 Introduction

In order to solve the problem identified in Chapter 2 the concept of types have been investigated. The innovation made has been a novel approach to how types are used and their interaction within programming languages. For purposes of illustrating and evaluating the proposed approach a language, Mesham, has been created and is used as vehicle to this end.

In this chapter the novel concept of types will firstly be explained and then, by providing a high level definition of Mesham, the reader will see how these can be used practically to solve the issues already identified. As for defining Mesham, this chapter will detail the type library and core language itself. Further language specification, specifically the preprocessor and function library, can be found in Appendix A.

3.1.1 Language Definition

In order to explain many of the syntactic aspects of Mesham, meta characters will be used. These are detailed in table 3.1 and will be used throughout this chapter. Each language construct and type will be explained in three parts, firstly the syntax, then the semantics and lastly example(s) of use. Where program keywords, variables or types are used within the thesis text these shall be *emphasised*. In designing the language and type based paradigm an important consideration has been the programability, by using examples the reader will be able to see how convenient it is to write code in this way.

Characters	Description
{}	Optional
{}*	Zero or more
{}+	One or more
name	A variable name
...	Continuation

Table 3.1: Meta Characters used in Chapter 3

3.2 Types

3.2.1 Concept

The concept of a type will be familiar to many programmers. A large subset of languages follow the syntax *Type Variablename*, such as *int a* or *float b*, to allow the programmer to declare a variable. Such a statement affects both the compiler and runtime semantics - the compiler can perform analysis and optimisation (such as type checking) and at runtime the variable has a specific size and format. Considering these sorts of languages, it can be thought of that the programmer provides information, to the compiler, via the type. However, there is only so much that one single type can reveal, and so languages often include numerous keywords in order to allow for the programmer to specify additional information. Taking C as an example, in order to declare a variable *m* to be a character in read only memory the programmer writes *const char m*, where *char* is a type and *const* an inbuilt language keyword. In order to extend the language, and allow for extra variable attributes (such as where a variable is located in the parallel programming context) then new keywords and statements would need to be introduced, which bloats the language.

The approach adopted by Mesham is to allow the programmer to encode all variable information, via the type system, by combining different types together to form a supertype. In the language, *const char m* becomes *var m: Char :: const[ ]*, where *var m* declares the variable, the operator *:* specifies the type and the operator *::* combines two types together. In this case, a supertype is formed by combining the type Char with the type const. It should be noted that some type coercions, such as *Int :: Char* are meaningless and so rules exist within each type to govern which combinations are allowed.

Type precedence is from right to left - in the example *Char :: const[ ]*, it can be thought of that the read only attributes of const override the default read/write attributes of Char. For instance the supertype (type chain) created by *A::B::C::D::E* is shown in figure 3.1, where type *E* is at the head of the type chain.

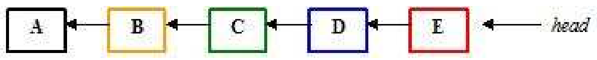

Figure 3.1: Type Combination Illustration

Using this approach many different attributes can be associated with a variable, the fact that types are loosely coupled means that the language designers can easily add attributes (types), and by only changing the type of a variable the semantics can change considerably. Another advantage is that the rich information provided by the programmer allows many optimisations to be performed during compilation that using a lower level language might not be obvious to the compiler.

On a more technical note, the type system implements a number of services. These are called by the core of the compiler and if the specific type does not honour that service, then the call is passed onto the next in the type chain - until all are exhausted. For instance, using the types *A::B::C::D::E*, if service *Q1* was called, then type *E* would be asked first, if it did not honour the service, *Q1* would be passed to type *D* - if that type did not honour it then it would be passed to type *C* and so forth.

3.2.2 Language Support

In order to support this innovative use of types, there are a number of keywords and operators built into the core language of Mesham. These allow the programmer the flexibility to use, combine and refer to types as needed.

A Type

A type can follow a number of different syntactic forms. The abstract syntax of a type is detailed in listing 3.1. Where *elementtype* is defined later in this chapter, *varname* represents a variable name and *type :: type* represents type combination to coerce into a new supertype.

```
type =     elementtype
         | compoundtype
         | type :: type
         | varname
```

Listing 3.1: Abstract syntax of type

Compound types are dealt with later in this chapter, to give the reader a feeling at this point they are comprised of a number of different categories.which are detailed in listing 3.2

```
compoundtype =     attribute
                 | allocation
                 | collection
                 | primitive communication
                 | communication mode
                 | partition
                 | distribution
                 | composition
                 | extended types
```

Listing 3.2: Compound type categories

Declarations

Syntax

var name{:type};

Where type, as explained, is an *elementtype*, a *compoundtype*, variable name or *type :: type*. The operator *:* sets the type and *::* is type combination (coercion).

Semantics

This will declare a variable to be a specific type. Type combination is subject to a number of semantic rules. If no type information is given, then the type will be found via inference where possible.

Examples

```
var i:Int :: allocated[multiple[]];
```

Here the variable *i* is declared to be integer, allocated to all processes. There are three types included in this declaration, the element type *Int* and the compound types *allocated* and *multiple*. The type *multiple* is provided as an argument to the allocation type *allocated*, which is then combined with the *Int* type.

```
var m:String;
```

In this example, variable *m* is declared to be of type *String*. For programmer convenience, by default, the language will automatically assume to combine this with *allocated[multiple]* if such allocation type is missing.

Statements

Syntax

name:type;

Semantics

Will modify the type of an already declared variable via the *:* operator. Note, allocation information may not be changed.

Examples

```
var i:Int :: allocated[multiple[]];
i:=23;
i:i :: const[];
```

Here the variable *i* is declared to be integer, allocated to all processes and its value is set to 23. Later on in the code the type is modified to set it also to be constant (so from this point on the programmer may not change the variable's value.) In this third line *i:i :: const[];* sets the type of *i* to be that of *i* combined with the *const* type.

Important Rule - Changing the type will not have any runtime code generation in itself, although the modified semantics will affect how the variable behaves from that point on.

Expressions

Syntax

name{::type}

Semantics

When used as an expression, a variable's type can be coerced with additional types just for that expression.

Example

```
var i:Int :: allocated[multiple[]];
(i :: channel[1,2]):=82;
i:=12;
```

This code will declare *i* to be integer, allocated on all processes. On line 2 *i :: channel[1,2]* will combine the channel type (primitive communication) just for that assignment and then on line 3 the assignment happens as a normal integer. This is because on line 2 we have not set the type of *i*, just modified it for that assignment.

currentype

Syntax

currentype varname

Semantics

Will return the current type of the variable.

Example

```
var i: Int;
var q:currentype i;
```

Will declare *q* to be an integer of the same type as *i*.

declaredtype

Syntax

declaredtype varname

Semantics

Will return the declared type of the variable.

Example

```
var i:Int;
i:i::const[];
i:declaredtype i;
```

Here in line 2 the programmer adds the constant type to the variable, however the type is then reverted back to the declared type (integer) in line 3.

Type Variables

Syntax

typevar name{::=type};

name::=type;

Note how *::=* is used rather than *:=*

typevar is the type equivalent of a new program variable declared using the keyword *var*

Semantics

Type variables allow the programmer to assign types and type combinations to variables for use as normal program variables. These exist only in compilation and are not present in the runtime semantics.

Examples

```
1 typevar m::=Int :: allocated[multiple[]];
2 var f:m;
3 typevar q::=declaredtype f;
4 q::=m;
```

In the above code example, the type variable m has the type value *Int :: allocated[multiple[]]* assigned to it. On line 2, the new (program) variable is created using this new type variable. In line 3, the type variable q is declared and has the value of the declared type of program variable *f*. Lastly in line 4, type variable q changes its value to become that of type variable *m*. Although type variables can be thought of as the programmer creating new types, they can also be used like program variables in cases such as equality tests and assignment.

3.3 Type Library

In the Mesham approach there is a very clear distinction between the core language and the library of types. By moving all the complexity of the language into the types, the result is a simple, elegant language.

3.3.1 Allocation

There are a number of types which the programmer can use to specify how and where a variable is located within the memory of different processes. Just this task alone adds many keywords to existing parallel languages which, using the proposed type approach, is avoided.

Allocated

Syntax

allocated[{type}];

Semantics

This type sets the memory allocation of a variable, which may not be modified once set.

Example

```
var i: Int :: allocated[];
```

In this example the variable *i* is an integer. Although the *allocated* type is provided, no addition information is given and as such Mesham allocates it to each processor.

Multiple

Syntax

multiple[{type}];

Semantics

Included in allocated will (with no arguments) set the specific variable to have memory allocated to all processes within current scope.

Example

```
var i: Int :: allocated[multiple[]];
```

In this example the variable *i* is an integer, allocated to all processes.

Commgroup

Syntax

commgroup[process list]

Semantics

Specified within the multiple type, will limit memory allocation (and variable communication) to the processes within the list given in this type's arguments.

Example

```
var i:Int :: allocated[multiple[commgroup[1,2]]];
```

In this example there are a number of processes, but only 1 and 2 have variable *i* allocated to them.

Single

Syntax

single[{type}]

single[on[{process}]]

Semantics

Will allocate a variable to a specific process. Most commonly combined with the *on* type which specifies the process to allocated to, but not required if this can be inferred. Additionally the programmer will place a distribution type within *single* if dealing with distributed arrays.

Example

```
var i:Int :: allocated[single[on[1]]];
```

In this example variable i is declared as an integer and allocated on process 1.

3.3.2 Element Types

An element type is a primitive type given to a variable. Mesham supports a number of element types, these are detailed in table 3.2.

Type	Description
Int	Integer
Float	Floating point number
Double	Double precision number
Bool	True or false value
Char	A character
String	A string of characters
File	A file handle
Long	A long (64 bit) integer

Table 3.2: Mesham's element types

Communication in Assignment

When a variable is assigned to another, depending on where each variable is allocated to, there may be communication required to achieve this assignment. Table 3.3 details the communication rules in the assignment *assigned variable := assigning variable*. If the communication is issued from MPMD programming style then this will be one sided. The default communication listed here is guaranteed to be safe, which may result in a small performance hit.

Assigned Variable	Assigning Variable	Semantics
multiple[ ]	multiple[ ]	local assignment
single[on[i]]	multiple[ ]	local assignment on process i only
multiple[ ]	single[on[i]]	MPI broadcast from process i
single[on[i]]	single[on[i]]	local assignment on process i
single[on[i]]	single[on[j]]	sent from j and received by i ($i \neq j$)

Table 3.3: Element type communication in assignment

Example

```
var a:Int;
var b:Int :: allocated[single[on[2]]];
var p;
par p from 0 to 3
{
if (p==2) b:=p;
a:=b;
};
```

This code will result in a onesided broadcast (due to being written MPMD style in *par* loop) where process 2 will broadcast its value of b to all other processes who will write it into a. As already noted, in absence of allocation information the default of allocating to all processes is used. In this example the variable a can be assumed to additionally have the type *allocated[multiple]*.

3.3.3 Attributes

Const

Syntax

const[ ]

Semantics

Enforces the read only property of a variable.

Example

```
var a:Int;
a:=34;
a:(a :: const[]);
a:=33;
```

The code in the above example will produce an error. Whilst the first assignment (*a:=34*) is legal, on the subsequent line the programmer has modified the type of *a* to be that of *a* combined with the type *const*. The second assignment is attempting the modify a now read only variable and will fail.

Tempmem

Syntax

tempmem[ ]

Semantics

Used to inform the compiler that the programmer is happy that a call (usually communication) will use temporary memory. Some calls can not function without this and will give an error, others will work more efficiently with temporary memory but can operate without at a performance cost. This type is provided because often memory is at a premium, with applications running towards at their limit. It is therefore useful for the programmer to indicate whether or not using extra, temporary, memory is allowed.

Share

Syntax

share[name]

Semantics

This type allows the programmer to have two variables sharing the same memory (the variable that the share type is applied to uses the memory of that specified as arguments to the type.) This is very useful in HPC applications as often processes are running at the limit of their resources. The type will share memory with that of the variable *name* in the above syntax. In order to keep this type safe, the sharee must be smaller than or of equal size to the memory chunk, this is error checked.

Example

```
var a:Int::allocated[multiple[]];
var c:Int::allocated[multiple[] :: share[a]];
var e:array[Int,10]::allocated[single[on[1]]];
var u:array[Char,12]::allocated[single[on[1]] :: share[e]];
```

In the example above, the variables a and c will share the same memory. The variables *e* and *u* will also share the same memory. There is some potential concern that this might result in an error - as the size of *u* array is 12, and size of *e* array is only 10. If the two arrays have different types then this size will be checked dynamically - as an *int* in C is usually 32 bit and a *char* usually only 8 then most likely this sharing of data would work in this case.

Extern

Syntax

extern[{location}]

Semantics

Provided as additional allocation type information, this tells the compiler NOT to allocate memory for the variable as this has been already done externally. The *location* argument is optional and just tells the compiler where the variable is to be found (e.g. a C header file) if required.

Directref

Syntax

directref[ ]

Semantics

This tells the compiler that the programmer might use this variable outside of the language (e.g. Via embedded C code) and not to perform certain optimisations which might not allow for this.

Example

```
var pid:Int :: allocated[multiple[]] :: directref[];
ccode["pid=(int) getpid();","","#include <sys/types.h>","#
    include <unistd.h>"];
print["My Process ID is ",pid,"\n"];
```

The code example above illustrates how the Mesham programmer can easily include native C code in their program, using normal program variables. First the variable *pid* is declared to be an integer, allocated to all processes and that it will be referenced directly by native C. The *ccode* function then allows the programmer to code directly in C and uses the POSIX function *getpid* to obtain the process ID of the current program, which is cast as an integer and stored directly in variable *pid*. The last line, once again Mesham code, will display this process ID.

3.3.4 Collections

Array

Syntax

array[type,d_1,d_2,...,d_n]

Semantics

An array, where *type* is the element type, followed by the dimensions. The programmer can provide any number of dimensions to create an n dimension array. Default is row major allocation (although this can be overridden via types.) In order to access an element of an array, the programmer can either use the traditional *name*[*index*] syntax or, alternatively *name#index* which is preferred by the thesis author.

Communication of Assignment

When an array variable is assigned to another, depending on where each variable is allocated to, there may be communication to achieve this assignment. Table 3.4 details the communication rules for this assignment *assigned variable := assigning variable.* As with the element types, default communication of arrays is safe.

Example

Assigned Variable	Assigning Variable	Semantics
multiple[]	multiple[]	memory copy
single[on[i]]	multiple[]	memory on process i only
multiple[]	single[on[i]]	MPI broadcast from process i
single[on[i]]	single[on[i]]	local memory copy on process i
single[on[i]]	single[on[j]]	sent from j and received by i ($i \neq j$)

Table 3.4: Array type communication in assignment

```
var a:array[String,2] :: allocated[multiple[]];
(a#0):="Hello";
(a#1):="World";
print[(a#0)," ",(a#1),"\n"];
```

This example will declare variable a to be an array of 2 Strings. Then the first location in the array will be set to *"Hello"* and the second location set to *"World"*. Lastly the code will display on stdio both these array string locations followed by newline.

Row and Col Types

Syntax

row[ ]

col[ ]

Semantics

In combination with the array, the programmer can specify whether allocation is row or column major. This allocation information is provided in the allocation type.

Example

```
1 var a:array[Int,10,20] :: allocated[col[] :: multiple[]];
2 ((a#1)#2):=23;
3 (((a :: row[])#1)#2):=23;
```

Where the array is column major allocation, but the programmer has overridden this (just for the assignment) in line 3. If one array of allocation copies to another array of different allocation then transposition will be performed automatically in order to preserve indexes.

<u>**Spaceshape**</u>

Syntax

spaceshape[type,min_1,max_1,min_2,max_2,...,min_d,max_d]

Semantics

This is an abstraction for storing data. Often the HPC programmer is dealing with data in a number of dimensions, this type allows for storing the data at specific points (and retrieve it) in a d dimensional space. The *spaceshape* type is an implementation of a sparse matrix, which are commonly used in the HPC domain.

Example

```
var a:spaceshape[String,1,3,0,5,2,5];
(((a#2)#3.4)#4.23):="hello!";
print[(((a#2)#3.4)#4.23),"\n"];
```

3.3.5 Primitive Communication

Primitive communication types ensure that all, safe, forms of communication supported by MPI can also be represented in Mesham. However, unlike the shared variable approach adopted elsewhere, when using primitive communication the programmer is responsible for ensuring communications complete and match up.

<u>**Channel**</u>

Syntax

channel[a,b]

Where a and b are both distinct processes which the channel will connect.

Semantics

The *channel* type will specify that a variable is a channel from process a (sender) to process b (receiver.) Normally this will result in synchronous communication, although if the *async* type is used then asynchronous communication is selected instead. Note that channel is unidirectional, where process a sends and b receives, NOT the otherway around.

Example

```
var x:Int::allocated[multiple[]];
var p;
par p from 0 to 2
{
(x::channel[0,2]):=193;
var hello:=(x::channel[0,2]);
};
```

In this case, x is a channel between processes 0 and 2. In the par loop process 0 sends the value 193 to process 2. Then the variable *hello* is declared and process 2 will receive this value.

Pipe

pipe[a,b]

Identical to *channel*, except it is bidirectional rather than unidirectional

Onesided

onesided[a,b]

Very similar to *channel*, but will perform onesided communication rather than p2p. This form of communication is less efficient than p2p, but there are no issues such as deadlock to consider.

Reduce

Syntax

reduce[root,operation]

Semantics

All processes in the group will combine their values together at the *root* process and then the operation will be performed on them. Numerous operations are supported, such as *sum*, *min*, *max* and *multiply*.

Example

```
var t:Int::allocated[multiple[]];
var x:Int::allocated[multiple[]];
var p;
par p from 0 to 3
{
x:(x::reduce[1,"max"];
x:=p;
t:=x;
};
```

In this example, x is to be reduced, with the root as process 1 and the operation will be to find the maximum number. In the first assignment *x:=p* all processes will combine their values of p and the maximum will be placed into process 1's x. In the second assignment *t:=x* processes will combine their values of x and the maximum will be placed into process 1's t.

Broadcast

Syntax

broadcast[root]

Semantics

This type will broadcast a variable amongst the processes, with the root (source) being that where the proess ID equals the *root* argument of the type. The variable concerned must either be allocated to all or a group of processes (in the later case communication will be limited to that group.)

Example

```
var a:Int::allocated[multiple[]];
var p;
par p from 0 to 3
{
(a::broadcast[2]):=23;
};
```

In this example process 2 (the root) will broadcast the value 23 amongst the processes, each process receiving this value and placing it into their copy of *a*.

Gather

Syntax

gather[elements,root]

Semantics

Gather a number of elements (equal to *elements*) from each process and send these to the root process.

Example

```
var x:array[Int,12] :: allocated[single[on[2]]];
var r:array[Int,3] :: allocated[multiple[]];
var p;
par p from 0 to 3
{
```

```
(x::gather[3,2]):=r;
};
```

In this example, the variable x is allocated on the root process (2) only. Whereas r is allocated on all processes. In the assignment all three elements of r are gathered from each process and sent to the root process (2) and then placed into variable x in the order defined by the source's PID.

Scatter

Syntax

scatter[elements,root]

Semantics

Will send a number of elements (equal to *elements*) from the root process to all other processes.

Example

```
var x:array[Int,3]::allocated[multiple[]];
var r:array[Int,12]::allocated[multiple[]];
var p;
par p from 0 to 3
{
x:(x::scatter[3,1]);
x:=r;
};
```

In this example, three elements of array r, on process 1, are scattered to each other process and placed in their copy of x.

Alltoall

Syntax

alltoall[elementsoneach]

Semantics

Will cause each process to send some elements (the number being equal to *elementsoneach*) to every other process in the group.

Example

```
var x:array[Int,12]::allocated[multiple[]];
var r:array[Int,3]::allocated[multiple[]];
var p;
par p from 0 to 3
{
(x:alltoall[3]):=r;
};
```

In this example each process sends every other process three elements (the elements in its *r*.) Therefore each process ends up with twelve elements in *x*, the location of each is based on the source process's PID.

Allreduce

Syntax

allreduce[operation]

Semantics

Similar to the *reduce* type, but the reduction will be performed on each process and the result is also available to all.

Example

```
var x:Int::allocated[multiple[]];
var p;
par p from 0 to 3
{
(x::allreduce["min"]):=p;
};
```

In this case all processes will perform the reduction on p and all processes will have the minimum value of p placed into their copy of x.

3.3.6 Communication Mode

By default, communication in Mesham is blocking (i.e. will not continue until a send or receive has completed.) Standard sends will complete either when the message has been sent to the target processor or when it has been copied into a buffer, on the source machine, ready for sending. In most situations the standard send is the most efficient, however in some specialist situations more performance can be gained by overriding this.

By providing these communication mode types illustrates a powerful aspect of type based parallelism. The programmer can use the default communication method initially and then, to fine tune their code, simply add extra types to experiment with the performance of these different communication options.

Asynchronous

Syntax

async[]

Semantics

This type will specify that the communication to be carried out should be done so asynchronously. Asynchronous communication is often very useful and, if used correctly, can increase the efficiency of some applications (although care must be taken.) There are a number of different ways that the results of asynchronous communication can be accepted, when the asynchronous operation is honoured then the data is placed into the variable, however when exactly the operation will be honoured is non-deterministic. Care must be taken if using dirty values which is where a variable has not yet been synchronised and for some short time has potentially different unstable values on each process.

The *sync* keyword allows the programmer to either synchronise ALL or a specific variable's asynchronous communication. The programmer must ensure that all asynchronous communications have been honoured before the process exits, otherwise behaviour is undefined.

Example

```
var a:Int::allocated[multiple[]] :: channel[0,1] :: async[];
var p;
par p from 0 to 2
{
a:=89;
var q:=20;
q:=a;
sync q;
};
```

In this example, *a* is declared to be an integer, allocated to all processes, and to act as an asynchronous channel between processes 0 and 1. In the par loop, the assignment *a:=89* is applicable on process 0 only, resulting in an asynchronous send. Each process executes the assignment and declaration *var q:=20* but only process 1 will execute the last assignment *q:=a*, resulting in an asynchronous receive. Each process then synchronises all the communications relating to variable *q*.

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: async[];
var c:Int::allocated[single[on[3]]] :: async[];
```

```
a:=b;
c:=a;
b:=c;
sync;
```

This example demonstrates the use of the *async* type in terms of default shared variable style communication. In the assignment *a:=b*, processor 2 will issue an asynchronous send and processor 1 will issue a synchronous (standard) receive, which will block until data is received from processor 2. The second assignment, *c:=a*, processor 3 will issue an asynchronous receive and processor 1 a synchronous send, where processor 1 will block until processor 3 receives the data as variable *a* still uses, default, synchronous communication. In the last assignment, *b:=c*, both processors (3 and 2) will issue asynchronous communication calls (send and receive respectively.) The last line of the program will force each process to wait and complete all asynchronous communications.

blocking

Syntax

blocking[ ]

Semantics

Will force P2P communication to be blocking, which is the default setting

Example

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: blocking[];
a:=b;
```

The P2P communication (send on process 2 and receive on process 1) resulting from assignment *a:=b* will force program flow to wait until it has completed. The *blocking* type has been omitted from the that of variable *a*, but is used by default.

nonblocking

Syntax

nonblocking[ ]

Semantics

This type will force P2P communication to be non-blocking. In this mode communication (send or receive) can be thought of as having two distinct states - start and finish. The *nonblocking* type will start communication and allows program execution to continue between these two states, whilst blocking (standard) mode requires the finish state has been reached before continuing. The *sync* keyword can be used to force the program to wait until finish state has been reached.

Example

```
var a:Int::allocated[single[on[1]]] :: nonblocking[];
var b:Int::allocated[single[on[2]]];
a:=b;
sync a;
```

In the P2P communication resulting from assignment *a:=b*, process 1 will issue a non-blocking receive whilst process 2 will issue a blocking send. All non-blocking communication with respect to variable *a* is completed by the keyword *sync a*.

standard

Syntax

standard[ ]

Semantics

This type will force P2P sends to follow the standard form of reaching the finish state either when the message has been delivered or it has been copied into a buffer on the sender. This is the default applied if further type information is not present.

Example

```
var a:Int::allocated[single[on[1]]] :: nonblocking[] :: standard
    [];
var b:Int::allocated[single[on[2]]] :: standard[];
a:=b;
```

In the P2P communication resulting from assignment *a:=b*, process 1 will issue a non-blocking standard receive whilst process 2 will issue a blocking standard send.

<u>**buffered**</u>

Syntax

buffered[{buffersize}]

Semantics

This type will ensure that P2P Send will reach the finish state (i.e. complete) when the message is copied into a buffer of size *buffersize* bytes. At some later point the message will be sent to the target process. If *buffersize* is not provided then a default is used.

Example

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: buffered[500];
var c:Int::allocated[single[on[2]]] :: buffered[500] ::
    nonblocking[];
a:=b;
a:=c;
```

The P2P communication resulting from assignment *a:=b*, process 2 will issue a (blocking) buffered send (buffer size *500* bytes), which will complete once the message has been copied into this buffer. The assignment *a:=c*, process 1 will issue another send this time also buffered but nonblocking where program flow will continue between the start and finish state of communication. The finish state will be reached once the value of variable *c* has been copied into a buffer held on process 2.

ready

Syntax

ready[ ]

Semantics

The *ready* type will force P2P Send to start only if a matching receive has been posted by the target processor. When used in conjunction with the *nonblocking* type, communication start will wait until a matching receive is posted. This type acts as a form of handshaking and can improve performance in some uses.

Example

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: ready[];
var c:Int::allocated[single[on[2]]] :: ready[] :: nonblocking[];
a:=b;
a:=c;
```

The send of assignment *a:=b* will only begin once the receive from process 1 has been issued. With the statement *a:=c* the send, even though it is non-blocking, will only start once a matching receive has been issued too.

synchronous

Syntax

synchronous[ ]

Semantics

By using this type, the send of P2P communication will only reach the finish state once the message has been received by the target processor.

Example

```
var a:Int::allocated[single[on[1]]];
var b:Int::allocated[single[on[2]]] :: synchronous[] :: blocking
    [];
var c:Int::allocated[single[on[2]]] :: synchronous[] ::
    nonblocking[];
a:=b;
a:=c;
```

The send of assignment *a:=b* (and program execution on process 2) will only complete once process 1 has received the value of *b*. The send involved with the second assignment is synchronous nonblocking where program execution can continue between the start and finish state, the finish state only reached once process 1 has received the message (value of *c*.) Incidentally, as already mentioned, the *blocking* type of variable *b* would have been chosen by default if omitted (as in previous examples.)

```
var a:Int :: allocated[single[on[0]]];
var b:Int :: allocated[single[on[1]]];
a:=b;
a:=(b :: synchronous[]);
```

The code example above demonstrates the programmer's ability to change the communication send mode just for a specific assignment. In the first assignment, process 1 issues a blocking standard send, however in the second assignment the communication mode type *synchronous* is coerced with the type of *b* to provide a blocking synchronous send just for this assignment only.

3.3.7 Partition

Often in data parallel HPC applications the programmer wishes to split up data in some way, shape or form. This is often a difficult task, as the programmer must consider issues such as synchronisation and uneven distributions. Mesham provides types to allow for the partitioning and distribution of data, the programmer needs just to specify the correct type and then behind the scenes the compiler will deal with all the complexity via the type system. It has been found that this approach works well, not just because it simplifies the program, but also because some of the (reusable) codes associated with parallelization types are designed beforehand by expert system programmers. These types tend to be better optimized by experts than the codes written directly by the end programmers.

When the programmer partitions data, the compiler splits it up into blocks (an internal type of the compiler.) The location of these blocks depends on the distribution type used - it is possible for all the blocks to be located on one process, on a few or on all and if there are more blocks than processes they can always "wrap around." The whole idea is that the programmer can refer to separate blocks without needing to worry about exactly where they are located, this means that it's very easy to change the distribution method to something more efficient later down the line if required.

The programmer can think of two types of partitioning - partitioning for distribution and partitioning for viewing. The partition type located inside the allocated type is the partition for distribution (and also the default view of the data.) However, if the programmer wishes to change the way they are viewing the blocks of data, then a different partition type can be coerced. This will modify the view of the data, but NOT the underlying way that the data is allocated and distributed amongst the processes. Of course, it is important to avoid an ambiguous combination of partition types. In order to access a certain block of a partition, simply use array access # or *[ ]* i.e. *(a#3)* will access the 3rd block of variable *a*.

In the code *var a:array[Int,10,20] :: allocated[A[m] :: single[D[]]];*, the variable *a* is declared to be a 2d array size 10 by 20, using partition type *A* and splitting the data into *m* blocks. These blocks are distributed amongst the processes via distribution method D.

In the code fragment *a:(a::B[])*, the partition type B is coerced with the type of variable a, and the view of the data changes from that of A to B.

Horizontal

Syntax

horizontal[blocks]

Where *blocks* is number of blocks to partition into.

Semantics

This type will split up data horizontally into a number of blocks. If the split is uneven then the extra data will be distributed amongst the blocks in the most efficient way in order to keep the blocks a similar size. The figure 3.2 illustrates the horizontal partitioning of an array into three blocks.

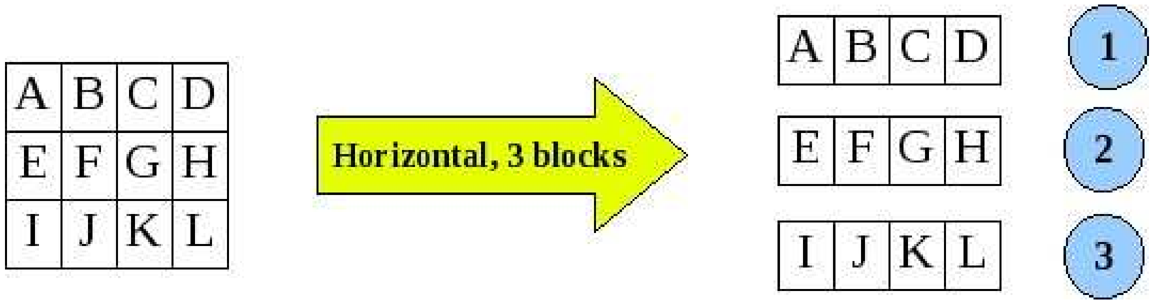

Figure 3.2: Horizontal Partitioning of data

Communication

There are a number of different default communication rules associated with the horizontal partition, based on the assignment *assigned variable:=assigning variable* which are detailed in table 3.5. As in the last row of table 3.5, if the two partitions are the same type then a simple copy is performed. However, if they are different then an error will be generated as Mesham disallows differently typed partitions to be assigned to each other.

Assigned Variable	Assigning Variable	Semantics
single	partition	Gather
partition	single	Scatter
partition	partition	Local Copy

Table 3.5: Partition type communication in assignment

Horizontal blocks also support *.high* and *.low*, which will return the top and bottom bounds of the block

Vertical

Same as *horizontal*, but will partition vertically rather than horizontally. Figure 3.3 illustrates partitioning an array vertically into 4 blocks.

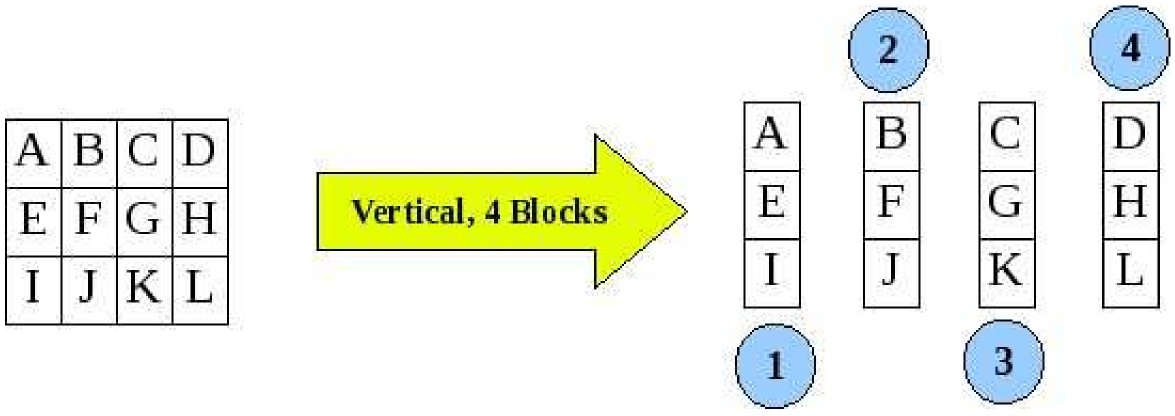

Figure 3.3: Vertical Partitioning of data

Arraymapped

Syntax

arraymapped[blocks,indexesperprocess,indexes]

Semantics

Given an index list (integer array), this type will move each element of a block to its new position based on this list. Moving between blocks, and any communication required, is automatically dealt with.

3.3.8 Distribution

<u>Evendist</u>

Syntax

evendist[ ]

Semantics

Will distribute data blocks evenly amongst the processes. If there are too few processes then the blocks will wrap around, if there are too few blocks then not all processes will receive a block. Figure 3.4 illustrates even distribution of 10 blocks of data over 4 processes.

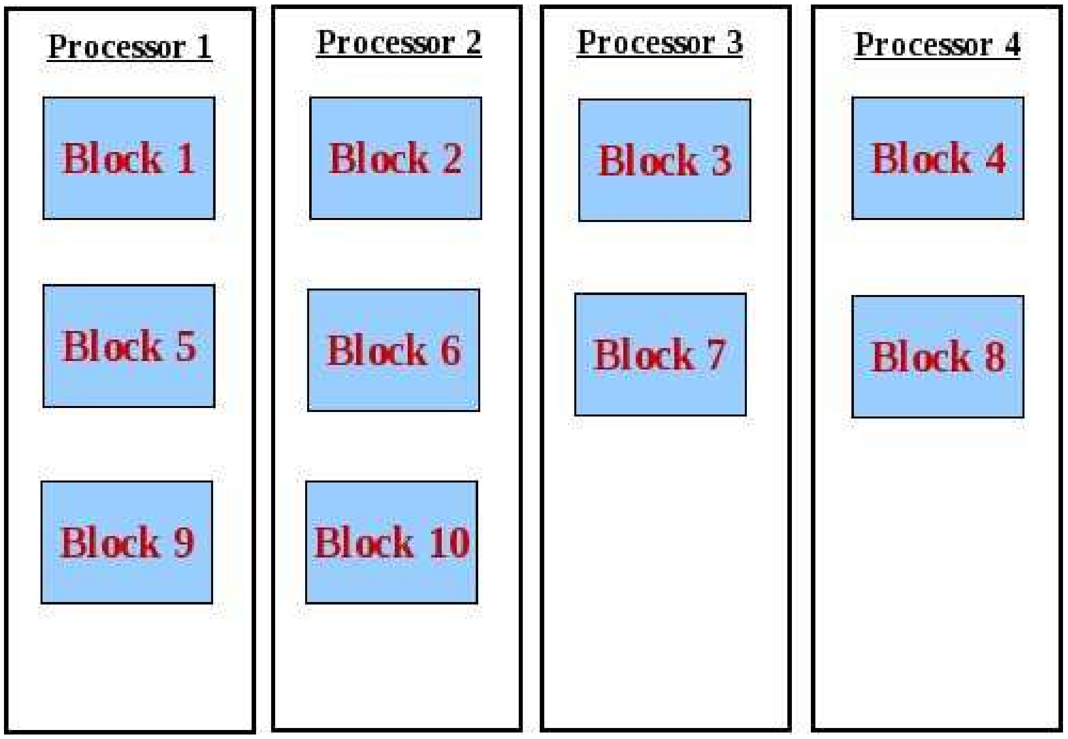

Figure 3.4: Even distribution of 10 blocks over 4 processes

Example

```
 1 var a:array[Int,16,16] :: allocated[row[] :: horizontal[4] ::
       single[evendist[]]];
 2 var b:array[Int,16,16] :: allocated[row[] :: vertical[4] ::
       single[evendist[]]];
 3 var e:array[Int,16,16] :: allocated[row[] :: single[on[1]]];
 4 var p;
 5 par p from 0 to 3
 6 {
 7         var q:=(((b#p)#2)#3);
 8         var r:=(((a#p)#2)#3);
 9         var s:=((((b :: horizontal[])#p)#2)#3);
10 };
11 a:=e;
```

In this example (which involves 4 processors) there are three arrays declared, *a*, *b* and *e*. Array *a* is horizontally partitioned into 4 blocks, evenly distributed amongst the processors, whilst *b* is vertically partitioned into 4 blocks and also evenly distributed amongst the processors. Array *e* is located on processor 1 only. All arrays are allocated row major. In the par loop, variables *q*, *r* and *s* are declared and assigned to be values at specific points in a processor's block. Because *b* is partitioned vertically and *a* horizontally, variable *q* is the value at *b*'s block memory location 11, whilst *r* is the value at *a*'s block memory location 35. On line 9, variable *s* is the value at *b*'s block memory location 50 because, just for this expression, the programmer has used the *horizontal* type to take a horizontal view of the distributed array. It should be noted that in line 9, it is just the view of data that is changed, the underlying data allocation is not modified. In line 11 the assignment *a:=e*, as per table 3.5, results in a scatter.

3.3.9 Composition

Record

Syntax

record[name$_1$, type$_1$,name$_2$,type$_2$,.....,name$_d$,type$_d$]

Semantics

The *record* type allows the programmer to combine d attributes into one, new type. There can be any number of names and types inside the record type. A record type is very similar to a typedef structure in C. To access the member of a record use the dot, .

Example

```
var complex : record["r",Float,"i",Float];
var person: record["name",String, "age",Int, "gender",Char];
var a:array[complex,10];
(a#1).i:=22.3;
var b:complex;
var me:person;
me.name:="nick";
```

In the above example, *complex* (a complex number) is a record with two *float* elements, *i* and *r*. The variable *b* is defined as a complex number and *a* as an array of these numbers. The variable *me* is of type *person*.

Reference Record

Syntax

referencerecord[name$_1$, type$_1$,name$_2$,type$_2$,.....,name$_d$,type$_d$]

Semantics

The *record* type may NOT refer to itself (or other records) where as reference records support this, allowing the programmer to create data structures such as linked lists and trees. There are some added complexities of reference records, such as communicating them (all links and linking nodes will be communicated with the record) and freeing the data (garbage collection.) This results in a slight performance hit and is the reason why the record concept has been split into two types.

Example

```
var node:referencerecord["prev",node,"data",Int,"next",node];
var head:node;
head:=null;
var i;
for i from 0 to 9
{
var newnode:node;
newnode.data:=i;
newnode.next:=head;
if (head!=null) head.prev:=newnode;
head:=newnode;
};

while (head != null)
{
        print[head.data,"\n"];
        head:=head.next;
};
```

In this code example a doubly linked list is created, and then its contents read node by node.

3.4 Core Language

3.4.1 General

Sequentially, the core language looks similar to an imperative language such as C. Each program statement is joined via sequential composition *;* or parallel composition ‖. In order to call functions (either user defined or language predefined) the programmer should use the syntax *fnname[args]*.

3.4.2 Declaration

Syntax

var name{:=value};

Semantics

Will define the variable in the current environment and assign a value to it if provided

Examples

```
var a;
var b:=23;
```

In this example variable *a* is defined, but no value associated. Variable *b* is defined to be the value 23 and, by type inference, has type *Int*.

Assignment

Syntax

lvalue:=rvalue; *(where rvalue is a variable or value, lvalue is a variable)*

Semantics

rvalue is assigned to *lvalue*

Examples

```
var a;
var b:=99;
a:="hello";
```

In this example variable *a* is defined, but no value associated initially. As the program progresses the string *"hello"* is assigned to *a* and by type inference the type of this variable becomes *String*. Variable *b* is defined to be the value 99 and, by type inference, has type *Int*.

3.4.3 Statements

Body

Syntax

body = statement | (body ; body)

Semantics

A variety of Mesham's statements include with them a code body.

Conditional

Syntax

```
if (condition)
    then-body;
{ else
    else-body;}
```

Semantics

If the condition is *true* then execute the *then-body*, otherwise execute the *else-body* (if it exists.)

Loops

Syntax

```
while (condition)
    while-body;
```

```
for i from a to b
    for-body;
```

Semantics

These will loop whilst the condition holds. The *for* loop can be thought of as syntactic sugar for a *while* loop, incrementing the variable after each pass.

Break

Syntax

```
break;
```

Semantics

Will "break" out of the directly enclosing loop.

Try

Syntax

```
try
    try-body;
catch (error string)
    error handing code;
```

Semantics

Will execute the code in the *try-body* and handle any errors. This is very important in parallel computing as it allows the programmer to easily deal with any communication errors that may occur.

String	Description
“”	All errors
“Array Bounds”	Accessing an array outside its bounds
“Divide by zero”	Divide by zero error
“Memory Out”	Memory allocation failure
“root”	Illegal root process in communication
“rank”	Illegal rank in communication
“buffer”	Illegal buffer in communication
“count”	Count wrong in communication
“type”	Communication type error
“comm”	Communication communicator error
“truncate”	Truncation error in communication
“Group”	Illegal group in communication
“op”	Illegal operation for communication
“arg”	Arguments used for communication incorrect

Table 3.6: Error strings supported by Mesham

Error Strings

Table 3.6 lists all the error strings built into Mesham. The programmer can specify additional error strings simply by *throwing* them.

Throw

Syntax

throw error string;

Semantics

Will throw the error string, and either cause termination of the program or, if caught by a try catch block, will be dealt with.

Example

```
try
{
throw "an error"
} catch "an error" {
print["Error occurred!\n"];
};
```

In this example, a programmer defined error (*an error*) is thrown and caught.

Parallel Composition

Syntax

a-body || b-body

Semantics

The parallel equivalent of sequential composition, code blocks *a-body* and *b-body* will execute at the same time on different processors.

Example

```
var j:=23 || (var q:=9; print[q,"\n"])
```

One process will declare j to be 23, whilst the other will declare q to be 9 and display it.

Par Loop

Syntax

par p from a to b

    par-body;

Semantics

The parallel equivalent of the *for* loop, each “iteration” will execute concurrently on different processes. This allows the programmer to write code MPMD style, with the limitation that bounds a and b must be known during compilation. All (variable sharing) communication in a *par* loop is performed using one sided communication, whereas variable sharing SPMD style is performed using synchronous communication for performance reasons. A *par* loop over n processors is a more convenient way of writing out the body n times using parallel composition.

Example

```
var p;
par p from 0 to 10
{
        print["Hello from process ",p,"\n"];
};
```

The code fragment will spawn 11 processes (0 to 10 inclusive) and each will display a message.

Process Selection

Syntax

proc n
    proc-body;

where n is a variable or value

Semantics

This will limit execution of a block to a certain process

Example

```
proc 0
{
        print["Hello from 0\n"];
};

proc 1
{
        print["hello from 1\n"];
};
```

The code example will run on two processes, the first will display the message *Hello from 0*, whilst the second will output the message *hello from 1.*

Synchronisation

Syntax

sync {name};

Semantics

Will synchronise processes where they are needed. For instance, if using the asynchronous communication type, the programmer can synchronise with a variable name and the keyword will ensure all communications of that variable are up to date. One sided communication (variable sharing MPMD style in a *par* loop) is also linked into this keyword and it will ensure all communication is completed. Without a variable will synchronise all outstanding variables that need synchronising. If a process has no variables that need syncing then it will ignore this keyword and continue.

Skip

Syntax

skip;

Semantics

Does nothing

3.4.4 Functions

Function

Syntax

function returntype name[arguments]

function-body;

Semantics

In a function all arguments are pass by reference (even constants). If the type of argument is a type chain (requires *::*) then it should be declared in the body

Example

```
function Int add[var a:Int,var b:Int]
{
return a + b;
};
```

This function takes two integers and will return their sum.

The main function

Returns void, and like C, it can have either 0 arguments or 2. If present, the first argument is number of command line interface parameters passed in, 2nd argument is a *String* array containing these. Location 0 of the *String* array is the program name.

3.4.5 Supported Operators

Mesham supports a variety of operators, these are detailed in table 3.7.

Operator	Description
$+$	Addition
-	Subtraction
$*$	Multiplication
%	Division
$\ll$	Bit shift to left
$\gg$	Bit shift to right
$==$	Test for equality
$!=$	Test for inverse equality
$=$	Test of equality on strings
$\prec$	Test lvalue is smaller than rvalue
$\succ$	Test lvalue is greater than rvalue
$\leq$	Test lvalue is smaller or equal to rvalue
$\geq$	Test lvalue is greater or equal to rvalue
$\|$	Logical OR
&&	Logical AND

Table 3.7: Operators supported by Mesham

3.5 Conclusion

As has been seen, Mesham is a language which utilises the type system to provide the expressiveness required when dealing with parallel programming. Other parallel languages require some mechanism for parallelism but whether this is provided for via keywords, functions or implicitly each has major downsides. By moving the complexity out into the type system, not only does this make the programmer's job easier (once they have learnt the new paradigm) but it also makes the language simpler to design and implement. The core language of Mesham is actually very simple, and by using the type-based approach it means that modifying the parallel aspects is simple to do by adding, removing or editing loosely coupled types only.

Chapter 4

Implementation

4.1 Introduction

"A successful language must grow out of clear ideas of design goals and of simultaneous attempts to define it in terms of abstract structures, and implement it on a computer." [Wirth1974] For all the language definition provided in Chapter 3, if it is not possible to implement then Mesham is of little use. This chapter provides an overview of the implementation, not only of the compiler itself but also important choices made about the target code.

4.2 Literature Review

In order to implement a compiler and produce highly efficient target code a number of existing literature resources have been employed to act as a solid foundation to the process. These are detailed in this section.

4.2.1 Flexibo

"Flexibo is an executable object-oriented specification language designed for open-source software development with different levels of trust in a decentralised programming environment." [Chen2004-2] Flexibo is an interpreted language designed, amongst other things, to be used to write translation tools. The type system of Flexibo is dynamic, allowing user defined types to be created during runtime, which provides the programmer with a degree of power and flexability. In order to form the basis of a compilation tool, Flexibo provides a reflection system. “Reflection provides a way to examine and manipulate the runtime environment programmatically. This has several benefits, such as being able to discover types, methods, and properties at runtime; being able to access and manipulate attributes at runtime; and being able to invoke new methods at runtime.” [Harrison2003] Within the context of Flexibo, each program construct can be viewed as an object of a particular reflection class and new reflection classes and thus source language constructs can be defined by the programmer. Flexibo is important with respect to this project, as it will be the language used for the creation of the translation tool.

The reflection system of Flexibo provides for a compilation tool quite different from existing tools. Instead of defining the language syntactically and then using a lexer and parser to work on the syntax, the source language of a Flexibo compiler has completely flat syntax and the translation is carried out from the semantics. An advantage to this approach is that if a language mechanism is to be added, modified or deleted, then substantial changes are not required to a lexer and parser. Taking this approach to compiling does not mean that there are no restrictions to syntax combination, as the compiler maintains a state it is quite easy to deduce what has come previously and using concepts such as a labelled transition system [Prasad2003] it is possible to create a graph of allowed syntactic combinations.

The translator described in [Zhou2005] has been written in Flexibo. This translator introduces a number of comfortable language mechanisms, such as types and LOGS specific constructs. The translator is far from being a polished finished product, however it demonstrates well the translation techniques used and, as it was written by the developers of Flexibo, it also illustrates what the language is capable of and how. As mentioned previously, the translator of [Zhou2005] uses abstract interpretation and range analysis, it is interesting to view how this is implemented. It has been decided to start from scratch with this project, however, the existing LOGS translator acts as a good illustration to both what is required process wise by the compiler and what can be done better.

Another translator [Brown2006] has been written in Flexibo, in this case to translate an imperative subset of Flexibo into efficient C code. The translator is much larger than the one in [Zhou2005] and translates a larger source language, although it is no where near as well written or illustrative of good analysis techniques. From these two translators, it can be concluded that Flexibo is certainly up to the job of providing a complex compilation tool. From the short-comings of [Brown2006] it can be seen that good software engineering techniques are important when constructing a compilation tool, due to the complexity and size of the tool to be produced.

[Chen2004-2] is a relevant paper describing the language. However, this paper is aimed at describing the innovative aspects of the language and does not provide a great amount of detail towards actually writting Flexibo code. Because of the new, experimental nature of the language there are no manuals or particularly detailed examples apart from the two translators. [Zhou2005] and [Brown2006] will be used extensively, as will the source code of Flexibo, as a reference to programming the language. This task is made harder due to the fact that Flexibo does not provide particularly useful error messages, so as noted a good software engineering process is essential.

4.2.2 C Programming

As mentioned previously, due to efficiency, writting programs using a combination of C with a parallel programming library (such as MPI) is common. Due to this efficiency and the maturity of MPI, it has been decided that this combination will be used as the target language for the compiler. On this note it will be important to be able to write efficient, safe C code and thus gaining a high understanding of C and MPI is critical. There are a number of resources available for use to this end.

In order to learn C a book [Kernighan1989] has been used. This book, written in part by the creator of C, is considered a core text and is hugely useful not just as a guide but also as reference material. The downside to [Kernighan1989] is that it is somewhat out of date, it covers C89 but not later versions of the standard. Having said that combined with other reference material it is considered an invaluable resource. A web based resource has been found at [Leslie2005] which not only provides the usual general information but also has a quick reference, with examples, to the

POSIX standard C functions also. A downside to [Leslie2005] is that it is still very much in production and there are parts which are unfinished nor does it cover the standard in as much detail as [Kernighan1989]. In addition, being a quick reference guide, it does not go into the amount of detail sometimes required. Lastly, the C FAQ [cFAQ] provides many answers to common problems and is useful as both a knowledge base and debugging guide.

In order to learn how to use the Message Passing Interface [MPI1995] practically the book, [Gropp1999], is considered a core text. This resource provides the reader with basic, intermediate and advanced details about MPI 1. Associated with this book are number of web based exercises which can be used to help both with practical knowledge about MPI and also assist in learning C. As noted in Section 2.5, the examples of [Pacheco1996] are also a very useful resource, both to use as a base for practising C programming and MPI function usage.

4.2.3 Compiler Creation

In Section 2.4 a number of theoretical concepts and their literature have been considered and discussed. However, in order for this project to be a success, it shall be important to use these practically to achieve an efficient, reliable compilation tool. As previously noted, [Aho2006] provides an in depth introduction to the topic of compiler construction and is considered by many as a core text in this area. One downside of [Aho2006], in reference to this project, is that it considers the more traditional compiler approach rather than the reflection approach adopted here. Having said that, there is considerable cross over and as such the text is still very relevant.

A useful resource is [Zhou2005]. This paper details the existing LOGS compilation tool and notes what techniques were used for analysis. The LOGS translator demonstrates the practical use of abstract interpretation and range analysis, which is considered both interesting the useful to use as an example. Due to the fact that the tool described was written in Flexibo, like the tool for this project, the literature is directly relevant and the tool might even provide some reusable functionality.

Another book [Terry1986] has also been used. This book is a practically oriented introduction to the topic and uses Pascal [Jensen1991] to illustrate the techniques discussed. This book is very much syntax oriented, with little information about other translation phases such as static analysis. The book also considers in detail assemblers, which is not relevant to this project. With that in mind however there are some useful introductory segments which consider the compilation as a whole. For instance, the book introduces the term of bootstrapping, where a source language is translated into a target language, which is then used as the source language for another translator and so on. This bootstrapping follows the process which will be implemented by this project. It can be concluded that the book provides for a useful introduction, however many of the topics considered are either too simple or not relevant with regard to this project. In addition, the large amount of concrete Pascal source code used to illustrate points and at the end of each chapter requires expertise in both programming and Pascal, it would have been better if the author had used pseudo code instead.

4.3 Overview

In the domain of parallel computing, in construction of the compiler there have been a number of important requirements.

- Simple to compile
- Executable simple to run
- Portable
- Run on multi-core machines

The core translator produces ANSI standard C99 C code which uses the Message Passing Interface (MPI) version 2 for communication. On the target machine, an implementation of MPI, such as OpenMPI, MPICH or a vendor specific MPI is required and as long as they implement MPI-2 they will be compatible with the generated code. The language runtime library must also be available, which contains language functionality support. Figure 4.1 details an overview of this process.

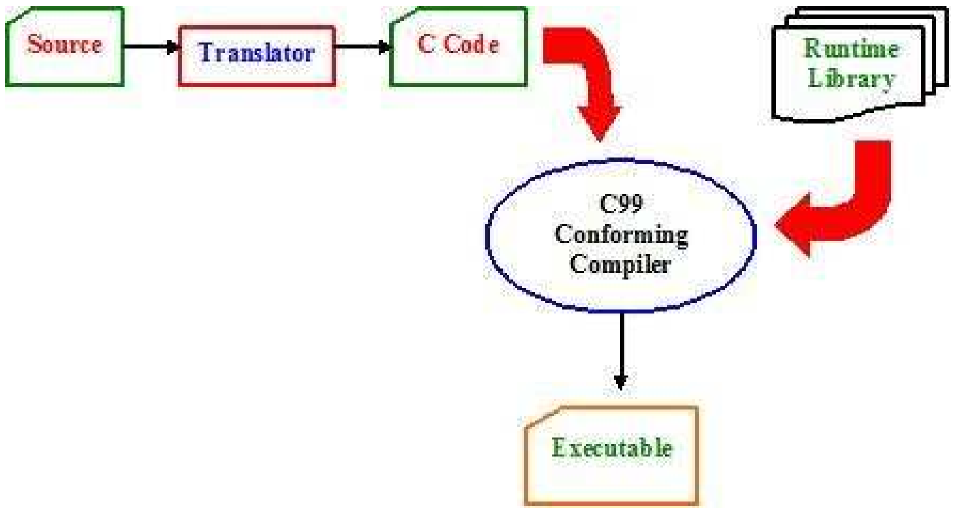

Figure 4.1: Overview of Compilation Process

The resulting executable can be thought of as any normal executable, and can be run like any other executable with the program automatically spawning the number of processors required. Additionally, the executable can also be run via the MPI daemon, and may be instigated via a process file or queue submission program which is common practice for execution on a cluster. It should be noted that, as long as the MPI implementation supports multi-core, then the code can be executed properly on any multi core machine with the processes wrapping around the cores (for instance 2 processes on 2 cores is 1 process on each, 6 processes on 2 cores is 3 processes on each.) This adds additional flexability to the language without any recoding being required for different architectures; although the programmer might experiment with different types to improve efficiency this will not affect correctness.

The translator itself, as detailed in figure 4.2 is contained within a number of different phases. Firstly, the Mesham code goes through a preprocessor, written in Java, which will do a number of actions such as adding scoping information and honouring preprocessor directives. When this is completed the code is then sent to the translation server - from the design of Flexibo, the language the translator is written in, the actual translation is performed by a server listening using TCP/IP. This server can be on the local machine, or a remote one, depending on the network configuration. Once translation has completed, the generated C code is sent back to

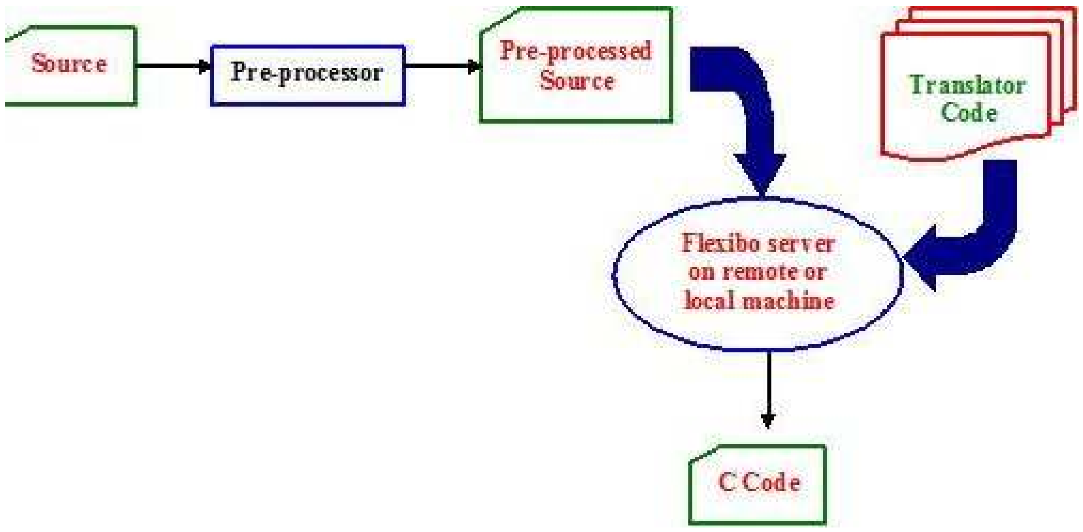

Figure 4.2: Overview of Translator

the client via TCP/IP and from there can be compiled. The most important benefit of this approach is flexibility Mesham code can be compiled and executed via the command line or a web based interface, with scope for further support.

4.4 Preprocessor

The preprocessor acts to turn Mesham source code into a slightly modified form understandable by the Flexibo reflection system. Written in Java, this stage of compilation performs a number of important jobs such as adding scope information, honouring preprocessor directives (to include other source code files) and turning some shortened, more convenient syntax into its complete form. The preprocessor has been designed to be lightweight, efficient and simple to modify. Internally, pattern matching is used to find specific syntactic atoms and apply rules to them.

4.5 Translator

The actual translator is written in a language called Flexibo. After passing through the preprocessor, the code is then fed into the Flexibo translation system. Flexibo is an executable object-oriented specification language that supports open-source software development in a decentralised multi-user environment with different levels of trust. Critically, with its reflection system, it is designed to be used to prototype and rapidly develop compilers. By creating subclasses of Flexibo's reflection classes the programmer is able to add their own specific functionality, with the language taking care of activities such as lexing and parsing. For instance the class *SemBinaryCondition* is used by Flexibo to represent conditional statements, by creating a subclass it is possible to add specific methods which made up the translation system.

In designing the compiler there was the aim of creating a flexible system, which could handle major changes to the language design. To this end the translator was split into three distinct parts - the core, the type library and the function library. The core, containing support for language statements (Section 3.4) and naturally tightly coupled, has been designed to contain minimal code. The complexity of the language and majority of support for parallelism is contained within the type library. The only interaction these two systems have is via a number of service calls, to and from the core to the library. Each type is similarly linked to other types via these same service calls. The major advantage of this approach is that adding, removing and modifying the majority of the language (the types) is very simple, and there is no worry of side effect. It was found that this was of great advantage in creating the compiler. The function library contains language defined functions, such as mathematical support and IO, similar to the type library, these functions honour specific services called from the translator core. This process can be seen in figure 4.3, which illustrates the concept. As can be seen the type and function libraries communicate, if required, via the translator core which will marshall messages.

In designing the type library as explained in Section 3.3 additional implementation issues than simply using an OO approach were found. As already explained, a variable's type is many individual types connected together, the concept is when a service is passed to the supertype if the first type in the chain does not honour it, then this is passed to the next type, if this does not honour it then it is passed to the next and so on. If no types in the supertype honour the service then at the end of the chain a set of defaults for each service are provided. Services are honoured via methods, for instance the service *generateAssignment* is honoured by a

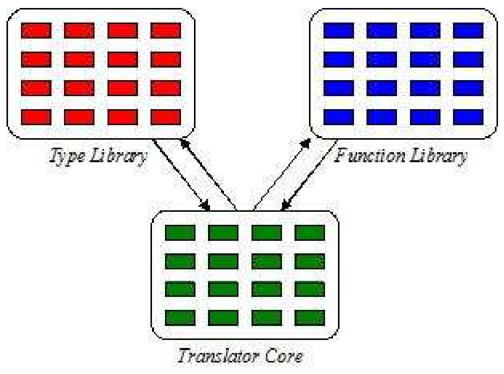

Figure 4.3: Interaction of translator core and libraries

method called *generateAssignment* which takes two arguments. Initially this looked like a traditional OO class hierarchy, with types forming subclasses, however as the Mesham programmer is allowed to combine types in many different ways this requires a much more dynamic approach. Rather than coding into the translator all the different possible type combinations (via class hierarchies) which would be error prone and tiresome, it was decided to allow these classes to "connect" to each other dynamically. To facilitate this, each type is a subclass of the *coretype* class. When a new type is combined the last type in the chain's *coretype* maintains a reference to this next type. The *coretype* honours all service calls and passes them into the next type, the assumption being that if the type honours a specific service call then this method will have been overridden and as such *coretype*'s method will never be reached. The last type in the chain's *coretype* has as its next type *default*, which for each service call provides a default action called if this has not been honoured by any of the types in the supertype chain. This approach has the added benefit of flexibility, it is very easy to modify the loosely coupled types and even change the underlying service calls.

Figure 4.4 illustrates this concept with an example type chain *array[] :: allocated[multiple[]]*. Here the start point is *allocated*, whose superclass *coretype* makes reference to the next type in the chain, *array*. As the *array* type is the last in the chain, its superclass simply points to the *default* type class. As an added complication, the *multiple* type is provided as an argument to the *allocated* type, in this instance the *allocated* type will reference this directly as an argument.

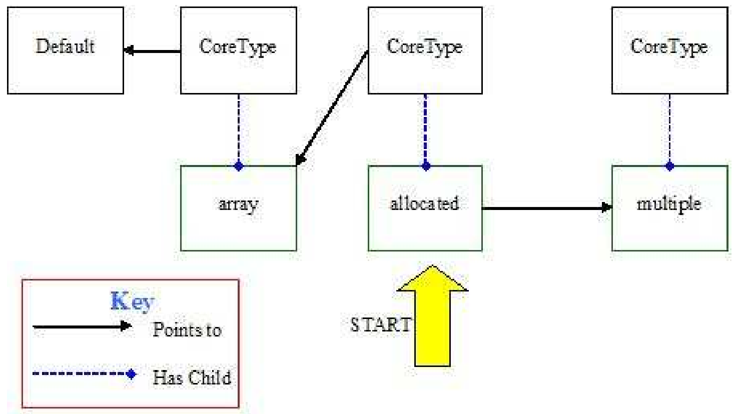

Figure 4.4: Dynamic linking of type classes

4.5.1 Reflection Representation Example

To some people the concept of reflection and how this relates to traditional compilers can be somewhat confusing. To this end a simple example is provided demonstrating a traditional abstract syntax tree and the reflection concept used by the Mesham Compiler. Code listing 4.1 is used as the Mesham source program in this example.

```
var j;
var i;
for i from 0 to 10
{
        j:=j + i;
};
print[j];
```

Listing 4.1: Reflection Example

Figure 4.5 illustrates the abstract syntax tree of the source code. In this tree terminals are either variables or constants with the compiler traversing the tree during its work. Source code manipulated into this form is a very common practice and Gnu's Bison is a popular tool to produce such a representation.

Figure 4.6 illustrates this same source code when processed by Flexibo's reflection system. It can be seen that, although these two representations do share some similarities, in many ways they are very different. The Flexibo class *SeqComp* is

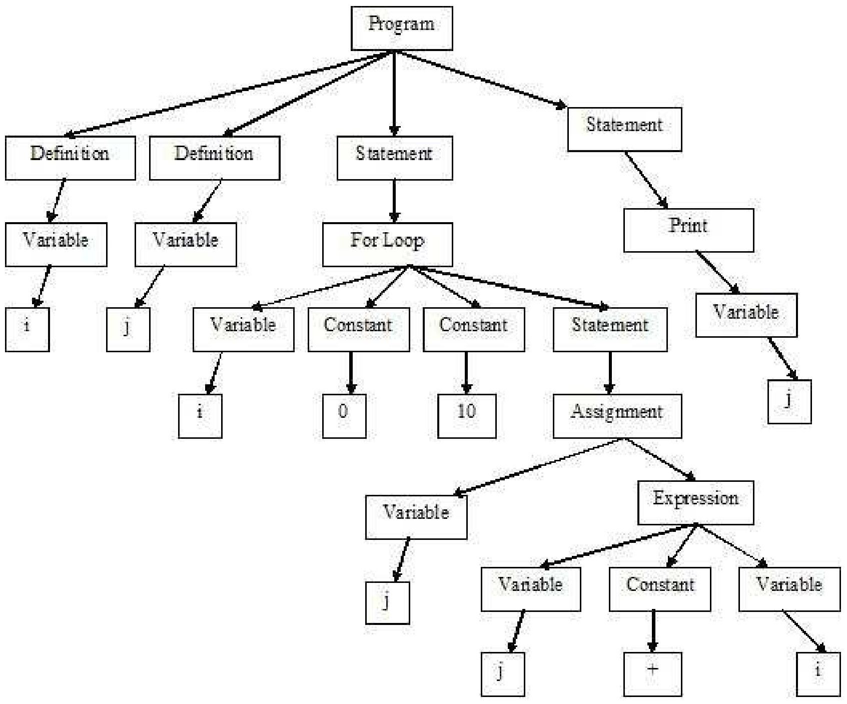

Figure 4.5: Abstract Syntax Tree of Code Listing 4.1

generated whenever sequential composition is encountered, this class has two variables *a* and *b*. Variable *a* points to the reflection system representation of the code before the *;* symbol and *b* to the code after this point. It can be seen from the diagram that this object can often point to other instances of the same object. An instance of the class *PubStaticVariable* is created for each distinct variable in the code, a member of this class *name* allows the programmer to reference the name of the variable. The benefit of this approach is that, as each variable is represented by one object throughout the code, keeping track of attributes such as the variable's value is relatively simple. As discussed in Section A.2.2, the *print* statement of Mesham is a function. This is dealt with initially by the class *SemMethodInvoke* which will point to the *print* class in the external function library.

An example of how this reflection representation might be used is with the method *generateCode*. For each class in figure 4.6 a subclass has been created with the method *generateCode*. This method is called on the first instance of *SeqComp*, which will call the same method on variables *a* and *b*. When this method reaches *VarInit* the class will not only pass this onto *PubStaticVariable*, which will output the variable name, but it will also output the other C code required to declare the variable. By following this example it can be seen how each class will encounter a call to this method, which can be implemented in a specific manner to satisfy that program construct.

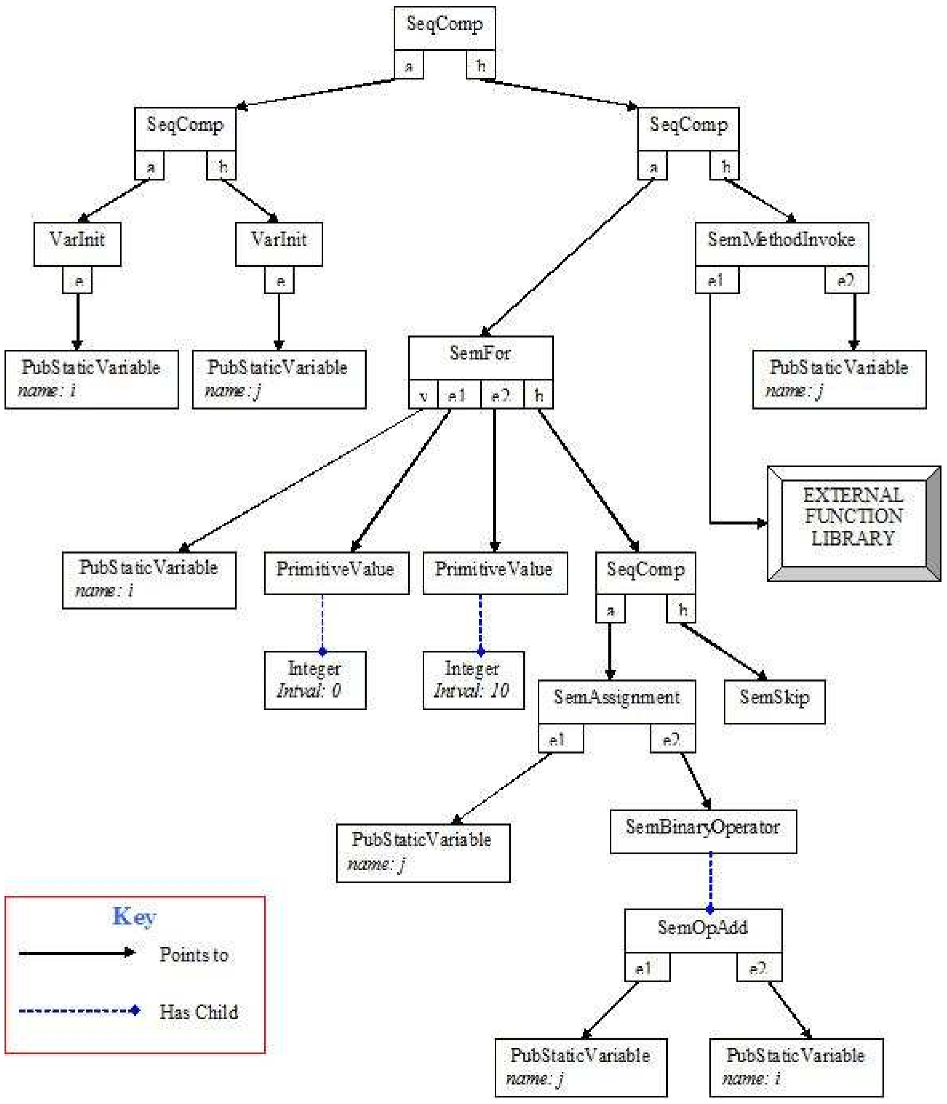

Figure 4.6: Reflection System Representation of Code Listing 4.1

4.6 Runtime Library

An important aspect of the design is that C code generated by the Mesham translator is linked with a language runtime library (RTL) which contains support for much of the language functionality. The first reason for this approach is for portability, all architecture specific (non-portable) code is contained within the library and as such a version exists for each target machine class (such as Linux, Windows, Solaris.) This means that the translator need only generate a single C program, which can then be sent to many very different machines for compilation and execution. Extensions, such as for Gadget-2, are by their very nature platform specific and require third party libraries such as HDF5, allowing for switches in the makefile has meant that these can easily be included or not as required by the end user.

Secondly using a runtime library cuts down on compilation time and code size, as the RTL contains commonly used functions which would otherwise need to be contained in the generated code.

The runtime library can either be included as a shared (dynamic) or static library. Using the shared approach, at runtime the executable will find the RTL on the machine and use its functions, advantages of this are that a change in the library need not require the entire code to be recompiled and the executable is smaller, the disadvantage is that each end user must have a version of this library compiled on their machine (such as a DLL file on Windows.) Statically linking to the RTL actually places a copy of the library inside the executable increasing its size and requiring entire recompilation after each modification of the library. However the main advantage of this is that the end user need not have a compiled version of the RTL on their machine. Using the shared linking approach has been found to be preferable, although configuration options are provided so this can be changed in the Mesham compiler.

4.7 Code Translation Example

The simple code in listing 4.2, similar to an example in Chapter 3, involves two processes. The first process (0) holds variable a, and as the program progresses will receive a number from process 1 and store it in this variable. Process 1, as well as holding variable a also holds variable b, the process will write the value of *23* into b (line 6) and then both copy this into its own a and also send the value to process

0 (line 9.) Both processes will also display a message on stdio.

```
1 var a:Int :: allocated[multiple[]];
2 var b:Int :: allocated[single[on[1]]];
3 var p;
4 par p from 0 to 1
5 {
6 if (p==1) b:=23;
7 print["Hello from process ",p,"\n"];
8 };
9 a:=b;
```

Listing 4.2: Mesham Code Example

The C code of listing 4.3 is that generated by the translator when the Mesham source code of listing 4.2 is used. The first five lines are generated in order to allow for tracking of the source code. Lines 7,8 and 9 include header files as required, in this example the only header files needed are *mpi.h* for communication with MPI, *mesham.h* for the runtime library (required during program start up) and C's *stdio.h* header for I/O. Lines 11 to 13 define some commonly used program constants, with lines 15 and 16 providing storage for global information (a process's id number, the number of processes, number of arguments passed to the program and the arguments themselves.)

The generated code is structured MPMD style, such that each process has its own section of code (although sequential functions will be SPMD to reduce code length.) Although this can increase compile time, under experimentation it was found to be the best option due to allowing for a great deal of optimisation for each process to be performed. The two functions *MESHprocessor0* and *MESHprocessor1* represent the code body for process 0 and 1 respectively. It can be seen that, as is required by the Mesham source, the integer b exists on process 1 but not 0, as does the assignment $b=23$. Outside of the Mesham par loop, from the generated code the reader can see how the $a:=b$ assignment and associated communication is dealt with. On process 1, in order to achieve the assignment, the code $a=b$ is first issued (line 9.) To send this value to other processes, as per the semantics of the relevant types, the compiler chooses to use a broadcast (via *MPI_Bcast*), which can be seen on lines 25 and 40 respectively. All the complexity of the communication (having to specify the data type, size, root, communication group and ensure communications complete) is taken care of by the compiler. As the reader can see, the Mesham programmer does not need to know or care exactly what communication method is used, as long as the result is achieved.

Lines 43 to 50 set up the processes' MPMD code as function pointers, with the *main* (program entry) function starting at line 52 performing tasks such as initialising MPI, the Mesham runtime library, setting up error handlers firstly and then passing execution to these functions. Before program termination execution returns to the *main* function which will shutdown MPI and return an integer as the C99 standard requires. The last three lines, 70 to 72 provide information about the generated code. This metadata is available to other tools and has been used to create automatic tool chain compilers.

```
 1 //Compiled on 30/3/2009 at 15:54:58 with Mesham V0.50 beta
 2 //Syntactic Check - OK
 3 //Type Check - OK
 4 //Static Optimise and Check - OK
 5 //Number of Processes - 2 with 0 Synchronisation Points
 6
 7 #include ”mpi.h”
 8 #include <mesham.h>
 9 #include <stdio.h>
10
11 #define null NULL
12 #define false 0
13 #define true 1
14
15 int myrank, numberofprocesses, MESHargc;
16 char ** MESHargv;
17 void MESHprocessor0()
18 {
```

```
int a;
int p;
{
p=0;
printf("Hello from process %d\n",p);
}
MPI_Bcast(&a,1,MPI_INT,1,MPI_COMM_WORLD);
}
void MESHprocessor1()
{
int a;
int b;
int p;
{
p=1;
{
b=23;
}
printf("Hello from process %d\n",p);
}
a=b;
MPI_Bcast(&a,1,MPI_INT,1,MPI_COMM_WORLD);
}

typedef void (*MESHProcess)();
MESHProcess MESHprocesses[2];

void MESHinit()
{
MESHprocesses[0] = &MESHprocessor0;
MESHprocesses[1] = &MESHprocessor1;
}

```

```
int main(int argc, char * argv[])
{
MPI_Init(&argc,&argv);
MESH_Init(500,200);
MESHargc=argc;
MESHargv=argv;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD,&numberofprocesses);
if (MESHcheckstartup(2,numberofprocesses,argc,argv)==1)
{
MESHinit();
MESHsetUpCommErrHandler(MPI_COMM_WORLD);
MESHprocesses[myrank]();
}
MPI_Finalize();
return 0;
}

//Compilation time was 0 minutes and 0 seconds
//mpicc -o output output.c -lmesham
//mpirun -np 2 ./output
```

Listing 4.3: Generated C Example

4.8 Conclusion

The implementation of the concept of type-based parallelism and the Mesham language has required much work, resulting in a compiler. The compiler is correct in terms of that it will generate “correct” program code from correct Mesham source. The compiler is also complete in terms of it supporting all aspects of the Mesham language. However, at this stage it is still unclear how efficient the compiler is, most importantly efficiency in terms of the end (executable) result.

Chapter 5

Case Studies - Performance

5.1 Introduction

To evaluate the type-based approach of Mesham a number of case study examples have been performed. These experiments have involved writing specific code in Mesham and testing both performance and programability. In order to assess whether or not the type-based approach is useful a number of important questions need answering. These are whether or not Mesham can produce code with competing efficiency, whether type-based code is actually simpler than existing parallel language solutions and if programmers can write a variety of parallel codes in Mesham with varying complexity. It is the aim of both this and the next chapter to present the experiments and results which will address these questions.

The current chapter details a number of smaller experiments and looks at both the timing results and programmability of these. In order to test the performance of the codes, each was run on Durham University's Hamilton Cluster. The cluster typically comprises of machines with two dual core 2.2GHz Opteron Processors and 8GB of memory, connected by Myrinet, a high-speed communication protocol. For experimentation the Portland Group Compiler combined with OpenMPI (an implementation of the MPI standard) was used to compile both the Mesham generated C code and the control C code. All timing was measured using the machine wall clock time during execution and the results shown have been averaged over at least three separate runs. Tabular versions of the performance results shown graphically in this section are included in Appendix D.

5.2 Mandelbrot

As introduced in Section 2.5.4, the Mandelbrot set calculation is a popular parallel example used in numerous texts. Due to its embarrassingly parallel nature, it forms not only a popular, but also a simple, problem which generates visual results.

The Mesham source code of listing 5.1 is the mandelbrot example in its entirety. In line one the programmer defines variable *pnum* to be the number of processors, the way the code has been written changing this variable is all that is required to modify the number of processors. Lines 2 to 5 define attributes such as image size, quality and magnification. In line 6 the programmer defines *pixel* to be a new record type, containing the red, green and blue components of a pixel. In line 7 a two dimensional array of pixels is defined, allocated row major and horizontally partitioned into *pnum* blocks, each of which is evenly distributed amongst the processors. The *evendist* type will deal with any additional complexities, such as having to allocate blocks amongst an uneven distribution of data in the case of the image not dividing amongst the processors evenly. Line 8 defines variable *s* to be a two dimensional array of pixels allocated only on processor 0, at the end of execution the completed image will be held in this array.

Line 11 of the code starts a *par* loop, an iteration of this will execute on all (*pnum*) processors. In this loop, the element *mydata#p* is often used, this accesses the *p*th block of partitioned array *mydata*, i.e. the block of data allocated to the current process. The elements (*mydata#p.low*) and (*mydata#p.high*) found on line 14 will return the start and end index of the process *p*'s block respectively. The majority of the *par* loop is concerned with simply computing the Mandelbrot set and will not be considered here, it should be noted that this is a prime example of data parallelism, the code sent to each process in the *par* loop is identical, with each processor working on different data elements.

After each processor has finished working on their data, line 62 is where communication will occur. The statement *s:=mydata* will copy the values held in variable

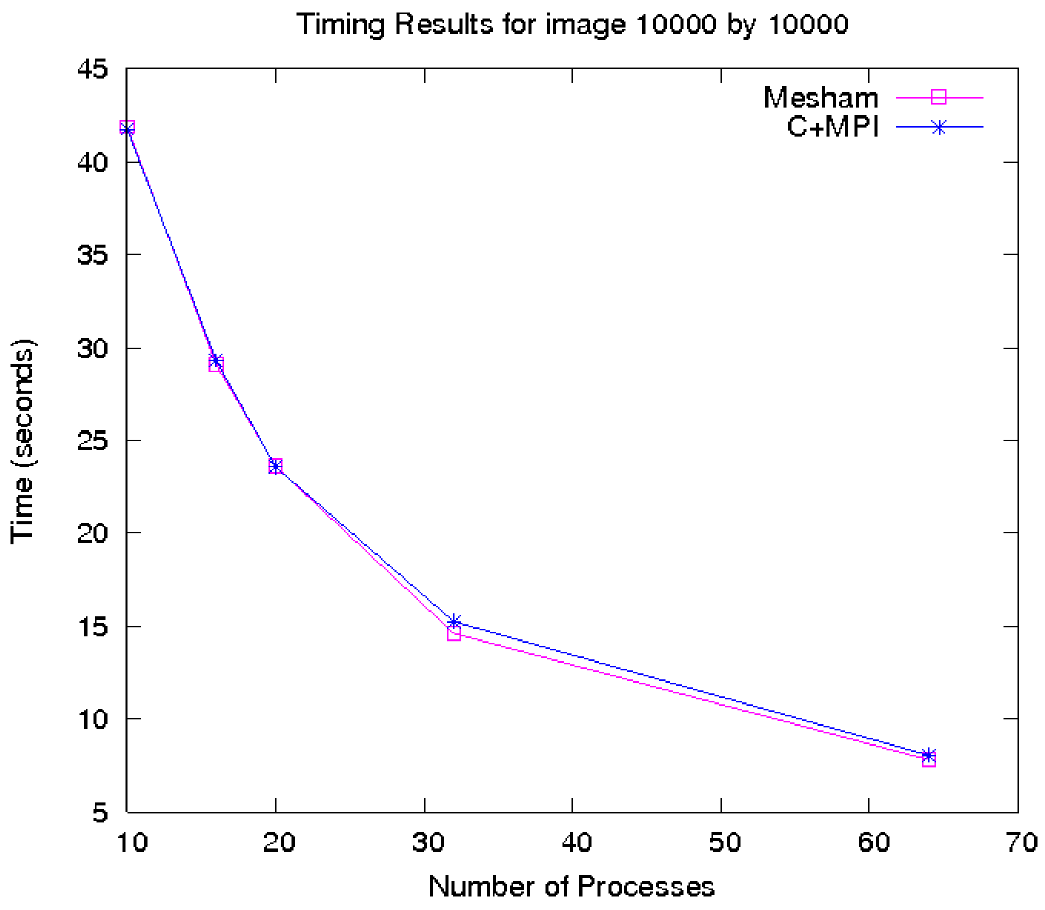

Figure 5.1: Mandelbrot performance test

mydata to that of array *s*. Because *mydata* is distributed amongst the processes and *s* is allocated on process 0 only, this will result in communication between all processes and process 0. The statement *proc 0* on line 63 will force the following block only to execute on process 0, which acts to write all the data in array *s* into a picture file for viewing.

Performance tests have been conducted against a similar parallel Mandelbrot program written in C-MPI. A snapshot of these results are shown in figure 5.1. The results obtained when running on 1, 2, 4 and 8 processors were identical between Mesham and C and hence were not shown in order to illustrate the small performance increase in the Mesham program once the number of processors becomes non-trivial. Due to the embarrassingly parallel nature of this example, the performance advantages of using Mesham only really start to stand out as the program runs on a large number of processors.

The reason for the small performance increase is that in Mesham parallelism is expressed in a much more high level manner via types. As the compiler knows the number of processors during compilation it can use this high level type information to make decisions statically which otherwise would need to be made dynamically adding additional overhead. For instance, in this example some additional overhead is required in the C-MPI Mandelbrot code to allow the programmer to run the code on any number of processors by simply changing one variable in the program code. In Mesham this is dealt with during compilation and as such this overhead is avoided.

Code listing C.1 is the C with MPI control code used to evaluate against. As can be seen, even for a very simple example the C code is none trivial.

```
var pnum:=4;      // number of processes to run this on
var hxres:=1000;
var hyres:=1000;
var magnify:=1;
var itermax:=1000;
var pixel:record["r",Int,"g",Int,"b",Int];
var mydata:array[pixel,hxres,hyres] :: allocated[row[] ::
    horizontal[pnum] :: single[evendist[]]];
var s:array[pixel,hxres,hyres] :: allocated[single[on[0]]];

var p;
par p from 0 to pnum - 1
{
   var hy;
   for hy from (mydata#p).low to (mydata#p).high
   {
      var hx;
      for hx from 1 to hxres
      {
         var cx:=((((hx % hxres) - 0.5) % magnify) * 3) - 0.7;
```

```
        var cy:=((((hy + (mydata#p).start) % hyres) - 0.5) %
            magnify) * 3;
        var x:Double;
        x:=0;
        var y:Double;
        y:=0;
        var iteration;
        var ts:=0;
        for iteration from 1 to itermax
        {
            var xx:=((x * x) - (y * y)) + cx;
            y:= ((2 * x) * y) + cy;
            x:=xx;
            if (((x * x) + (y * y)) > 100)
            {
                ts:=iteration;
                iteration:=999999;
            };
        };
        var red:=0;
        var green:=0;
        var blue:=0;
        if (iteration > 999998)
        {
            blue:=(ts * 10) + 100;
            red:=(ts * 3) + 50;
            green:=(ts * 3)+ 50;
            if (ts > 25)
            {
                blue:=0;
                red:=(ts * 10);
                green:=(ts * 5);
            };
```

```
                if (blue > 255) blue:=255;
                if (red > 255) red:=255;
                if (green > 255) green:=255;
            };
            (((mydata#p)#hy)#hx).r:=red;
            (((mydata#p)#hy)#hx).g:=green;
            (((mydata#p)#hy)#hx).b:=blue;
        };
    };
};
s:=mydata;
proc 0
{
    var fname:="picture.ppm";
    var fil:=openfile[fname,"w"];          // open file
    // generate picture file header
    writetofile[fil,"P6\\n# CREATOR: LOGS Program\\n"];
    writetofile[fil,1000];
    writetofile[fil," "];
    writetofile[fil,1000];
    writetofile[fil,"\\n255\\n"];
    // now write data into the file
    var j;
    for j from 0 to hyres - 1
    {
        var i;
        for i from 0 to hxres - 1
        {
            var f:=((s#j)#i).r;
            writechartofile[fil,f];
            f:=((s#j)#i).g;
            writechartofile[fil,f];
            f:=((s#j)#i).b;
            writechartofile[fil,f];
        };
    };
    closefile[fil];
};
```

Listing 5.1: Mesham Mandelbrot Code

5.3 NAS-IS Benchmark

The NAS Parallel Benchmarks (NPB), which were reviewed in Section 2.5.3, act as an objective, official, evaluation of Mesham. A version of Integer Sort (IS) has been written in Mesham, and then fine tuned for performance via testing and modifying the code.

The code in listing C.2 details the Mesham IS implementation, which not only completes the integer sort as per the NPB specification [Baily1994], but also honours the specification's verification and number generation rules too. Using abstractions such as data structures, implemented using the *referencerecord* type (line 6), helped to simplify the process of writing the 380 lines of code. It is appreciated however that having existing code to understand did help with the implementation of this benchmark. NASA's official C code [Saphir1996] is over 1000 lines long and deals heavily with low level details such as pointers and sharing the same block of memory in order to maximise performance. The Mesham code does not require the programmer to worry about these low level details, instead the extra information provided by the programmer's use of types allows the compiler to perform these optimisations.

The first benchmark to be done was using class B (33 million numbers.) NASA's version was compared directly against the one written in Mesham, the results of which can be seen in figure 5.2. From this graph it can be seen that, up until 32 processors, the performance of both benchmarks is comparable, although the NAS code is slightly faster on one processor whilst the Mesham code is slightly faster on 4 and 8 processors. However, after the optimum number of processors (around 22) the worsening runtimes start to diverge, with the Mesham code approximately 12%

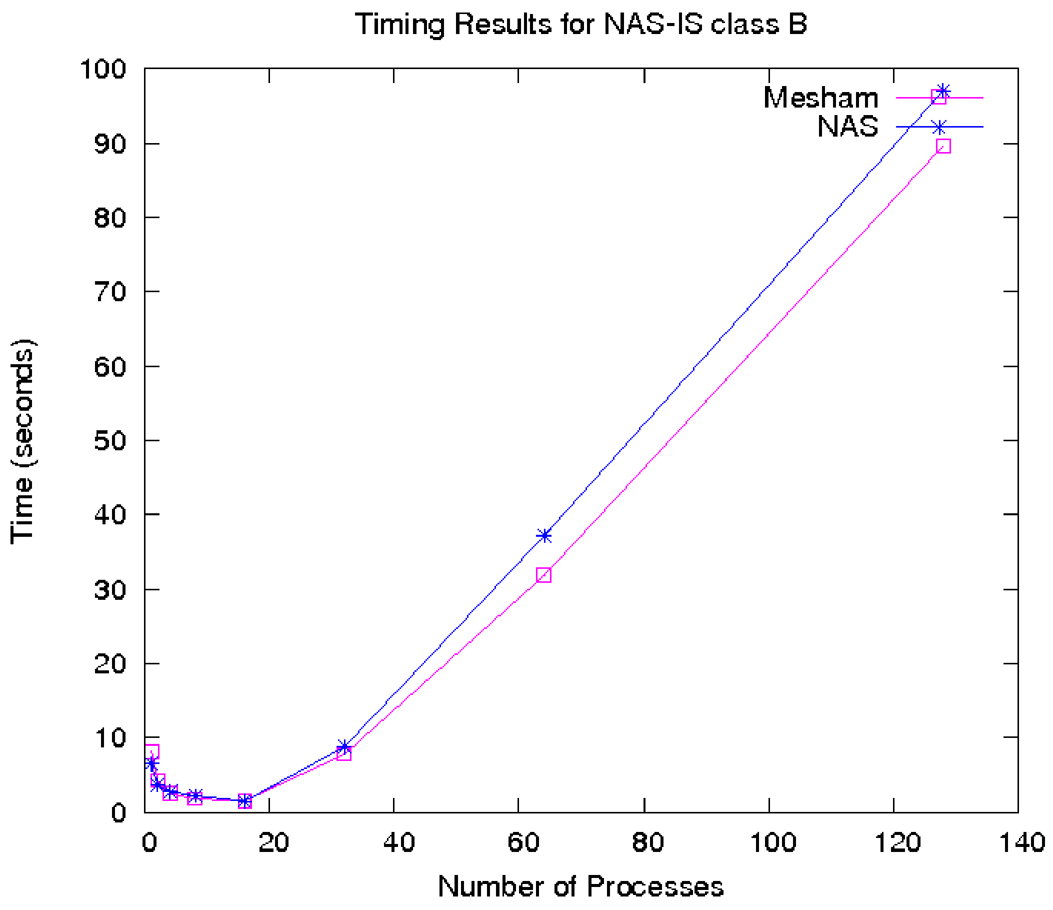

Figure 5.2: NAS-IS benchmark, Class B

faster when using 64 processors and 8% faster on 128 processors. This difference is due to extra information provided to the Mesham compiler, and its ability to optimise knowing, amongst other things, the number of processors which has been set statically in the code.

One such reason in this case for this competing, and in some cases superior efficiency, is that as discussed in Section 4.7, the Mesham compiler will generate code for each processor and as such can tailor a processor's code specifically. When writing parallel code in a language such as C-MPI it would often be far too inconvenient to write code for each processor. Instead the programmer will often write their parallel code following a general SPMD style which will mean that firstly processors will all receive pretty much the same code, although not all of it is relevent, and secondly it can be more difficult for the programmer to statically set specific values to specific processors. The Mesham approach is to take all of the type information and, with this high level description, tailor each processor's code as much as possible to firstly try and complete as many operations during compilation and secondly ensure each processor's code is completely relevant to it.

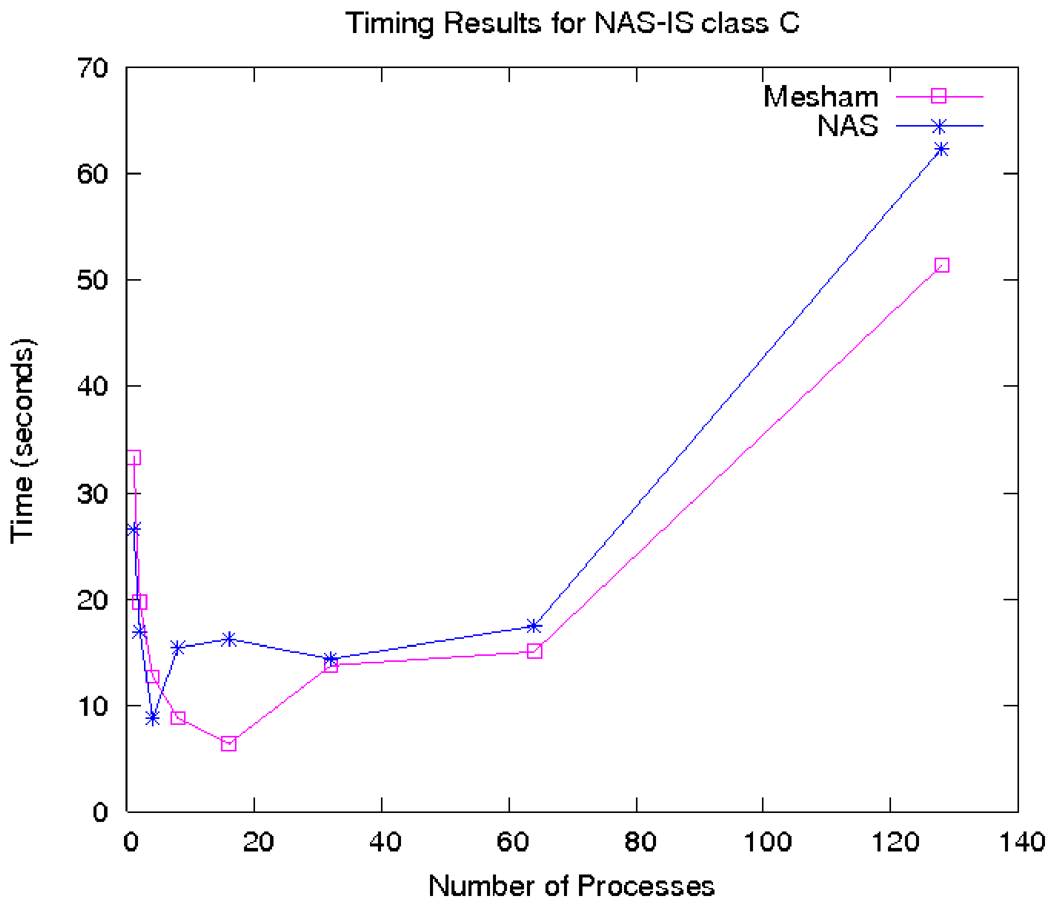

Figure 5.3: NAS-IS benchmark, Class C

Figure 5.3 details the results of the IS benchmark when using class C (134 million numbers.) Again, the performance is comparable, although there is some instability in the NAS benchmark (which was rerun to ensure the absence of anomalies) over 8 and 16 processors whilst the Mesham code produces a smooth curve. It is interesting that, unlike with class B, after the optimum number of processors the performance decrease between 32 and 64 processors is only very slight for both implementations. As with class B, after the optimum point Mesham's performance is more favourable than NASA's, with Mesham IS around 17% faster than NAS-IS on 128 processors.

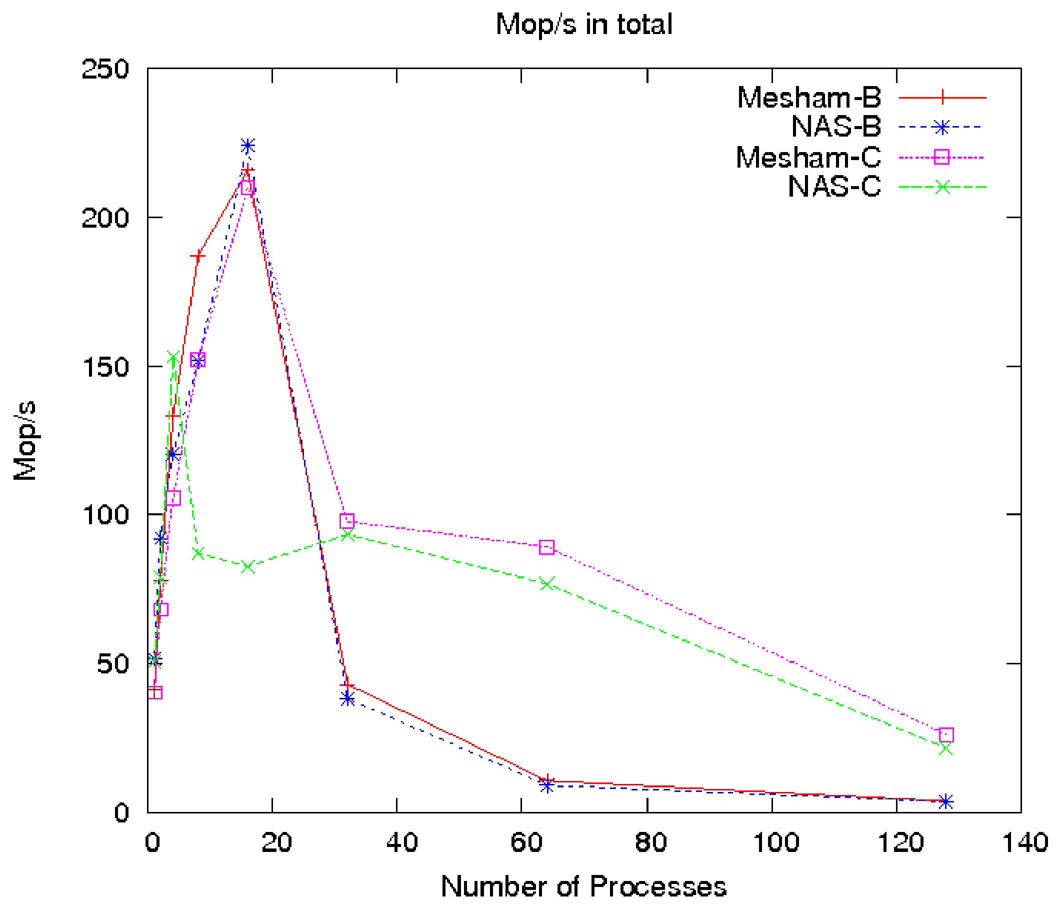

Figure 5.4: NAS-IS benchmark total Mop/s

Figure 5.4 illustrates the total Million Operations per Second (Mop/s) against number of processes in the parallel system. Interestingly Mesham class B, NAS class B and Mesham class C all seem very similar up until the optimum number of processors. It can be deduced that for class C over 8 and 16 processors the Mesham code obtains higher Mop/s than its C counterpart. Interestingly after the optimum point each class of experiment curves seem very similar, with class C maintaining a higher Mop/s rate rather than class B. In figure 5.5, showing Mop/s per process, it can be seen that initially both classes start at around the same figure (45 Mop/s per process), as the number of processors is increased Mesham class B, NAS class B and Mesham class C all seem very similar up until the optimum point. After this point class C exhibits considerably more Mop/s per process than class B which is to be expected as the processors are better utilised.

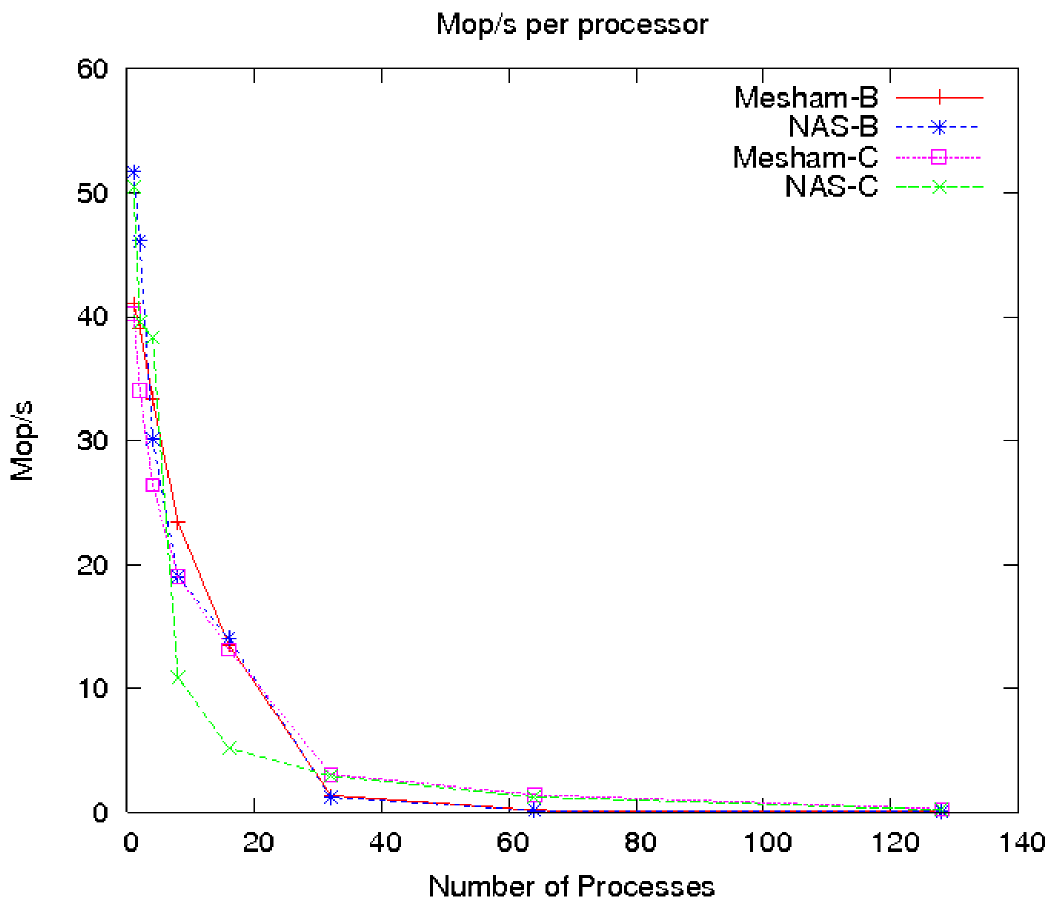

Figure 5.5: NAS-IS benchmark Mop/s per process

One very intersting observation from figures 5.2 and 5.3, is that for 128 and 64 processors, class C (134 million numbers) is faster than class B (33 million numbers), by around 39% with 128 processors. Hence by adding extra data, it has actually made the benchmark run faster on these number of processors, which is very counter intuitive. One explanation for this might be found when comparing the computation saving against the communication cost of adding an extra processor. For class B, due to the smaller data size, adding an extra processor will have a far smaller computation saving than it will for class C, hence performance decrease after the optimum point will be more severe the smaller the data size. Considering the fact that a large majority of computation cost lies in setting up the link, synchronising the processors and sending the message header, actually sending on average four times the data in this case probably does not have a huge performance hit with large numbers of processors - the fact that each processor has computationally four times the amount of data to solve will be more important. This is supported by that fact that class C, on average, performs worse compared with class B with a small number of processors and as the number of processors is increased the performance gap decreases. The reason for this could be that for a small number of processors the extra data is important compared with the overhead, however when dealing with a large number of processors the overhead will be much greater with the same amount of data and as such the data size will be of far less relavence.

5.4 Fast Fourier Transformation

"Parallelised 2D Fast Fourier Transformation (FFT) MPI code is much more complicated than the sequential code. For example, array transposition is simple on one processor but more sophisticated if the array is partitioned and distributed over multiple processors. Direct MPI programming requires the end programmer to handle every detail of FFT's parallelization including writing the appropriate communication commands, synchronisations, and correct index expressions that delimit the range of every partitioned array slice. A small change of how the array is partitioned or distributed may result in code rewriting. Type-based parallelization, however, can relieve the end programmer from writing details of parallelization if the details can be derived from the type information." [Brown2008]

5.4.1 FFT code in Mesham

The Mesham code for FFT is shown in listing 5.2.

```
var complex : record[r,Float, i,Float];
var n:=512;
var p:=5;
var k;

var S : array[complex,n,n]::allocated[row[]::single[0]];
var A : array[complex,n,n]::allocated[row[]::horizontal[p]::
    single[evendist[]]];
var B : array[complex,n,n]::allocated[col[]::horizontal[p]::
    single[evendist[]]];
var C : array[complex,n,n]::allocated[row[]::horizontal[p]::
    single[evendist[]]]::share[B];
var i : Int::allocated[multiple[]];

var sins : array[complex,n/2]::allocated[row[]::multiple[]];
ComputeSin[sins];

proc 0 {readfile[S, "image.dat"]};
A:=S;
par k from 0 to p-1
    for i from A#k.low to A#k.high  FFT[A#k#i, sins];
B:=A;
par k from 0 to p-1
    for i from C#k.low to C#k.high  FFT[C#k#i, sins];
S:=C;
proc 0 {writefile[S, "image.dat"]};
```

Listing 5.2: FFT code in Mesham as from [Brown2008]

The code first declares the record type for complex numbers, the size n of the input array, the number p of processes and the process ID index variable k. Array S is allocated row major on process 0, containing the source data. Row-major array A is partitioned p times horizontally and evenly distributed to all processes by broadcasting, storing the source data after initial broadcasting. Array B is similar to A but declared to be column-major. Assignment from A to B essentially transposes A and shuffles the blocks of array A across processes. This allows each process to perform linear FFT on the other dimension locally. Array C is row-major but partitioned horizontally and shared with B for allocation. The type *share[B]* provides a different typing view on the same data. Performing row-wise FFT on C is the same as performing column-wise FFT on B. The multiple variable i is used as a loop index on every process. The array *sins* stores the pre-calculated constant sinusoid parameters needed in FFT for both dimensions, the function call *ComputeSin* initialises this *sins* array. Process 0 first reads data from a file into the source array and the assignment *A:=S* dynamically distributes this source data. Every process *k* performs FFT on every row of the *k*-th slice of *A*. The *.low* and *.high* in the example provide the lower and upper bounds of a block, and are calculable in compile time. The assignment *B:=A* performs the entire parallel transposition with multiple asynchronous communications and an ending synchronization. FFT is performed again on the transposed array *C* on every process. The result array *C* is gathered into the source array, which is then written back into the file. Sequential functions *ComputeSin* and *FFT* are omitted due to a lack of space.

5.4.2 The generated FFT code

The types in the Mesham code have provided enough information for the compiler to determine statically the size and location of each block during the compilation process, resulting in more optimisation performed during compilation and hence increased efficiency. Listing C.3 shows part of the generated code (for data transposition), which is not intended for human reading.

The code rearranges the process's data such that the data sent to each processes is continuous (and convenient for message passing). It then computes the index and displacements of each processes data before the data is ready to be sent. In order to increase efficiency the process will send out a chunk of data to every other process asynchronously, so that it does not need to wait for other processes. The last part of the transposition code deals with receiving data from each other process, again for efficiency reasons this is done asynchronously, with the process registering the receive requests and then allowing them to complete in any order. Asynchronously moving data in this way comparatively performs much better than synchronous communication as the number of processes is increased.

Very commonly in HPC the programmer, due to a lack of knowledge or wishing to keep the code simple, will take shortcuts and pick an easier yet less efficient communication function. By abstracting the physical communication away from the programmer, system programmers who are good at optimization can select the most efficient form of communication to be automatically generated by the compiler.

In the conventional approach of writing high performance code, changing details (for instance the form of communication) would require major recoding. However, following the type-based approach this change is all handled automatically by the partition types and requires no recoding, resulting in a more maintainable code. It can be seen that the approach of adding type information into the declared type results in a much simpler, easier-to-maintain and efficient program which is communication and computationally safe. As Mesham relieves the programmer from many of the low-level parallelization details, it enables programmers to obtain complex, more optimized code which is normally difficult to achieve.

5.4.3 Performance

Two groups of experiments of different problem sizes have been carried out on the Hamilton cluster of Durham University. In the first group of experiments, different numbers of processes with problem size of 128MB (4096 by 4096) were tested on the Mesham FFT code in listing 5.2, the Fastest Fourier Transformation in the West (FFTW [Frigo1998-2]) library version two and the C-MPI FFT book example of Pacheco [Pacheco1996]. Figure 5.6 shows the speedup results of testing the three different FFT programs on 128MB data.

As explained in Section 2.5.2, FFTW chooses a sub-algorithm according to the problem size and the number of processors. Mesham contains the end programmer's code that provides enough high-level information and the carefully pre-optimized communication code designed by system programmers and generated by the compiler. In the textbook code, both computation and communication parts are handwritten and "optimized up to the convenience of the end programmer".

Initially FFTW is more efficient than the other two codes on one process due to its algorithm selection. With more than one process, the Mesham code is faster than FFTW, and this trend continues as the number of processes grows. Eventually, under the problem size, the importance of efficient communication (as detailed in Section 5.4.2) outweighs efficient computation – making Mesham more efficient than

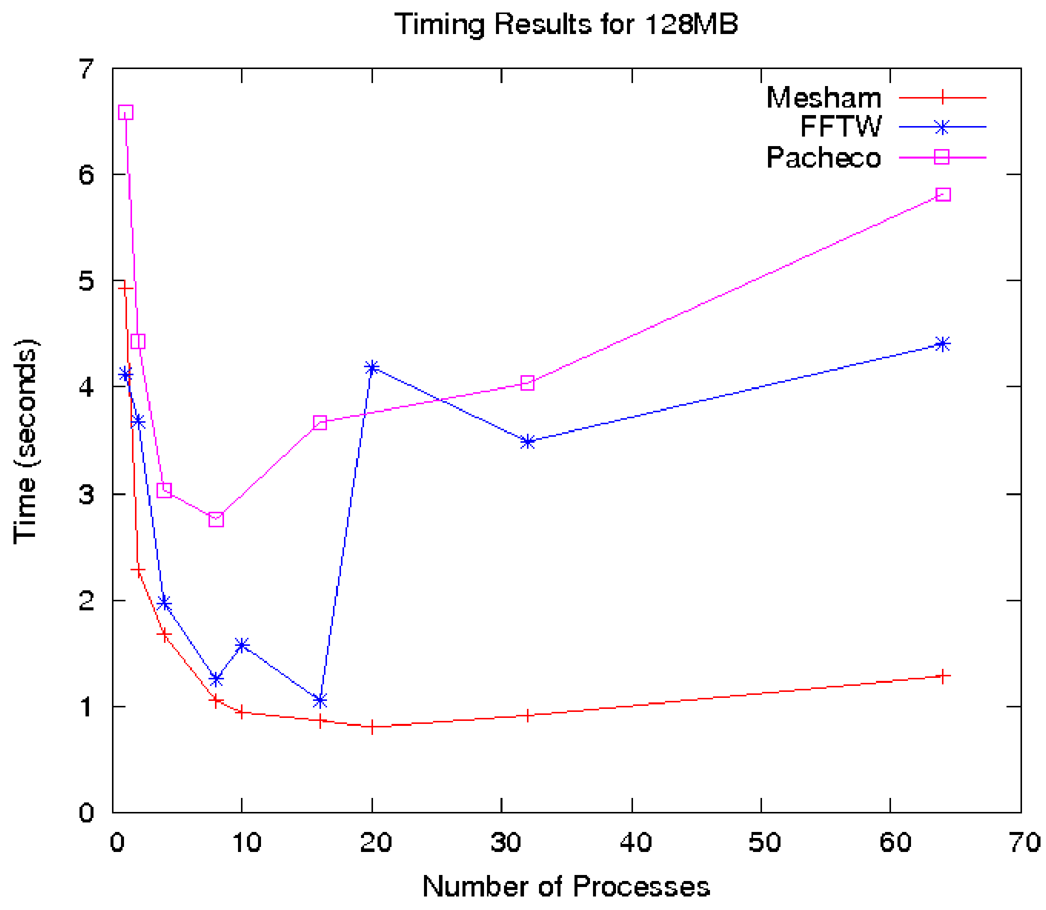

Figure 5.6: Performance of FFT on 128MB Data

FFTW and the textbook code.

When run on 10 and 20 processes the FFTW speedups drop and the Pacheco code has no results. The simplistic Pacheco textbook code only works with process numbers 1,2,4,8,... and cannot utilise an arbitrary number of processors. In FFTW, the library dynamically computes the data size on each process. When the partition is uneven each process must inform, via communication, the root process of its size for the initial Scatter and the last Gather. The speedup spikes reveal some instability in the FFTW library's implementation for an uneven distribution of data. Uneven distribution of data is automatically handled by the Mesham compiler, specifically the partition and distribution types as detailed in Section 3.3, yielding a more smooth curve of performance.

As explained in Section 5.4.2, one reason for this increased performance is the choice of communication guided by the high level type information. Knowing important aspects, such as the size of data, number of blocks and where each block should belong means that the compiler can do much of the calculation work during compilation rather than it being done dynamically. Additionally, any issues raised during compilation (such as an uneven distribution of data) can be dealt at that point, for instance by issuing specialist algorithms, rather than relying on general less efficient solutions. As also mentioned due to the high level nature of Mesham, the programmer is completely abstracted away from the mechanics of communication. In this case, a significant performance gain was found by the partition types generating specialised transposition communication code. This specialised code allows for the communications to be carried out asynchronously which, although more efficient is also more complex to write and depends upon knowing numerous code attributes. It is not practical for either the FFTW or Pacheco implementer to consider this level of complexity in their handwritten code, which would limit the number of processors and size of data anyway. Because this is all carried out during Mesham's compilation, there is no such limitation nore does the programmer experience any increased complexity.

The performance result on 128MB data is re enforced by a larger data set with 2GB image (16384 by 16384) (see Figure 5.7). With more data, communication bandwidth and computation outweigh communication latency and other overheads. Note that the textbook code cannot handle the declaration of such large arrays and are hence not tested.

For 2GB data on one process (sequentially), the FFTW code (which selects different algorithm) is about twice as fast as the Mesham code, but when the number of processes grows, the Mesham code then demonstrates a convincing lead in speedups.

5.5 Conclusions

As has been seen throughout this chapter, a variety of programs have been written in Mesham. This has been to assess firstly performance and secondly programmability. It has been seen that by using the type-based approach, performance during

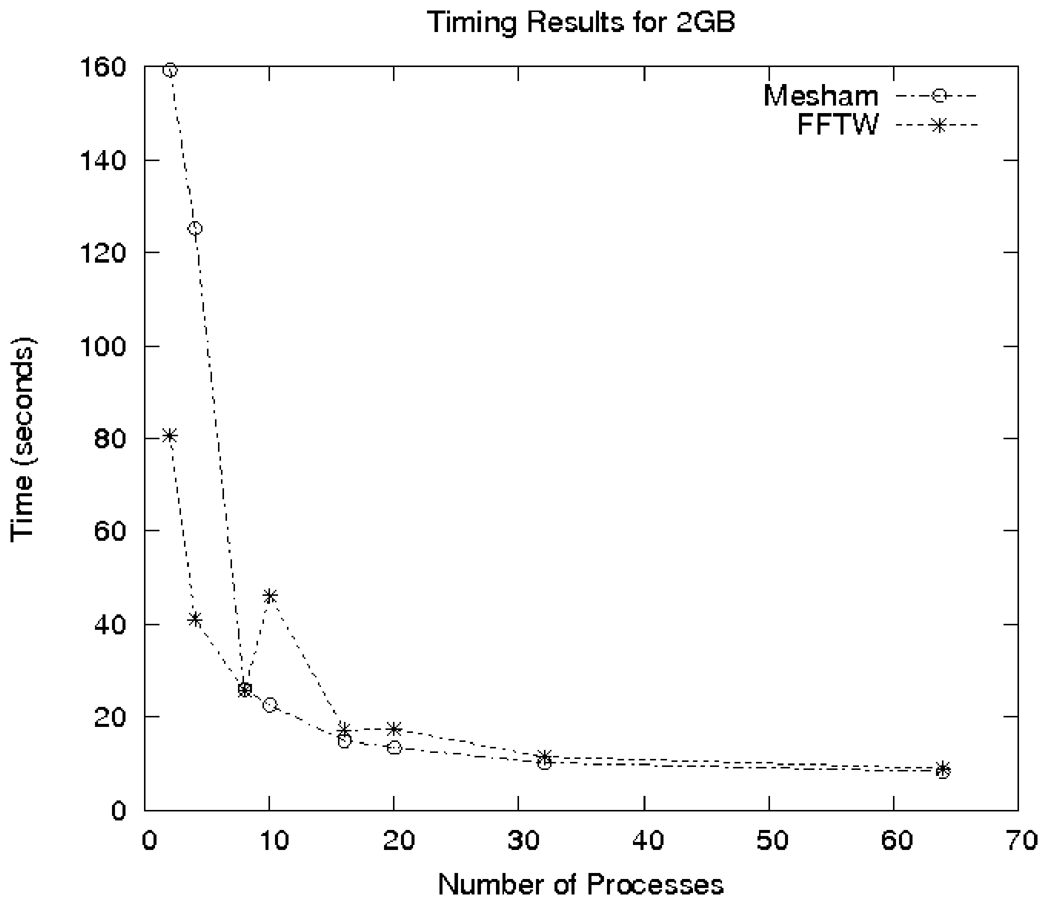

Figure 5.7: Performance of FFT on 2GB Data FFT

benchmarking is comparable to existing codes and languages. It can also be seen that by writing code in this type-based manner has simplified the programming task and allows the coder to concentrate on the more high-level aspects. Each of the applications considered here are very different from each other, demonstrating that the language and proposed type approach can be applied successfully to many different scenarios.

Chapter 6

Case Studies - Gadget-2

6.1 Introduction

The parallel cosmological simulation package, Gadget-2, was introduced in Section 2.5.1. In order to evaluate factors, such as the simplicity and flexibility, of this type based approach, aspects of Gadget-2 were ported into Mesham. By looking at a complex application, it can be seen that this proposed type-based approach is not just limited to parallel computing. Some of the simplifications obtained via types in Gadget-2 have been on sequential aspects of the code.

6.2 Extension Types

The design of the type system means that it is very simple to add additional types and functionality to Mesham. An example of such extensions are those added to the language in order to support the porting of the physics simulation package Gadget-2. These types are used to greatly simplify the ported Gadget-2 code and illustrate nicely how having these types available greatly simplifies the programmer's job.

<u>Particle</u>

Syntax

Particle

Semantics

This type represents a particle (element) type. A number of attributes are possessed by a *particle*, these are listed in table 6.1.

Member	Dimensions	Type
id	1	Int
type	1	Int
position	3	Double
velocity	3	Double
gravityaccceleration	3	Double
mass	1	Double
potential	1	Double
oldacc	1	Double
gravitycost	1	Double
tiend	1	Int
tibegin	1	Int

Table 6.1: Particle element type members

Example

```
var a:Particle;
a.id:=0;
(a.position#0):=4.3;
(a.position#1):=1.3;
(a.position#2):=9.3;
```

Will create particle *a*, set the particle's *id* to be 0 and specify a position.

Gadget Parameter File

Syntax

gadgetparamfile[filename]

Semantics

This type will open a gadget parameter file and allow the programmer to read (and write) to it, automatically formatting the IO, abstracting these low level details from the programmer. The parameter file follows a very specific format, which this type guarantees to maintain. There are many attributes of this type, which can be accessed via dot (.).

Example

```
var a:gadgetparamfile["galaxy.param"];
var outd:=a.OutputDir;
print[outd,"\n"];
var b:gadgetparamfile["newgalaxy.param"];
b.OutputDir:=outd;
```

In this example the code reads the attribute *outputdir* from *galaxy.param*, displays it and then opens another parameter file *newgalaxy.param*, creating the file if it does not already exist, and writes this attribute data in. As can be seen from the example, the Mesham programmer need not worry about the physical aspects such as file writing and closing.

Snapshot

Syntax

snapshot[filename]

Semantics

Gadget-2 uses snapshot files to record both the current state of the simulation and also to start from an initial state. There are a number of different parts of the file, including the header (with information about the snapshot) and then all the data about each particle, split up into different blocks which must be carefully formatted. This complexity is all taken care of via the *snapshot* type, the programmer is abstracted away from the low-level details and can read or write an array of particles and also access the header attributes from a high level.

Example

```
var u:snapshot[s] :: allocated[single[on[0]]];
var a:array[Particle,NumPart] :: allocated[multiple[]];
u.numberofparticles:=totparts;
u.massofparticles:=massofparticles;
u.totalnumberofparticles:=totparts;
u.time:=All.Time;
u.redshift:=0;
u.files:=1;
u.boxsize:=All.BoxSize;
u.omega0:=All.Omega0;
u.omegalambda:=All.OmegaLambda;
u.hubble:=All.Hubble;
u.entropy:=0;

u:=a;
```

Taken directly from the Gadget-2 Mesham code, this code will write a snapshot file allocated on process 0, setting the header attributes and then the assignment *u:=a* actually copies the particle data into the snapshot file.

Snapshot2

Syntax

snapshot2[filename]

Semantics

Identical to *snapshot*, except the file is in a different format which some users of Gadget-2 prefer.

Example

```
var a:snapshot["format1"];
var b:snapshot2["format2"];
b:=a;
```

The example converts a snapshot file of format1 to format2 (snapshot format to snapshot 2 format.) Incidentally, in Gadget-2 a file conversion tool (which does not exist) would be a complex undertaking as it must reformat the data. Using the type-based approach the Mesham programmer need not worry about these details and can easily achieve the end result.

Snapshot HDF5

Syntax

snapshotHDF5[filename]

Semantics

Identical to snapshot, except the data is written (and read) in HDF5 format.

PHCurve

Syntax

PHcurve[name]

Semantics

This will automatically construct a PH curve from a spaceshape, *name*, ordering each member by its Peano Hilbert key. Apart from accessing elements on the curve (via #), there is a *.totalcells* attribute which returns the total number of Peano Hilbert keys available (by default this is 2^{20} for each dimension, so 60 bit for 3 dimensions.) For each member there are the attributes *.key* (to get the Peano Hilbert key) and *.orderadded* to return the order the member was added to the curve.

Example

assuming s is a shapespace, with items in it

```
var peano:PHCurve[s];
var keysize:=peano.totalcells;
var itemonekey:=(peano#0).key;
var itemoneorderadd:=(peano#0).orderadded;
var itemone:=(peano#0);
```

In this example the variable *peano* is a Peano Hilbert curve acting on the shape space *s*. The variable *keysize* is equal to the number of PH keys available, *itemonekey* is the PH key of the first element on the curve with *itemoneorderadd* the order in which this element was added to the curve and *itemone* set to the data held at this point on the curve.

6.3 Porting I/O

I/O is a major part of Gadget-2, with the simulation package starting from a configuration file and setting up its galaxy from a file known as the Initial Condition. Periodically, the simulation will write out its data to a file for backup or analysis. This is further complicated by the fact that Gadget-2 may use one of three formats for the data files. Gadget-2 contains around 2500 lines of code dedicated to these tasks, whereas by using the extension types of Mesham the total count is 153 lines.

```
var u:snapshotHDF["galaxy.param"] :: allocated[single[on[0]]];

u.numberofparticles:=totparts;
u.massofparticles:=massofparticles;
u.totalnumberofparticles:=totparts;
```

```
6  u.time:=All.Time;
7  u.redshift:=0;
8  u.files:=1;
9  u.boxsize:=All.BoxSize;
10 u.omega0:=All.Omega0;
11 u.omegalambda:=All.OmegaLambda;
12 u.hubble:=All.Hubble;
13 u.entropy:=0;
14
15 u:=P;
16 print["Done with writing snapshot file ",s,"\n"];
```

Listing 6.1: Creating a snapshot in Mesham

Code listing 6.1 contains the entire code required to create a snapshot of the current state of the simulation. In line 1, the *snapshotHDF* type is used to signify that variable u represents a snapshot file in HDF5 format, combined with the allocation type this is located on processor 0 only. Lines 3 to 11 write current simulation information to attributes of variable u and hence the snapshot file. In Line 13 the data in variable P, which is a representation of each particle in the simulation, is collected on processor 0 and written into the snapshot file. As can be seen, the code is simple and concise, if the programmer wished to change an aspect such as the format of snapshot file then this can be easily achieved. From this code it can be seen that the type-based approach is not just limited to parallel programming, from the mainly sequential IO aspects of Gadget-2 it has been used to simplify the code.

6.4 Domain Decomposition

The simulation performed by Gadget goes in steps. In each step particles are simulated to collide with each other, after this the code must calculate which, following the action, now belong on each processor and exchange these particles amongst the processors. In order to identify the location of particles in 3D space, a space filling curve known as the Peano Hilbert curve is used. The assumption is made that each particle will sit somewhere along this curve and the order along the curve, or Peano key as it is known, of each particle can be used to make sense of its location. Figure 6.1 illustrates example 2D and 3D Peano Hilbert curves. Mathematically, a 3D PH curve is used in Gadget-2. Gadget-2 dedicates around 400 lines to computing, ordering and finding Peano keys for each particle. By using the type based approach of Mesham, this has been abstracted away into types.

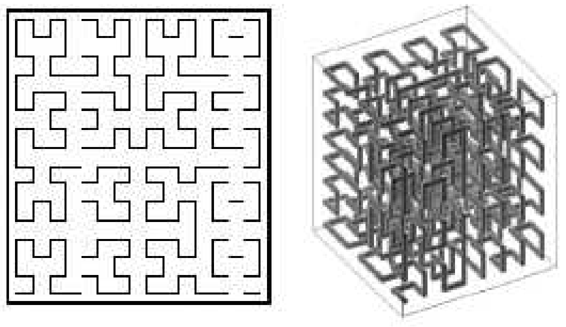

Figure 6.1: Sample 2D and 3D Peano Hilbert Curves taken from [Springel2006]

```
var space:SpaceShape[Particle,(min#0),(max#0),(min#1),(max#1),(
    min#2),(max#2)] :: allocated[multiple[]];
var i;
for i from 0 to NumPart - 1
{
        var x:=((P#i).position)#0;
        var y:=((P#i).position)#1;
        var z:=((P#i).position)#2;
        (((space#x)#y)#z):=P#i;
};

var peano:PHCurve[space] :: allocated[multiple[]];
for i from 0 to NumPart - 1
{
        var pkey:=(peano#i).key;
        var theparticle:=(peano#i);
        print["The ",i,"th particle on the curve has PH key ",
            pkey," and has ID ",theparticle.id,"\n"];
17  };
```

Listing 6.2: Peano Hilbert Curves in Mesham

Code listing 6.2 details an example use of how Peano Hilbert curves are created and then used in the Mesham Gadget-2 port. Firstly a space shape is created using the *SpaceShape* type. This is an abstract collection type, a more complex cousin of the array, which allows the programmer to store data at specific abstract locations - in this case particles in 3D space (as described by *min#d max#d* where *d* is the dimension.) The loop of lines 3 to 9 places particles in the array *P* into the spaceshape, *space*. Once this has been completed, the variable *peano* is defined with the *PHCurve* type, which informs the compiler that this is a Peano Hilbert curve. Passing the variable *space* to the *PHCurve* type signifies that the curve will be created in respect of that spaceshape. The loop in lines 12 - 17 will loop through each element on the curve (in curve order), retrieving each's Peano Hilbert key and the particle data.

By abstracting away all the concrete details such as key caching, reordering and curve mathematics, the programmer is free to concentrate on their actual code. Comparing this with code listing C.4, which is a small part of Gadget-2's Peano Hilbert key finding function, one can see that the C programmer has had to deal with all the nitty gritty details. Even the mathematical details of the PH curve must be considered and are entered into the arrays. From reading the C code, it is actually very difficult to figure out what is happening, operators such as bit shift just add further confusion and it is easy to get bogged down in these details and for bugs to appear rather than take a high level view of the whole parallel picture. The number of dimensions is also hard coded into this code, whereas the Mesham programmer can have any number of dimensions (which is decided by the spaceshape provided as an argument.)

The next important aspect of domain decomposition is to create the Barnes Hut tree. Where *d* is the number of dimensions, one can think of this as a tree where at each node is a square split into *2 * d* sections. The children of this node are these sections, which are again split into *2 * d* sections which are their children and on it goes. The result is a large rectangle (or space in this case) at the root, and then as progress is made down the tree the area of interest becomes smaller and smaller until the desired resolution is reached. The tree need not go as deep in all areas - if a section has no data in it then it can stop there, whereas if there are many particles, in Gadget's case, the code will keep going until a threshold number is reached. Of most use here is that it affords an overview of the space and allows for individual particles to be placed into each area and then very easily these can be assigned to specific processors with a good load balancing.

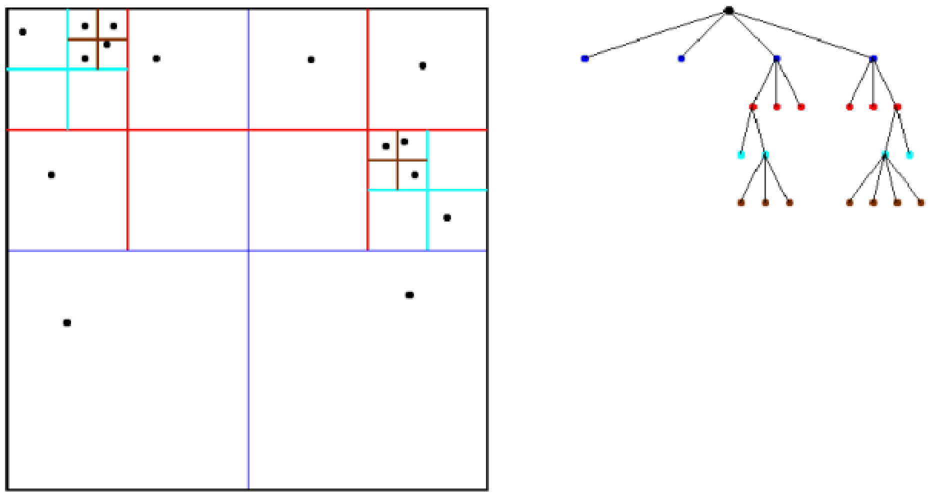

Figure 6.2: BHTree Example taken from [Springel2006]

Figure 6.2 shows an example of how this process will progress. The shape is first split (in blue) into *2 * d* sections (4). As can be seen, each section has a particle and as such is added as a child of the root. However, the lower two sections have no further particles and as such work stops on them. The top two sections split into 4 segments, both have three subsections with particles and as such are added as children (the empty subsections are not added to the tree.) As the algorithm progresses sections with multiple particles are further partitioned and added to the tree until each node in the tree contains only one particle. In this simple example, the resolution threshold is one particle - in reality this would be too time consuming to work through and as already explained Gadget-2 selects a higher number for this.

Concretely, each child of the BH tree has a number of attributes associated with it which are required when analysing and working on the tree. In Mesham, the programmer can use the *referencerecord* type to define records and link them together in a tree manner to form this tree. By using this abstraction it allows the programmer to easily work with these nodes, not worry about details such as memory usage and node attributes can be added, removed or modified with little worry.

```
var BHTree:referencerecord["children",array[BHTree,8],"size",
    Long,"count",Int,"startkey",Long,"pstart"];
var BHChildren:referencerecord["next",BHChildren,"size",Long,"
    count",Int,"startkey",Long,"pstart",Int];
var childroot:BHChildren :: allocated[];
.......
function void buildBHtree[var space, var totparticles, var
    globroot]
{
.......
var root:BHTree :: allocated[multiple[]];
root.count:=totparticles;
root.size:=peano.totalcells;
root.pstart:=0;
root.startkey:=0;
BHtreeCons[space,root,0,peano];
.......
}

function void BHtreeCons[var space, var root, var startk, var
    peano]
{
root:BHTree :: allocated[multiple[]];
startk:Long :: allocated[multiple[]];
space :SpaceShape[Particle] :: allocated[multiple[]];
```

```
peano:PHCurve[space] :: allocated[multiple[]];
if (root.size > 7)
{
        var i;
        for i from 0 to 7
        {
                var daughter:BHTree :: allocated[multiple[]];
                daughter.size:=(root.size) % 8;
                daughter.startkey:=startk + (i * (daughter.size)
                    );
                daughter.pstart:=root.pstart;
                ((root.children)#i):=daughter;
        };

        var j;
        for j from (root.pstart) to ((root.pstart) + (root.count
            )) - 1
        {
                var bin:=((peano#j).key - startk) % ((root.size)
                    % 8);
                var thenode:=(root.children)#bin;
                if (thenode.count == 0) thenode.pstart:=j;
                thenode.count:=thenode.count + 1;
        };

        for j from 0 to 7
        {
                var thenode:=(root.children)#j;
                if (thenode.count > (60000 % (20 * processes[] *
                    processes[])))
                {
                        BHtreeCons[space,thenode,thenode.
                            startkey,peano];
50                     } else {
51                               var newchild:BHChildren::allocated[
                                     multiple[]];
52                               newchild.pstart:=thenode.pstart;
53                               newchild.size:=thenode.size;
54                               newchild.count:=thenode.count;
55                               newchild.startkey:=thenode.startkey;
56                               newchild.next:=childroot;
57                               childroot:=newchild;
58                     };
59           };
60 };
61 };
```

Listing 6.3: Building the BHTree in Mesham

The Mesham code of listing 6.3 contains a sample of that used to build the BH Tree. The aim of the code is two fold, firstly to build the tree and secondly, for performance reasons, to link each of the end nodes together in a linked list to avoid traversing the tree every time. Lines 1 and 2 define the *BHTree* and *BHChildren* reference records respectively, instances of *BHTree* are used to build the tree, whilst instances of *BHChildren* to create the end node linked list. On line 3, the variable *childroot* is the head of the end node linked list, and will be referred to later in the code. As the domain decomposition progresses, the function *buildBHtree* is called where in this listing some irrelevant code has been omitted. Lines 8 to 12 create the root node of the BHTree, *count* is set to be the total number of particles and *size* is set to be the maximum number of particles possible (the total number of PH keys available.) The recursive function *BHtreeCons* is then called with this node.

The function *BHtreeCons* takes a node, will split it up into *2 * d* subnodes (8 in 3D space), assign particles to each of these subnodes via their PH key and if required will then call itself on each subnode to further divide them. Lines 25 to 33 create the daughter nodes, will set the size of each so they all share the same number of possible PH keys and then will specify the PH key each will start from (node 0 will start from PH key 0, node 1 from node 0 startkey + node 0 size etc....) Lastly each daughter node is linked in as a child of the node. Lines 36 to 42 will start off from the node's first particle number *root.pstart* and will go through each of its particles to determine which daughter node it belongs to. On line 38 the programmer is accessing the PH key of each particle, making this relative to the node and then dividing it by the number of daughters (8) in order to determine which daughter node the particle belongs to. This daughter node's information is then updated in lines 40 and 41 to reflect this. Once all daughter nodes have been populated, the code loops through them individually on lines 44 to 60. On line 47 the code determines the BH tree node resolution threshold number is *60000 / (20 * processes * processes)*, if the daughter contains more particles than this then the function is recalled on that node to further split it. If there are less than the threshold number of particles, then the daughter is an end node, a node of type *BHChildren* is created and populated in lines 51 to 55, and in lines 56 and 57 this is added to the head of the end node linked list.

Comparing this with listing C.5, which is part of the native Gadget-2 code for building the BH tree, one can see that the Mesham code is a lot more abstract. The C code is constructed such that the programmer is simply accessing elements of an array, *TopNodes*, which is not only less obvious but also specifies a limit to the number of nodes depending on the array size; unlike the Mesham version which by the very nature of a tree can keep growing whilst memory allows. Another disadvantage of this approach is that memory is allocated in the C version regardless whether it is used or not. As can be seen, there are many globals being accessed by this C function which, due to the possibility of complicated side effects, is often considered bad programming style. Unlike in the Mesham code, the C code does not create a linked list of end nodes - instead Gadget-2 maintains a list of array indexes. Generally speaking, as a technique this is not only more complex to understand and maintain, but it is also fragile due to if an element of the array is entered or removed then the indexes must be updated. Gadget-2 does avoid this by not allowing the array to change, but this is a bug prone area where future updates by third parties could very easily cause unforeseen problems.

```
1  function void communicateChildren[]
2  {
3          var collected:array[BHChildren,processes[]] :: allocated
              [multiple[]];
4          var i;
5          for i from 0 to processes[] - 1
6          {
7                  (collected#i):=(childroot::broadcast[i]);
8                  if (i > 0) linknodes[collected#0,collected#i];
9          };
10         childroot:=(collected#0);
11 };
12 function void linknodes[var a, var b]
13 {
14         a:BHChildren :: allocated[multiple[]];
15         b:BHChildren :: allocated[multiple[]];
16         var tempa:BHChildren :: allocated[multiple[]];
17         tempa:=a;
18         while ((tempa.next) != null)
19         {
20                 tempa:=tempa.next;
21         };
22         tempa.next:=b;
23 };
```

Listing 6.4: Communicating the BH Tree

Once the BH trees have been built, every processor must send every other processor its part of the tree containing information about process's particles. It is important that each processor has the same information, so that they can all correctly identify which particles need sending to and receiving from other processes. Listing 6.4 illustrates the Mesham code required to broadcast the tree to each other processor and link these nodes up. The function *communicateChildren* is called, in the loop of lines 5 to 9, each process will in turn broadcast their linked list of tree end nodes to every other processor. The *referencerecord* type of variable *childroot* has been written such that in communication, each reference to other variables will be analysed, packaged up into the communication and sent along with the original data. On the receiving side the data will be unpackaged and references reissued. This abstraction allows the programmer to send an entire linked list data structure with only one line of code, saving them from worrying about the low level, complex, details of the operation which requires some expertise to complete in a timely manner. After each process's end nodes have been received they are added onto the end of the *collected* end node linked list via the function *linknodes*. In order to achieve the same result, the Gadget-2 C code uses an MPI Gather to transmit and receive all the arrays of data, which has a number of disadvantages. Not only does the size of data from each process need to be known prior to this operation but also adding or removing tree attributes is very difficult with considerable side effect - in the Mesham code all this is dealt with automatically allowing the programmer to take a much more high level view of the parallelism.

6.5 Communicating Particles

Once each process has agreed on which particles need to go where, an efficient method of exchange is required. In order to achieve this, an *arraymapped* array is used in the Mesham port of Gadget-2. As can be seen from the code in listing 6.5 variable *a* is defined to be an array of Particles, size *numparts*, allocated to processes via a mappedarray and shares its memory space with variable *P*. The *arraymapped* type will split the array into *processes* blocks, each process's block is of size *totalsizes* (an integer array) and the variable *allsize*, also an integer array, actually maps each element in the array to a specific block. For instance the value *3* at location *120* in array *allsize* will inform the type that element number 120 in the array belongs to process 3. These blocks are then evenly distributed amongst the processes with the *evendist* type. The assignment of line 2 *a:=P* is where the actual communication is completed.

```
1 var a:array[Particle,numparts] :: allocated[arraymapped[
    processes,totalsizes,allsize] :: single[evendist[]]] :: share
    [P];
a:=P;
```

Listing 6.5: Exchanging particles in Mesham's Gadget-2

Compared with the Gadget-2's C code, the Mesham code is very simple. In Gadget-2 each process will loop through sending specific particle data to every other process via synchronous MPI sends. Each other process must match these sends with a receive and the programmer must ensure that this is written correctly as to avoid deadlock. Additionally, only a specific buffer space is allowed in Gadget-2, so that this communication may need to be completed in multiple cycles. Using the Mesham types, the programmer has been abstracted away from all this detail - in reality the same buffer checks and matching sends and receives are also being issued by the type's code generation, but the programmer need not worry about this nor about the actual form of communication which has been decided during compile time.

6.6 Conclusions

As has been seen throughout this chapter, Gadget-2 is a complex application. Even in porting the sections detailed here has taken considerable time and requires an in-depth understanding of the application. It can be seen that the Mesham code is at a higher, more simple, level than the original C. This new code is also more flexible as it is relatively simple to experiment with new concepts and ideas simply by changing the type of variables.

Chapter 7

Evaluation

7.1 Introduction

It is important to answer the question of whether or not the type-based approach and associated Mesham language are a success. Of all the languages reviewed in Section 2.3, those which are simple are not efficient and the efficient ones are complex to use. The real question is does the type-based approach solve this problem? Another key consideration is exactly where the language lies in comparison with other parallel programming solutions. In this chapter, using the results of the case studies described in Chapters 5 and 6, the type-based approach will be evaluated and compared with other solutions to assess its viability.

In Chapter 1 it was mentioned that existing parallel languages exhibit a tradeoff between simplicity and efficiency. In evaluating the type-based approach the concept of programability is considered, this is not just simplicity but all the other factors, such as flexibility, which contribute to making the programmer's job as easy as possible.

7.2 Results

In Chapters 5 and 6 a number of different case studies, which have been written using Mesham were described. These studies range from the very simple Mandlebrot example to the complex port of key aspects of Gadget-2. As has already been discussed, two main objectives must be met - does Mesham allow for the programmer to write (relatively) simple and also efficient code?

7.2.1 Performance

Of all the timing experiments carried out, code written in Mesham performs comparatively or better than existing language solutions. This is due to the extra type information provided by the programmer to assist in static analysis and optimisation of the Mesham compiler. The code generated by the Mesham compiler for specific types can be tuned for performance without having to modify any of the source code. This tuning happens during development of the type library and additional types could easily be added to the language, without side effect, to further enhance performance.

It can be seen that the type-based approach of Mesham provides for a smoother performance curve. For instance during FFT experimentation in Section 5.4, when run over an uneven number of processors, the FFTW code experienced severe slowdowns whereas the Mesham code did not. The reason for this smoother curve is that the relevant types determined implicitly, during compilation, an alternative form of communication was required to maintain levels of efficiency. Performance transparency is an important factor for parallel programmers by providing types which allow for a smooth performance curve, regardless of data decomposition and is a major benefit of Mesham. By providing this higher-level of abstraction, Mesham facilitates these sorts of optimisations whereas lower-level parallel languages require time consuming end programmer optimisations to achieve the same results.

From all the timing experiments performed it can be seen that after the optimum number of processors has been reached the performance of the Mesham code drops far less severely than that of the control code. This is most obvious with 128MB FFT in figure 5.6 but can also be seen in the NAS IS benchmarks of Section 5.3. This result re-enforces the fact that code written in Mesham is more optimal communication wise than in the currently popular used, lower level, C-MPI. The reason for the optimised communication is that the compiler has a rich source of information, provided by the programmer, to perform analysis and optimisation upon.

A popular choice amongst current parallel programmers is to use language features such as low-level pointers to their advantage. This approach is most obvious in NASA's IS benchmark code where the programmer manipulates pointers to save both execution time and memory space. Apart from the obvious programmability penalties to this approach it can provide a useful performance increase. The high level approach adopted by Mesham does not allow for this unsafe practice or programmer optimisation at this low level. Instead the compiler, and specifically type library, of Mesham will use the information provided by the programmer to perform these sorts of optimisations as much as possible. However it is appreciated that well programmed C code using these pointer optimisations written by experienced programmers will often have a performance advantage when it comes to computation. This sentiment is reflected in the timing results, specifically for NAS-IS. Initially (up to 4 processes) computation is the most important factor and as such NASA's C code performs better. However as the number of processors is increased there is a shift towards the importance of communication outweighing that of computation and as such the Mesham code performs more favourably.

Scalability is another important aspect of parallel computing. It is important for a language to promote code writing which is scalable - both in the number of processes and size of the input. Skillicorn [Skillicorn1998] quotes that " Scalable architectures are not powerful and powerful architectures are not scalable." However, by allowing the programmer to write code in a high-level shared memory abstract model provides a mechanism which is scalable and transforming this (via all the program information) into lower-level message passing supports performance. Unlike many other languages, if written correctly, Mesham code with the type library should automatically deal with an increase in the number of processors and/or the input size without having to modify the code. An example of this at work can be seen in all the case studies conducted. The existing C code has in all cases had to be written carefully to handle a dynamic number of processors and input size, or carefully modified in the case of Mandlebrot. When writing the Mesham code, no specific scalability issues needed addressing due to the compiler handling these issues automatically.

7.2.2 Programability

Current parallel languages are, quite rightly, seen as difficult to work with, maintain and can often by their very nature allow for the programming of unsafe parallel code. By abstracting away from all the low-level details it has been an aim to provide the programmer with a flexible, simple to use language which can promote increased use of these parallel resources. Programability is very much a subjective area, although there are some general conclusions which can be drawn from the case studies already reviewed.

The first issue to consider with programability is number of lines. This measure is certainly not an exact indicator, it is possible in some languages to write very short programs which are highly complex. However, in all the case studies reviewed the Mesham code is considerably shorter than the control code. The Gadget-2 case study is the most revealing of these, where the Mesham I/O code is 16 times shorter and the domain decomposition 3 times shorter than its C counterpart. The reason for the dramatic reduction in code size is that the programmer is just concerned about the high level parallel details with, mainly abstractions provided by, the type library dealing the lower level mundane ones. This abstraction makes the code easier to write, understand and, most importantly with Gadget-2, maintain.

Section 2.3.1 introduced the concept of using an automatic parallelising compiler to allow for the programmer to write sequential code and then have it run in parallel. The paper [Lou2005] introduced an experiment whereby four existing benchmarks were stripped of all the parallel details and the number of lines and execution time of this resulting code was compared. This process resulted in between a 4 and 11 times reduction in code size with performance hit of the new code being between 2 to 6 times slower depending on the benchmark. In all case studies considered the Mesham code is shorter than the existing C code, taking Gadget-2 as an example there is a code size reduction of between 3 to 16 times. However the major benefit of Mesham over the approach described in [Lou2005] is with respect to performance. As has been shown, the performance of code written in Mesham is certainly comparable to existing parallel codes, which is not the case in the experiments detailed by [Lou2005].

Parallel codes written in Mesham are generally more flexible and maintainable than those written in other languages. For instance changing key program attributes, such as communication form, only requires a simple change in type. Many other current parallel languages would require far more work to achieve the same result, often with the programmer having to consider issues of side effects. The reason flexibility is important is because often parallel programmers wish to fine tune their working code. Mesham's type-based approach allows for programs to be written using default options and then the programmer can added additional types to improve on performance. Existing languages, especially the lower level ones, simply do not provide for this and often key attributes must be considered and "hard coded" into the program from the outset which, in retrospect, are not always the best choice.

By abstracting away from the more mundane lower-level details does very much enhance software development and help avoid issues such as program bugs. With other parallel languages, especially those such as C-MPI, subtle errors can cause problems in communication and computation. These issues are all abstracted away from the Mesham programmer. Types can be thought of as building blocks which the programmer can rely on to write their code. Although not formally proved correct, the type library has been extensively tested and use of these types does make programming easier as the coder need only be concerned with the correctness of their specific code. However this does highlight a problem with the current language. Although types are very easy to add to the language, such as Gadget-2 extension types, they must be added at the language level to the compiler rather than created by end programmers. Allowing end programmers to create and use their own program types within their code is an interesting idea and certainly a candidate for further exploration when the type-based concept has been developed further. It is thought that allowing for types to be created within programmer's code could bring programability benefits similar to the OO paradigm.

A major issue with some of the existing languages used for parallelism is safety. In some languages it is all too easy to encouter parallel issues such as deadlock if the code is not carefully constructed. Mesham provides for a number of language defaults which are "guaranteed" to be safe. When using these inbuilt features the programmer will not encounter common problems such as race conditions, deadlock or livelock. This is an important feature of the language - many parallel programmers are not formally trained in computer science and those coming from a sequential background often find it difficult to understand and deal with these extra potential problems. Having said that, by enforcing safety the programmer's expressiveness can be somewhat limited and there are additional performance hits. To this end the language does provide types which override the default behaviour giving the programmer more control (and performance) yet the cost of these is that they are not guaranteed to be safe. A common example is that of communication where the default behaviour, which is often sufficient, will have a minor performance hit if communication is used dynamically (MPMD style in a par loop.) This can be overridden by communication primitive types, which may improve performance when used dynamically, but the responsibility rests on the programmer to ensure for safety when using these.

The programming approach encouraged by Mesham is for the programmer to write code which is portable amongst architectures. In other languages it is often difficult to write portable code, especially when communication must be considered, with many pitfalls facing programmers. By design the type-based approach and Mesham are architecturely unspecific, disallowing the programmer to write non-portable code. In reality factors such as the code generated by the Mesham compiler, the communications library used and target architecture are unimportant and can be hidden from the end programmer, thus simplifying the entire process.

7.3 Skillicorn's criteria

Chapter 2 introduced a number of parallel programming language evaluation criteria chosen by Skillicorn in his paper [Skillicorn1998]. These six are the ease of programming, existence of method for development, independence of target architecture, simplicity and abstractness, guaranteed performance and the existence of costs which can be inferred from the program. As was seen in Section 2.3.4, none of the existing languages met all these objectives.

7.3.1 Ease of Programming

The first of Skillicorn's criteria, ease of programming, is essential to any language. This criteria is a subjective one and has been a major aim of the type-based approach and Mesham. From the results of experimentation, with the different types provided to the programmer, it can be seen that Mesham is comparatively easy to program. Compared with other languages code written in Mesham is shorter, easy to modify without side effects and automatic default options mean that it can be written without an in-depth knowledge of the language. In his paper Skillicorn argues that a model which is easy to program should hide decomposition of the program into parallel threads, abstract away mapping these threads onto processors, keep communication away from the programmer and conceal synchronisation.

However, it should be noted that the languages which do take away this control from the programmer often rely heavily on static analysis and optimisation which does not always produce optimum results. To this end the approach of Mesham, providing a number of simple defaults which the programmer may rely on but also supply mechanisms so that the more advanced programmer may override them, is considered very important. The Mesham language meets all of these concealment requirements, but also allows the programmer greater control via the type system to override the imposed defaults. As performance is key in parallel computing, it is considered that this is an important evolution of Skillicorn's first criteria.

7.3.2 Software Development Methodology

For this criteria it is argued that a firm semantic foundation is required onto which transformation techniques can be built. Skillicorn notes that, due to the complexity, the methodology of testing via execution and then debugging, rather than proving correctness, does not extend to portable parallel programming. Instead it should be possible to build software which is correct by construction. Unfortunately, at the moment, it is not as easy to prove correctness as it might be using an alternative, well theorised language. Although, semantically, the imperative model is well known, we do not have the appropriate mathematical tools to model the type-based approach making it difficult to theorise about Mesham programs.

Traditionally types are modelled via operational semantics, however the way types are used in Mesham follow more of a denotational approach but still require operational semantics. This mix of semantic models is not ideal and as such currently it is difficult to theorise about and prove language and model properties. Developing such a semantic model is quite a task and as such is considered further work, as is formally proving properties such as correctness of the types. Having said that, it is made very easy for the programmer to improve their code via informally testing and debugging, although this does not meet Skillicorn's second requirement.

7.3.3 Architecturally Independent

In [Skillicorn1998] it is argued that code should be able to run from one parallel computer to another without any change. The type-based approach allows Mesham to be sufficiently abstract such that the programmer implicitly writes architecturally independent code. It is inevitable that performance between parallel machines will vary, to this end Mesham allows the programmer to trivially experiment with different types in order to pick the best combination. On an implementation view point, as already explained, the Mesham compiler will generate C code conforming to the C99 standard and uses the MPI-2 standard for communication. All non-portable functions of the language implementation are contained within a runtime library, which exist for each class of machine and is linked into the C99 code during compilation. Therefore, to run any Mesham code on a parallel machine all that is required is a C99 conforming compiler, implementation of MPI-2 standard and version of the Mesham runtime library.

7.3.4 Easy to Understand

Easy to understand is another criteria specified in [Skillicorn1998]. Not only should the model be understandable, it should also be easy to teach. Skillicorn argues that if parallel programming models are able to hide the complexities associated with parallel computing and provide an easy to use interface then they have greater chance of being accepted. From the specification in Chapter 3 and the studies in Chapters 5 and 6 it can be seen that the type-based approach is a simple one once explained fully. Of course it would require educating programmers to this new programming paradigm but it can be seen from the case studies that this fairly simple task will result in easier to understand code. In designing Mesham it was decided to centre this type approach around an imperative language, which is what the majority of parallel programmers are familiar with and use at the moment, thus making the transition easier.

7.3.5 Guaranteed Performance

Skillicorn argues that providing guaranteed performance is of great importance and as such it is the fifth criteria to be considered. It is stated that a model should have guaranteed performance over a variety of parallel architectures but this does not mean that the code must run as fast as possible - instead a balance between programmability and performance must be met. In the paper Skillicorn mentions that one of the most powerful architectures is distributed memory MPMD and, as in Section 4.7, this is the model the Mesham compiler's generated code follows. By providing well documented types the Mesham programmer knows exactly what they are getting performance wise. Additional types can be written and optimised by experts to produce the best performance possible over a variety of architectures and uses. An example of guaranteed performance is in the FFT case study, where over an uneven number of processors the FFTW code experiences a large drop in performance whereas the Mesham code is unaffected.

7.3.6 Cost Measures

Providing a model which is transparent enough for the programmer to be able to easily determine the performance is a must. It is essential for the programmer to be able to determine if algorithm A is "better" than algorithm B. Skillicorn issues a word of caution when dealing with this criteria however, cost measures are not a licence to remove all abstraction from the model. Instead the model should provide just enough information to the programmer such that it is possible to determine its cost from the text, minimum computer properties (such as number of processors) and information about the size of the input. He goes on to say that models must provide predicable costs and that compilers should not optimise programs. In reality compiler optimisation is essential - it would be impossible to have a simple abstract language without static optimisation to produce good performance. Instead it is considered more important that any compiler optimisation performed should be predictable and transparent. In designing Mesham and the type library transparency has been a major goal - the programmer knows exactly what they are getting when they use a language construct or type. As has been seen in Chapters 5 and 6, this transparency means that it is relatively simple to construct highly efficient codes because the costs of such are obvious.

7.3.7 Summary

	Mesham
Easy to Program	Y
Software Dev Methodology	N
Architecture Independent	Y
Easy to Understand	Y
Guaranteed Performance	Y
Cost Measures	Y

Table 7.1: Mesham considered wrt Skillicorn's criteria

Table 7.1 summarises how Mesham and the type based approach fits in with the six parallel model evaluation criteria laid down in [Skillicorn1998]. It can be seen that all are met except *Software Development Methodology*. The reason for this is that Skillicorn calls for formal methods to prove properties such as correctness. As a language design field the mathematical foundations really are not up to the job, as of yet, for supporting these activities. Even in his paper he states that these calculation approaches are "goals rather than practices in the medium term". However from the simplicity at the core of the type based approach, it is considered that the model will lend itself to this once the calculation concept has matured somewhat to more mainstream languages.

7.4 Other Languages

In Chapter 2 a number of existing parallel languages were reviewed and considered. An important question of this chapter is to ascertain where abouts Mesham sits in relation to these existing solutions, and whether or not the type-based approach provides for advantages over these other languages.

The main concern in this thesis is the development of a type-based approach for parallel programming, and this has been encapsulated in the language Mesham. In the comparison below references to Mesham are also references to the type-based approach.

7.4.1 High Performance Fortran

As reviewed in Section 2.3.3, HPF is a popular language for parallel work. Like HPF, the Mesham programmer can specify data partitioning and allocation but they can also control the communication and computation distribution aspects of their code if they so wish. On this level, much of the data partitioning, allocation and communication can be specified by the Mesham programmer but can equally be omitted and reliance placed upon language defaults. However, in HPF the programmer is limited to specifying the first 2 of these attributes and has no control over the third. As already mentioned, in designing the type-based approach, much emphasis has been placed upon transparency so that the Mesham programmer knows exactly what they are getting with their code. For HPF, with many important parallelization details left to the compiler, transparency is not a feature.

Another difference between the languages is that the, limited, parallel expressiveness of HPF is done via keywords whereas in Mesham types are used. Using the keyword mechanism very much tightly couples these into the language and makes future modifications difficult. By abstracting many of the parallelization details into a loosely coupled type library, Mesham avoids these disadvantages. In summary, HPF and Mesham are very different languages. Mesham allows the programmer a varied degree of control from the default options aimed at novices to high levels of control if needed by experts, whilst maintaining transparency. The HPF programmer is very much stuck with having to define certain parameters and then must rely on the non-transparent compiler transformations to determine, probably the most important parallel factors, communication and computation distribution implicitly. One factor HPF and Mesham do have in common is that if all parallel information is removed, then the program simply becomes a sequential one.

7.4.2 Co-Array Fortran

As already discussed the CAF programmer specifying explicitly data partitioning, computation and synchronisation has a greater degree of control over parallelism than the HPF programmer. However the CAF programmer has only one-sided communication which is often inefficient. In Mesham dynamic communication is, by default, one sided but this can be overridden by the programmer if required. Mesham also allows for a varied choice when allocating data amongst processes or groups there of, whilst in CAF the programmer may only define data to be local or global. The CAF programmer must also concern themselves with certain low-level issues such as index management, which can become a real difficulty when dealing with uneven distribution of data. Via the type system these details are all handled automatically in Mesham, providing simpler and more flexible code.

7.4.3 ZPL

The array programming language, ZPL, takes advantage of the fact that many parallel applications involve working with arrays of data. In array programming, the statement $C{:=}A + B$ will combine arrays A and B into C, thus cutting down on mundane programming work. In the type-based approach, the type system is flexible enough to provide for features, such as array programming, within the system itself. For instance, Mesham's array type supports array programming in this form even though this is not part of the core language specifically. By maintaining a clear distinction between the type system and the core language, it has been found that adding these additional features is simply a matter of modifying the specific type, without having to worry about language-wide side effects which can get very complicated using traditional approaches. The ZPL programmer has no control over communication, as already mentioned the Mesham programmer can take control of this if they so wish or rely on transparent defaults.

7.4.4 NESL

In Section 2.3.3 it was mentioned that functional parallel languages, due to their abstract nature, hide most of the parallel details from the programmer. The source programmer does not have explicit control over various parallelization options such as the choice of communication method and synchronization. That means good performance is guaranteed only when the default options with optimization are just right for the problem. Mesham provides a much more concrete model with which the programmer can control some or all aspects of parallelism if they so wish. The lack of transparency provided by NESL means that although the work and depth of an algorithm can be easily deduced from the functional source code, this often does not relate to runtime. When writing code in Mesham the programmer can gain a feel for how their code will perform and which algorithms are best suited from language documentation and of course some experimentation.

It is a belief of many that programming in functional languages is hard. Whilst this might not always be completely true, in conceiving this many programmers stay away from these languages due to the initial steep learning curve. In evaluating Mesham it has been shown from the case studies that writing code is relatively simple, as long as the programmer understands the underlying type-based paradigm adopted.

7.4.5 Titanium

A major advantage to Titanium is that being based on Java, an OO language, the programmer can abstract away parallel details via objects. There are some similarities between the use of types in this thesis and objects, as already mentioned an idea for further work would be to allow for the Mesham programmer to implement their own types in program code. However there is a cost associated with this, OO can impose hidden overhead which is not transparent as mentioned in Section 2.3.3.

The type-based approach of Mesham avoids these hidden overheads, there is no expensive serialisation or deserialisation in communicating data and the way types are implemented, specifically *referencerecords* means that they are careful to maintain memory locality whereever possible. Of course if the type-based approach were extended to allow types to be created dynamically in program text then some additional overhead might be encountered, but in implementing this feature the language designers would need to be careful to maintain transparency and, as much as possible, performance.

Other similarities between Titanium and Mesham is that both languages enforce safety (by default with Mesham), they are portable and it is easy to build complex data structures in both (via the record type in Mesham.) Already mentioned, the Titanium programmer is stuck writing code SPMD style which is somewhat limiting in many cases. Mesham is flexible enough to allow the programmer to use SPMD, MPMD or a mixture of both depending on which works best for the problem to be solved.

7.4.6 Message Passing Interface

Due to its popularity and performance, implementations of the MPI standard using C have been reviewed in detail in comparison with Mesham code during the case studies. It has been mentioned that this choice is low level, with the programmer responsible for all options of parallelization. Being a sequential language, writing code this style also requires the programmer to think in a sequential manner and means that it can be difficult for them to consider the "big" parallel picture. Additionally, code written using MPI is often not flexible unless written very carefully. When written by experts parallel codes using this option can perform very well, but even the best do make mistakes; in porting Gadget-2 into Mesham minor bugs were found in how the original C code handles HDF5 I/O.

The differences between MPI and Mesham are vast. The programming model adopted by Mesham is much more abstract, allowing the programmer to easily take a high-level view of their code whilst giving them high-level ways of control the aspects of parallelism they wish to. Additionally, when used with its default options (specifically communication) the Mesham language guarantees safety, which MPI does not. Using MPI it is all too easy to write unportable code, especially when used from C because of the ability to write code which is subtly not according to the standard but still allowed on a specific architecture and compiler. In Mesham portability is designed into the language.

However, there are some enviable qualities of MPI, namely the performance and expressiveness it can provide for. In designing Mesham and the type library these have been important qualities to incorporate. Firstly, in order to achieve performance the Mesham compiler will translate source code into C using MPI, applying as much optimisation as possible. This approach should provide the best of both worlds, a high level simple to use language which translates to as efficient as possible target code. As shown in the performance section of case studies this approach does work well generally although the MPI programmer could apply further, time consuming, optimisations on their code over and above what a compiler can provide for.

As for expressiveness it has been an aim in designing the type library, specifically the primitive communication types which the programmer can use to override default communication, to provide the Mesham coder with all the communication methods which the MPI standard also supports. This is an important factor, some parallel programmers really do find these options useful and as such should be available to advanced users. Via the design of the type library if additional forms of communication were required in the future then it would be a trivial task to supply them.

Functions provided by an implementation of MPI can offer a high degree of flexibility in their use, especially when combined with specific languages. For instance, the MPI library implementers might never have envisaged sending a specific form of data but this can be done by providing the memory address and specifying the type to be *MPI_BYTE*. Whereas the type library which has been designed into Mesham is somewhat more limiting. The reason for these limits is to provide for safety and compiler optimisation, but some existing programmers might very well prefer the more flexible library design.

7.4.7 Bulk Synchronous Parallelism

Like MPI, BSP is a standard which has been implemented in library form and used in conjunction with different languages. One of the more popular combinations has been BSPLib with C. In providing the programmer with a shared memory model this choice is at a higher level, and somewhat easier to use, than MPI. Additionally, being shared memory, communication is much simpler with fewer options. Mesham provides the programmer with a shared memory view, but allows for far more expresivity via the type system if they so wish. Additionally, when using an implementation such as BSPLib, the programmer must still consider low-level details such as pointers which is avoided in Mesham.

The last point to consider is that of performance, "the performance of barriers on distributed-memory machines is predictable, although not good." [Hill1999] This might be one of the reasons why other solutions, such as MPI, have become more popular. In using the rich program information provided by the programmer, it has already been shown that Mesham can compete performance-wise with solutions such as MPI and as such is deduced to outperform BSP.

Standards such as MPI and BSP do have an important advantage though. Implementations of these standards can be used from many different languages. For instance, MPI bindings exist for C, C++, Fortran and Java as well as many others, so the programmer can pick their favourite language and then use the bolt on parallelism. At the moment the type-based approach is limited to the imperative Mesham language which the programmer would be forced to use. As has already been considered, the type library and actual language are distinct, so it would be relatively simple to attach the type library to another language which provides basic type support. This is an area for further consideration.

7.4.8 Cilk

Cilk and Mesham differ fundamentally, whereas Mesham is based around data parallelism, Cilk is based around task parallelism. Although task parallelism is an important issue, often threads can be handled on an operating system level with a reasonable level of success. The major selling point of Cilk is that it places emphasis on efficient scheduling of these tasks by using extra information the programmer has provided. Cilk limits itself to SMPs, connected to the same shared memory and as such is not scalable like other parallel solutions considered. In doing this means that communication need not be considered, with only simple language mechanisms in place to synchronise tasks.

Task-based parallelism is only useful up to a point, there becomes a limit to the number of tasks which a program can be split into. Data parallelism, based upon each processor solving the same task on different parts of the data, is often far more relevant in the scientific domain which parallel computing is popular with. Having said that, data parallelism is more difficult to achieve, inevitably requiring a data parallel language to be more complex than task-based one - hence only two additional keywords in Cilk compared with the novel type approach developed for Mesham. By the design of the language, a Mesham programmer can execute their code transparently on SMPs, over a cluster of machines or on a mixture of both which is unavailable to those using Cilk.

7.4.9 Summary

As can be seen in this section of all the existing language solutions in current use Mesham is unique and exhibits certain advantages over the rest. Providing the programmer with certain transparent parallel defaults which can be easily overridden promotes both ease of programming and expressiveness. Of all the other languages considered, those which abstract away some parallel details are not transparent and often suffer from a lack of efficiency in their implementations. Those parallel languages which have built in explicit control often do this via keywords which are hard coded into the language and difficult to modify if needed.

Sequential languages using specifications such as MPI might produce an efficient result, but are often difficult to program and the results hard to maintain. By taking a high-level view of parallelism the Mesham programmer can write simple code yet with all the extra information provided to the compiler often much optimisation can be performed to match performance with these existing solutions.

A weakness exposed in Mesham is that, for the programmer to use the type-based approach to its full advantage, these types must exist within the library. Whilst every effort has been made to produce a set of general, flexible types it is inevitable that there will be some requirements for additional types. The language in its current form would require these types to be added to the compiler, which is relatively simple but not appealing to many programmers. As already mentioned, a major improvement to this approach, and Mesham, would be to allow end programmers to create types dynamically - although careful consideration must be given to any side affects of this. Table 7.2 illustrates how Mesham ties in with the other existing languages that have been considered.

	Imperative	Functional	Library Extension	OO	Type Based
Message Passing	-	-	MPI	-	-
Shared Memory	HPF,CAF,ZPL,Cilk	NESL	BSPLib	Titanium	Mesham

Table 7.2: Overview of Parallel Languages Considered

7.5 Conclusions

In this chapter the type-based approach has been evaluated and is found to perform well and solve the problem of programability. These objectives which have been considered contradictory up to this point are both very important if parallel programming is to grow and succeed. Out of the six criteria defined by Skillicorn, Mesham is shown to honour five of these (which is impressive), with only one criteria being a basis for further work and improvement.

In order to give the reader some indication of where Mesham sits in relation to other solutions, the language has been compared and contrasted against existing ones. It has been shown that Mesham, and the type-based approach, do provide some unique properties which exhibit certain advantages over other languages which are currently in use.

Table 7.3 summarises all the languages considered with respect to Skillicorn's evaluation criteria. It can be seen that, in comparison with the other languages, Mesham meets the most objectives. In second place is Titanium which meet four of six criteria. Third place goes to ZPL, HPF and CAF which satisfy three. NESL is in fourth place, meeting two objectives, BSP and Cilk are in fifth place and MPI is in last place only meeting 1 criteria. The irony of table 7.3 is that, although meeting the least of Skillicorn's criteria, MPI is still by far the most popular parallel programming choice.

	HPF	CAF	ZPL	NESL	Titanium	MPI	BSP	CILK	Mesham
Easy to Program	Y	Y	Y	N	Y	N	N	Y	Y
Software Dev Methodology	N	N	N	Y	N	N	N	N	N
Architecture Independent	Y	Y	Y	Y	Y	N	N	N	Y
Easy to Understand	Y	Y	Y	N	Y	N	N	Y	Y
Guaranteed Performance	N	N	N	N	Y	Y	Y	N	Y
Cost Measures	N	N	N	N	N	N	Y	N	Y

Table 7.3: All Languages considered wrt evaluation criteria

Chapter 8

Conclusions and Further Work

8.1 Introduction

This thesis has aimed at addressing the parallel programming language problem. As introduced in Chapter 1, an area which has not seen a great deal of improvement, although considerable research has been done, is in the development of languages used to create parallel codes. The difficulty of programming has been the main challenge to parallel computing over the past several decades. As discussed, up to this point, none of the existing parallel languages are sufficient for the task in hand - they are either conceptually simple or efficient, but none exhibit both these qualities. If the use of parallelism is to move from the domain of the few experts towards the more general computing user, opened up by recent developments in CPU and GPU technology, then this problem must be solved.

This thesis first identifies the principals employed within parallel programming and the existing tools currently used. Secondly, through surveying current parallel languages, it has been determined that none of these solve the problem of allowing a programmer to write conceptually simple yet highly efficient parallel programs. The third, and most important contribution of the thesis, has been the introduction of the type-based approach and the associated programming language, Mesham, to act as a vehicle for this. The thesis then describes the implementation of the Mesham compiler and some associated issues with actually creating this type-based approach. Finally this type-based approach, including Mesham, has been evaluated by implementing a number of case studies and considering important parallel issues such as efficiency and simplicity.

8.2 Contributions

The principles and tools employed within parallel computing and programming language design are the first thing that this thesis identifies. From the outset it was considered that in order to design a parallel programming language one first needs to become a parallel programmer and build expertise in this area. The popular technologies employed in parallel computing and how they are used was an important starting point. Also considering specific parallel algorithms and applications such as that of the FFT and Gadget-2 which are both commonly used has helped identify the complexity of these existing programs and the attributes which are of most importance.

When considering the existing parallel languages available to the programmers it was seen that none actually meet the requirement of being conceptually simple yet highly efficient and expressive. Each parallel language was subject to a number of tradeoffs, with the language designers often taking decisions away from programmers in the name of simplicity. An example of this was in Co-Array Fortan, described in Section 2.3.3, where the programmer had no say over the actual form of communication. When looking at Skillicorn's six parallel language criteria introduced in Section 2.3, it could be seen that no existing language met all of these. This has been a long standing problem in the parallel computing field, there is no ideal language. Performance is more often than not the most important factor in parallel computing, which is why parallel programmers have chosen languages such as C linked with a communications library over simpler less efficient ones such as High Performance Fortran.

The major contribution (Chapter 3) of the thesis has been that of introducing the type-based approach itself. By following this new programming paradigm the complexity of parallel programming has been taken out of the core language and put into a loosely coupled type library. The programmer can use and combine these types to control all aspects of parallelism, as well as defaults built into specific types in case information is missing. An example of these defaults is that of communication. Built into element types such as *Int* is default forms of communication which guarantee safety, but the programmer can override these with primitive communication types to obtain, for instance, an increase in performance. However defining this type-based approach was not enough, in order to present and test it, a language had to be created which could act as a vehicle for the concept. This language, Mesham, has been designed as an imperative language with access to a loosely coupled type and function library. Minimal constructs are incorporated in the language to support the type-based approach.

This type-based approach supports simplicity because as long as the types are well documented, as in Section 3.3, the programmer knows exactly what they are getting. It is also possible for the programmer to rely initially on some simple types and then experiment using more complex ones without having to rewrite large portions of their code. A prime example of this is again with communication, as mentioned to change the form of communication in Mesham simply requires changing the type, whereas with languages such as C often a large amount of work would be needed. This high level view as imposed by the type system also simplifies things a great deal, such as with the FFT case study in Section 5.4 where an uneven distribution of data is automatically dealt with in an efficient manner. From a language and compiler design point of view, importantly, this type-based approach is flexible. It is possible for instance to add, remove and modify types without worrying about language wide side effects which are often present in other paradigms.

This thesis describes the implementation of the Mesham compiler and the type-based concept. Although quite a minor consideration there were issues which the approach raised on an implementation level requiring addressing. One such issue was that of how to dynamically link the compiler's type objects together during translation. As the programmer is free to combine types together in many different ways the traditional OO approach of class hierarchies was not sufficient and an alternative method needed to be found as explained in Section 4.5. The implementation issues were not just limited to the compilation phase however, it was also important to consider the target code and how to provide for a language which was both portable and efficient. It has been demonstrated that the concept of allowing the programmer to write code in one communication model (shared memory) and then translating it into another model (message passing) does work well, combining the advantages of both models into a single language. The approach that followed, of generating only C99 conforming code which linked with an implementation of the MPI standard and a language runtime library containing all the platform specific code worked well.

Finally, the type-based approach and associated language were evaluated by implementing a number of varied case studies in Mesham. The two main considerations were that of simplicity and efficiency. Due to the programmer providing much high-level information with which the compiler can use to perform static analysis and optimisation performance was demonstrated to be comparable, and in some cases better, than existing code. The most important performance case study was that of NASA's Parallel Benchmark (NPB) suite and specifically the Integer Sort (IS) specification. From the timing results in Section 5.3 it can be seen that, overall, the type-based approach and Mesham provide the programmer with as good performance as existing high performance language solutions. A common feature of the performance tests has been that in existing code certain "hacks" are used to improve performance computationally. When the number of processors becomes significant, often 4 or more, then the importance of efficient communication becomes increasingly important and as such Mesham starts to compare with or outperform existing high performance languages. This communication efficiency continues even past the optimum number of processors, generally with code written in Mesham suffering from less of a performance drop.

In order to evaluate simplicity of the type-based approach, this thesis included case studies with varying degrees of complexity. The most important case study in terms of programability was that of Gadget-2. Chapter 6 describes the process of porting aspects of Gadget-2 into Mesham and the extra types which were added to the language to facilitate this. As demonstrated by this thesis, the ported parallel and I/O code is much simpler than the existing C code with a dramatic reduction in size. As can be seen by the codes in Chapter 6, being able to take this high level parallel view and abstract away the low level complexity does make a huge difference. These programability improvements are not just limited to Gadget-2, all the case studies considered are simpler in Mesham than their original form.

8.3 Criteria for Success

In Chapter 1 a number of criteria were specified as being the criteria for success. Whether or not the research has addressed these is considered in this section.

"Support code which is simple yet expressive: This criterion specifies that parallel code should be conceptually simple to write yet still allow for advanced programmers to enjoy a high degree of control over parallel decisions."

As detailed in Chapter 3, the type-based approach allows for the programmer to omit much type information if they so wish and rely on in built defaults which, in most situations, will be acceptable. To the more advanced programmer additional types can be specified which afford a greater degree of control. A prime example of this is with communication where the element types of Section 3.3.2 provide for default communication which is guaranteed to be safe but might have a slight performance hit. More control, and performance in some cases, can be obtained by combining these element types with a primitive communication types detailed in Section 3.3.5. This demonstrates that the first criterion for success has been met.

"Provide for flexible parallel programming: Parallel programmers often wish to get their code working and then fine tune for performance. With many existing languages changing parallel details later down the line can be very time consuming and as such programmers can be stuck with initial, ill informed, decisions."

This second criterion has been met. In Chapter 3 it was discussed how the programmer can modify the type of a variable throughout program code, either permanently or for only an expression. It was also demonstrated that, in order to change many parallel options, all required is a simple change in type. For instance in Section 3.3.7 the notion of horizontal and vertical partitioning was introduced. It is very possible to initially partition an array one way, say horizontally, and then once the code works satisfactorily experiment with other types such as partitioning vertically. As the type library deals with all the low level complexity of this operation, such as associated communication and index management all that is required from the end programmer is the change in type. Comparing this with a language such as C, considerable modifications would be needed to be made to the source code to achieve the same result.

"Be general and non-application specific: There are a wide variety of parallel applications currently being used. As such it is important to develop an approach which is general and can be applied to not only existing problems but future ones too."

A variety of wide ranging case studies have been developed in Chapters 5 and 6. In designing the approach and type library of Mesham it was a key consideration to keep these types as general as possible. Having said that, as detailed in Section 6.2, a number of types did have to be added to the language to support the porting of Gadget-2. Whilst it was a simple task to add these it is not practical to expect the end programmer to modify the language and compiler that they are using. To this end the type-based approach is general and applicable to many problem domains, but it could be made more general by allowing the programmer to specify their own types in source code as suggested in Section 7.2.2. To this end the third criterion has been met to a degree, but some further work and development of this approach would improve the applicability.

"Exhibit a high degree of performance: Performance is the main concern within parallel computing. Any proposed approach must be, at least, as efficient as existing high performance language solutions to stand a chance of adoption."

As demonstrated in Chapter 5 the type-based approach, and Mesham, performed comparatively against existing high performance parallel programming languages. A general theme of the performance graphs was that the computational aspects were often faster in the existing C code, with the parallel aspects faster in Mesham. One of the main reasons for the computational difference was due to the nature of C allowing the programmer to carefully optimise their code using low-level concepts such as pointers. This was especially apparent in the NAS-IS benchmark considered in Section 5.3. Therefore it can be concluded that the type-based approach does exhibit a high degree of performance and when run on a none negligible number of processors (4+) will compare with or even beat existing languages. There is some work to be done on the programming language, Mesham, and compiler to improve the computational efficiency.

"Must be implementable: Arguably there is little point of a paradigm or language if it can not be implemented on a computer. From the specification it must be possible to produce translation tools which work in a timely fashion."

Chapter 4 summarised the implementation of the Mesham language. Not only was the compiler discussed, but additional concerns such as the form of generated code to maximise performance, scalability and portability were also addressed. From this chapter it can be seen that, from the language definition in Chapter 3 and Appendix A, a compiler writer can quite easily implement the type-based approach and supporting language. Therefore it is concluded that this final criterion for success has been met.

8.4 Further Work

This thesis has introduced the type-based approach and demonstrated that using the concept is a realistic possibility with a number of potential advantages. Having said this, there are still a number of important avenues which warrent further work and exploration.

1. Develop a mathematical, semantic, definition of this type-based approach. When considering the semantics of the approach, using current techniques, it would exhibit attributes both of denotational and operational semantics as introduced in Section 2.4.2. It is the feeling of the author that the semantic definition of this type-based approach would be quite similar, although with some differences, to that of OO if such existed. The creation of such mathematical tools would allow for a precise definition of this approach and support proving important language properties such as correctness. This could be the starting point for Skillicorn's second criteria of software development methodology as introduced in Section 2.3. For this criteria he argues that parallel software should be correct by construction rather than extensive testing and debugging as is currently the norm. Having a strong mathematical foundation and specific proven properties of the approach is key if criteria is to be realised.

2. Allow programmers to define their own types. Evident in Chapter 6, to support the implementation of Gadget-2, a number of new types were added to the language. Whilst it would have been possible without these extension types, albeit it making the process a lot more difficult, adding them was a simple task for the language designer. However, the average programmer can not be expected to do this and as such a mechanism by which these types could be defined in the source code would be of benefit to the approach. Considering for the minute a language such as Java which has an extensive object based API, if the programmer were stuck just with the objects in this API and could not create their own then it would feel far more restrictive. Although type-based and OO approaches are different, it is a similar point of importance. Having said that, if this extension were to be persued, then the research would have to address how to incorporate this in an efficient manner. Additionally allowing the end programmer to create their own types in the source code may allow for more fine tuning of performance, providing specific types for this, over and above what the language supplies at the moment.

3. The idea of expanding the type-based approach to other languages is certainly one worth persuing. In this thesis the type-based approach has, mainly, been considered in conjunction with parallel programming. However there is nothing to say that this can not work in sequential programming too. An example of this can be seen where the I/O sections of Gadget-2, which are sequential, were ported as described in Section 6.3 into Mesham. The result of this was that the type-based code was simplified over the existing C code, which demonstrates that the approach works not only in parallel computing but also more generally. One prime area where this might be extended to, is that of GUIs which present a number of attributes and often result in time consuming large code. A modification of this further work concept is to retrofit existing languages with the type-based approach. For instance, many parallel programmers are comfortable using C, it may be of benefit to them to provide for these higher level constructs within an existing language that they trust and are happy with.

4. Investigate the provision of multiple type libraries for a single language. Taking the idea of plugable types as reviewed in Section 2.4.4 as a starting point, it would be possible to have a simple core language with which a number of type libraries are provided. By selecting a specific type library the programmer could dramatically change the language that they are using. A concrete example of this can be found by considering Mesham, the current type library could be thought of as the *parallel library*. There could also exist a *sequential library* which uses the high level of type information to generate well optimised sequential target code. A third library, the *embedded library*, could also exist which uses the source code type information to produce code aimed at embedded devices. By selecting the appropriate type library, the programmer could easily achieve very different results without modifying their code. However there is a downside, by allowing the type library to be changed in this manner would result in specific libraries being incompatible with specific language source codes. More difficultly, some type libraries might seem to work with specific source codes but the semantics of the code could be very different which might not be obvious to the end programmer.

5. Investigate the optimum number of processors for a specific problem and input size. For all the case studies in Chapter 5, after the optimum point a severe drop in performance was experienced. Interestingly for the NAS-IS benchmark in Section 5.3 after this optimum point increasing the input size actually improved performance as can be seen in figures 5.2 and 5.3. Many existing parallel programmers believe that simply "throwing" processors at a problem will make it run faster, which has been shown to be completely untrue. It is clear that some further research should be done into this behaviour in order to develop some tools such that parallel programmers can predict the optimum point. The concept that a parallel language could analyse source code and determine the optimum number of processors is a worthwhile aim, although it would require considerable further work to achieve.

References

[Aho2006] A.V.Aho, R.Sethi and J.D.Ullman (2006), *Compilers: Principles, Techniques, and Tools*, Prentice Hall, 0-321-49169-6

[Aldinuccia2000] M.Aldinuccia and M.Daneluttob (2000), *Skeleton based parallel programming: functional and parallel semantics in a single shot*, Computer Languages, Systems and Structures, Volume 33, Issue 3-4, Pages 179-192

[Baily1994] D.Baily, E.Barscz, J.Barton, D.Browning, R.Carter, L.Dagum, R.Fatoohi, S.Fineberg et al (1994), *The NAS Parallel Benchmarks*, NAS Technical Report RNR-94-007

[Baker2006] M.Baker, B.Carpenter and A.Shafi (2006), *MPJ Express Meets Gadget Towards a Java Code for Cosmological Simulations*, In Proceedings of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI 2006), September 17-20, 2006

[Blelloch1990] G.E.Blelloch (1990), *Vector Models for Data-Parallel Computing*, MIT Press, 1990, 0-262-02313-X

[Blelloch1995] G.E.Blelloch (1995), *NESL: A Nested Data-Parallel Language*, Carnegie Mellon University, Technical Report: CS-93-129

[Bracha2004] G.Bracha (2004), *Pluggable Type Systems*, In OOPSLA Workshop on Revival of Dynamic Languages, 2004

[Brown2006] N.E.Brown (2006), *Flexibo to C Translator*, MSc Thesis Durham University

[Brown2008] N.E.Brown and Y.Chen (2008), *Type-Based Parallelization And Code Generation for MPI*, Technical Report of Durham University

[Cardelli1985] L.Cardelli and P.Wegner (1985), *On Understanding Types, Data Abstraction and Polymorphism*, Communications of the ACM, Volume 17, Issue 4, Pages 471-523

[Cardelli1997] L.Cardelli (1997) *Type Systems*, The Computer Science and Engineering Handbook 1997, Chapter 103, Pages 2208-2236, CRC Press, 0-8493-2909-4

[Carpenter2000] B.Carpenter, V.Getov, G.Judd, A.Skjellum and G.Fox (2000), *MPJ: MPI-like message passing for Java*, Concurrency: Practice and Experience, Volume 12, Number 11

[cFAQ] comp.lang.c Users, *C Frequently Asked Questions*, http://www.c-faq.com/, *(Last accessed October 2009)*

[Chamberlain1998] B.L.Chamberlain, S.Choi, E.C.Lewis, C.Lin, L.Snyder, and W.D.Weathersby (1998), *The case for high level parallel programming in ZPL*, IEEE Computational Science and Engineering, Volume 5, Issue 3, Pages 76-86

[Chen2004] Y.Chen and J.W.Sanders (2004), *Logic of Global Synchrony*, ACM Transactions on Programming Languages and Systems, Volume 26, Issue 2, Pages 221-262

[Chen2004-2] Y.Chen (2004) *A Languange of Flexible Objects*, Technical Report Department of Computer Science, Leicester University

[Cousot1975] P.Cousot and R.Cousot (1975), *Static Verification of Dynamic Type Properties of Variables*, Laboratoire IMAG, Universite Scientifique et Medicale de Grenoble, Research Report R.R. 25

[Cousot1977] P.Cousot and R.Cousot (1977), *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Pages 238252

[Cousot2002] P.Cousot (2002), *Abstract Interpretation: Theory and Practice*, Lecture Notes in Computer Science, Volume 2318/2002, Pages 3-4

[Cousot2005] P.Cousot (2005) Abstract Interpretation in a Nutshell, http://www.di.ens.fr/ cousot/AI/IntroAbsInt.html, *(Last accessed October 2009)*

[Deitz2003] S.J.Deitz, B.L.Chamberlain, S.Choi, and L.Snyder (2003), *The design and implementation of a parallel array operator for the arbitrary remapping of data*, ACM Conference on Principles and Practice of Parallel Programming, Pages 155-166

[Dijkstra1975] E.Dijkstra (1975), *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*, Communications of the ACM, Volume 18, Issue 8, Pages 453-457

[Dotsenko2004] Y. Dotsenko, C. Coarfa and J. Mellor-Crummey (2004), *A Multi-Platform Co-Array Fortran Compiler*, Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, pages 29 - 40

[Floyd1967] R.Floyd (1967) *Assigning Meanings to Programs*, Symposium on Applied Mathematics, Volume 19, Pages 19-31

[Fortune1978] S.Fortune and J.Wyllie (1978), *Parallism in Random Access Machines*, Tenth Annual ACM Symposium on Theory of Computing, Pages 114-118

[Foster1997] I.Foster, J.Geisler, C.Kesselmanz and S.Tuecke (1997), *Managing Multiple Communication Methods in High-Performance Networked Computing Systems*, Journal of Parallel and Distributed Computing, Volume 40, Pages 35-48

[Frigo1998] M.Frigo, C.Leiserson and K.Randall (1998), *The Implementation of the Cilk-5 Multithreaded Language*, ACM SIGPLAN Conference on Programming Language Design and Implementation, Volume 33, Issue 5, Pages 212-223

[Frigo1998-2] M.Frigo and S.G.Johnson (1998), *FFTW: An adaptive software architecture for the FFT*, IEEE Conference on Acoustics, Speech, and Signal Processing, Volume 3, Pages 1381-1384

[Frigo1999] M.Frigo (1999), *A Fast Fourier Transform Compiler*, ACM SIGPLAN Conference on Programming Language Design and Implementation, Volume 34, Issue 5, Pages 169-180

[Gropp1999] W.Gropp, E.Lusk and A.Skjellum (1999), *Using MPI - 2nd Edition*, MIT Press, 0-262-57132-3

[Gropp2005] W.Gropp (2005), *How to Replace MPI as the Programming Model of the Future*, Frontiers of Extreme Computing 2005/Zettaflops Workshop, Santa Cruz, CA October 23-27

[Gusciora1995] G.Gusciora, R.Leibensperger and B.Barney (1995), *MPI Matrix Multiply*, http://www.hku.hk/cc/home/facilities/sp2.htm, *(Last accessed October 2009)*

[Harrison2003] N.Harrison, *Understanding Reflection*, http://www.ondotnet.com/pub/a/dotnet/2003/10/06/reflectionpt1.html, *(Last accessed October 2009)*

[Hill1999] J.Hill and D.B.Skillicorn (1999), *Practical Barrier Synchronisation*, 6th EuroMicro Workshop on Parallel and Distributed Processing, Pages 438-444

[Hill1998-2] J.Hill, B.McColl, D.Stefanescu, M.Goudreau, K.Lang, S.Rao, T.Suel, T.Tsantilas and R.Bisseling (1999), *BSPlib: The BSP programming library*, Parallel Computing, Volume 24, Issue 14, Pages 1947-1980

[Hilfinger2005] P.Hilfinger, D.Bonachea, K.Datta, D.Gay, S.Graham, B.Liblit, G.Pike, J.Su and K.Yelick (2005), *Titanium Language Reference Manual*, Technical Report U.C. Berkeley, CSD-01-1163

[Hinsen2009] K. Hinsen (2009), *The Promises of Functional Programming*, Computing in Science and Engineering, July/August 2009, Pages 86 90

[Hoare1969] C.A.R.Hoare (1969) *An Axiomatic Basis for Computer Programming*, Communications of the ACM, Volume 12, Issue 10, Pages 576-580

[Hook2005] B. Hook (2005), *Write Portable Code: A Guide to Developing Software for Multiple Platforms*, No Starch Press, 978-1593270568

[HPF1997] High Performance Fortran Forum (1997), *High Performance Fortran Language Specification*, Technical Report of Rice University, CRPC-TR92225

[Jensen1991] K.Jensen and N.Wirth (1991), *PASCAL - User Manual and Report*, Springer-Verlag, 0-387-97649-3

[Jones2007] T. Jones, *Linux and Symmetric Multiprocessing*, http://www.ibm.com/developerworks/library/l-linux-smp/, *(Last accessed October 2009)*

[Kernighan1989] B.Kernighan and D.Ritchie (1989), *The C Programming Language*, Prentice Hall Software Series, 0-13-110362-8

[Lavoie1996] P. Lavoie (1996), *A high-speed CMOS implementation of the Winograd Fourier transformalgorithm*, IEEE Transactions on Signal Processing, Volume 44, Issue 8, Pages 2121 - 2126

[Leslie2005] M.Leslie, *C Programming Reference*, http://www.space.unibe.ch/, *(Last accessed October 2009)*

[Luo2002] Y.Luo (2002), *Parallel and Distributed Computing*, http://www.cs.gsu.edu/ cscyip/csc4310/, *(Last accessed October 2009)*

[Lou2005] E.Lou, M.Vanter and L.Votta (2005), *Can Software Engineering Solve the HPCS Problem?*, Second International Workshop on Software Engineering for High Performance Computing System Applications, Pages 27-31

[Luecke1997] G. Luecke and J. Coyle (1997), *High Performance Fortran Versus Explicit Message Passing On The ISB SP-2*, Technical Report Iowa State University

[Meunier1997] J.A.Meunier (1997), *Function Currying in Scheme*, Technical Report University of Connecticut

[Milner1993] R.Milner (1993), *The polyadic pi-calculus: a tutorial*, Technical Report University of Edinburgh, ECS-LFCS-91-180

[Mozafari2008] B. Mozafari, A. Agarwal, N. Laptev and N. Gayam *ZPL - Parallel Programming Language* http://www.nikolaylaptev.com/master/classes/, *(Last accessed October 2009)*

[MPI1995] Message Passing Interface Forum (1995), *A Message-Passing Interface Standard*, Message Passing Interface Forum, Technical Report University of Tennessee, UT-CS-94-230

[MPI1995-2] Message Passing Interface Forum (1995), *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, Message Passing Interface Forum, Parallel Computing, Volume 22, Issue 6, Pages 789 - 828

[Numrich1998] R.W.Numrich and J.K.Reid (1998), *Co-Array Fortran for parallel programming*, ACM SIGPLAN Fortran Forum, Volume 17, Issue 2, Pages 1-31

[Numrich2003] R.W.Numrich (2003), *Co-Array Fortran What is it? Why should you put it on BlueGene/L?*, https://asc.llnl.gov/computing_resources/bluegenel, *(Last accessed October 2009)*

[Pacheco1996] P.S.Pacheco (1996), *Parallel programming with MPI*, Morgan Kaufmann Publishers Inc, 1-55860-339-5

[Pierce1997] B.Pierce (1997), *Foundational Calculi for Programming Languages*, The Computer Science and Engineering Handbook 1997, Pages 2190-2207

[Prasad2003] S.Prasad and S.Arun-Kumar (2003), *An Introduction to Operational Semantics*, Compiler Design Handbook: Optimizations and Machine Code, Chapter 22

[Reynolds1989] J.C.Reynolds (1989), *Theories of Programming Languages*, The Press Syndicate of the University of Cambridge, 0-521-59414-6

[Richardson1996] H.Richardson (1996), *High Performance Fortran: history, overview and current developments*, Technical Note of Thinking Machines Corporation, TMC-261

[Saphir1996] W.Saphir, R.van der Wijngaart, A.Woo and M.Yarrow (1996), *New Implementations and Results for the NAS Parallel Benchmarks 2*, 8th SIAM Conference on Parallel Processing for Scientific Computing, March 14-17, 1997

[Sil1999] SIL International, *What is syntax?*, http://www.sil.org/lingualinks/literacy/, *(Last accessed October 2009)*

[Skillicorn1998] D.B.Skillicorn (1998), *Models and Languages for Parallel Computation*, ACM Computing Surveys, Volume 30, Issue 2, Pages 123-169

[Skillicorn1999] D.B.Skillicorn, J.Hill and W.F.McColl (1999), *Questions and answers about BSP*, Scientific Programming, Volume 6, Issue 3, Pages 249-274

[Slonneger1994] K.Slonneger and B.L.Kurtz (1994), *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, Addison Wesley, 0-201-65697-3

[Springel2005] V.Springel, *Gadget*, http://www.mpa-garching.mpg.de/gadget/, *(Last accessed October 2009)*

[Springel2006] V.Springel (2006), *Summer school on cosmological numerical simulations 3rd week MONDAY*, Helmholtz School of Astrophysics

[Steele2008] M.Steele (2008), *A Tale of Two Algorithms: Multithreading Matrix Multiplication*, www.cilk.com *(Last accessed October 2009)*

[Tennent1976] R.D.Tennent (1976), *The Denotational Semantics of Programming Languages*, Communications of the ACM, Volume 19, Issue 8, Pages 437-453

[Terry1986] P.D.Terry (1986), *Programming language translation : a practical approach*, Addison-Wesley, 0-201-18040-5

[Wagner2000] D.Wagner, J.S.Foster, E.A.Brewer and A.Aiken (2000), *A First Step towards Automated Detection of Buffer Overrun Vulnerabilities*, Network and Distributed System Security Symposium, Pages 3-17

[Wang2007] Y.Wang, W.Wang, C.Huang (2007), *Enhanced Semantic Question Answering System for e-Learning Environment*, Advanced Information Networking and Applications Workshops, Volume 2, Issue 21-23 May 2007 Pages 1023-1028

[Wikipedia] Wikipedia, *Wikipedia*, http://www.wikipedia.org, *(Last accessed October 2009)*

[Winskel1993] G.Winskel (1993), *The Formal Semantics of Programming Languages*, Foundations Of Computing Series, 0-262-23169-7

[Wirth1974] N.Wirth (1974), *On the Design of Programming Languages*, IFIP Congress 1974, Pages 386-393

[Yanagawa2004] T.Yanagawa and K.Suehiro (2004), *Software system of the earth simulator*, Parallel Computing, Volume 30, Issue 12, Pages 1315-1327

[Yelick2002] K.Yelick (2002) *Global Address Space Languages*, http://titanium.cs.berkeley.edu/, *(Last accessed October 2009)*

[Zhou2005] J.Zhou and Y.Chen (2005), *Generating C code from LOGS specifications*, 2nd International Colloquium on Theoretical Aspects of Computing, Volume 3722/2005, Pages 195-210

[Zhu2005] Y. Zhu, X. Li, Y. Gong and Z. Wang (2005), *PN-based Formal Modeling and Verification for ASIP Architecture*, Lecture Notes in Computer Science, Volume 3605/2005, Pages 203-209

Appendix A

Additional Language Specification

A.1 Pre Processor

<u>**%combine**</u>

Syntax

%combine [sourcefile]

Semantics

Will read in the Mesham source file specified and will automatically put it all into the program at the point combine was in the source, before the source code of the original file.

Example

```
%combine a.mesh
%combine b.mesh
```

After preprocessing the file will look like the contents of a.mesh, followed by b.mesh and then the code in the file. Mesham will look in the current directory for the files, but extra directories can be specified by arguments to the preprocessor.

<u>**%use**</u>

Syntax

%use [sourcefile]

Semantics

The sourcefile is read, its global variables and functions can be referenced however the contents of that file is NOT compiled. Instead the linker will link against that (compiled) source file in the final stages. This not only speeds up compilation, it also means that different program modules can be written and modified without having to recompile the whole program.

A.2 Function Library

As detailed in Chapter 3, the programmer can create their own functions within source code. In addition the Mesham language comes with a number of common functions in built. The language's function library is split into six sections, these are **Maths**, **Input/Output**, **Parallelism**, **Bits**, **String** and **System**. This section shall provide an informal definition of the language's in built functions.

A.2.1 Maths

<u>**cos**</u>

This cos[n] function will find the cosine of the value or variable n passed to it.

Pass

A double or float to find cosine of

Returns

A double representing the cosine

Example

```
var a:=cos[10];
var y;
y:=cos[a];
```

floor

This floor[n] function will find the largest integer less than or equal to n.

Pass

A double or float to find floor of

Returns

A double representing the floor

Example

```
var a:=floor[10.5];
var y;
y:=floor[a];
```

getprime

This getprime[n] function will find the nth prime number.

Pass

An integer

Returns

An integer representing the prime

Example

```
var a:=getprime[10];
var y;
y:=getprime[a];
```

log

This log[n] function will find the logarithmic value of n.

Pass

An integer

Returns

A double representing the logarithmic value

Example

```
var a:=log[10];
var y;
y:=log[a];
```

mod

This mod[n,x] function will divide n by x and return the remainder.

Pass

Two integers

Returns

An integer representing the remainder

Example

```
var a:=mod[7,2];
var y;
y:=mod[a,a];
```

neg

This neg[n] function will return the result of negating n.

Pass

An integer to negate

Returns

An integer representing the negated result

Example

```
var a:=neg[15];
var y;
y:=neg[a];
```

negsin

This negsin[n] function will return the result of negating the sine of n.

Pass

A double or float to find the sine value of and then negate

Returns

An double representing the result

Example

```
var a:=negsin[15];
var y;
y:=negsin[a];
```

pi

This pi[] function will return PI. *Note: The number of significant figures of PI is implementation specific.*

Pass

None

Returns

A double representing PI

Example

```
var a:=pi[];
```

pow

This pow[n,x] function will return n to the power of x.

Pass

Two integers

Returns

An integer representing the result

Example

```
var a:=pow[2,8];
```

randomnumber

This randomnumber[n,x] function will return a random number between n and x. *Note: A whole number will be returned UNLESS you pass the bounds of 0,1 and in this case a decimal number is found.*

Pass

Two integers defining the bounds of the random number

Returns

A double representing the random number

Example

```
var a:=randomnumber[10,20];
var b:=randomnumber[0,1]
```

In this case, a is a whole number between 10 and 20, whereas b is a decimal number

sqr

This sqr[n] function will return the result of squaring n.

Pass

An integer to square

Returns

An integer representing the squared result

Example

```
var a:=sqr[10];
```

sqrt

This sqrt[n] function will return the result of square rooting n.

Pass

An integer to find square root of

Returns

A double which is the square root

Example

```
var a:=sqrt[8];
```

A.2.2 Input/Output

closefile

This closefile[n] function will close the file represented by handle n.

Pass

A file handle of type File

Returns

Nothing

Example

```
var f:=openfile["myfile.txt","r"];
closefile[f];
```

input

This input[n] function will ask the user for input via stdin, the result being placed into n.

Pass

A variable for the input to be written into, of type String

Returns

Nothing

Example

```
var f:String;
input[f];
print[f,"\n"];
```

openfile

This openfile[n,a] function will open the file of name n with mode of a.

Pass

The name of the file to open type String and mode type String

Returns

A file handle of type File

Example

```
var f:=openfile["myfile.txt","r"];
closefile[f];
```

print

This print[n] function will display n to stdout. The programmer can pass any number of values or variables split by ,

Pass

A variable to display

Returns

Nothing

Example

```
var f:="hello";
var a:=23;
print[f," ", a ," 22\n"];
```

<u>**readchar**</u>

This readchar[n] function will read a character from a file with handle n. The file handle maintains its position in the file, so after a call to read char the position pointer will be incremented.

Pass

The file handle to read character from

Returns

A character from the file type Char

Example

```
var a:=openfile["hello.txt","r"];
var u:=readchar[a];
closefile[a];
```

<u>**readline**</u>

This readline[n] function will read a line from a file with handle n. The file handle maintains its position in the file, so after a call to readline the position pointer will be incremented.

Pass

The file handle to read the line from

Returns

A line of the file type String

Example

```
var a:=openfile["hello.txt","r"];
var u:=readline[a];
closefile[a];
```

writetofile

This writetofile[n,a] function will write the values of a to the file denoted by handle n.

Pass

The file handle to write to (type File) and also the value (any time) to write into file

Returns

Nothing

Example

```
var a:=openfile["hello.txt","r"];
writetofile[a,"hello - test"];
var q:=19;
writetofile[a,q];
closefile[a];
```

A.2.3 Parallelism

pid

This pid[] function will return the current processes' ID number.

Pass

Nothing

Returns

An integer representing the current process ID

Example

```
var a:=pid[];
```

processes

This processes[] function will return the number of processes

Pass

Nothing

Returns

An integer representing the number of processes

Example

```
var a:=processes[];
```

A.2.4 Bits

bitreverse

This bitreverse[d,n] function will bit reverse the data held in d up to the number of elements n.

Pass

Data to bit reverse and an integer to of the number of elements held

Returns

Nothing

A.2.5 String

charat

This charat[s,n] function will return the character at position n of the string s.

Pass

A string and integer

Returns

A character

Example

```
var a:="hello";
var c:=charat[a,2];
```

lowercase

This lowercase[s] function will return the lower case result of string or character s.

Pass

A string or character

Returns

A string or character

Example

```
var a:="HeLlO";
var c:=lowercase[a];
```

strlen

This strlen[s] function will return the length of string s.

Pass

A string

Returns

An integer

Example

```
var a:="hello";
var c:=strlen[a];
```

substring

This substring[s,n,x] function will return the string at the position between n and x of s.

Pass

A string and two integer

Returns

A string which is a subset of the string passed into it

Example

```
var a:="hello";
var c:=substring[a,2,4];
```

toint

This toint[s] function will convert the string s into an integer.

Pass

A string

Returns

An integer

Example

```
var a:="234";
var c:=toint[a];
```

tostring

This tostring[n] function will convert the variable or value n into a string.

Pass

An element type (i.e. Integer, Float, Char, Double)

Returns

A string

Example

```
var a:=234;
var c:=tostring[a];
```

uppercase

This uppercase[s] function will return the upper case result of string or character s.

Pass

A string or character

Returns

A string or character

Example

```
var a:="HeLlO";
var c:=uppercase[a];
```

A.2.6 System

<u>**ccode**</u>

This ccode[code,library,headers] function will embed the native C code represented by a for execution. No error checking is performed on Ccode, use at own risk!

Pass
A string representing the C code (can be over multiple lines), optional library to link to, optional headers to link to. Strings must be delimited.
Returns
Nothing

Example

```
ccode["int a=23;a++;"];
ccode["char * data=malloc(sizeof(char) * 10);
sprintf(data,\"hello %d\",21);
","","<stdlib.h>"];
```

Note in the second ccode, how quotation marks " inside of the code require delimiting.

<u>**collectgarbage**</u>

This collectgarbage[] function will collect any garbaged data. This is commonly used with string handing, where often the strings are dereferenced and so it is important to maintain a list of them to avoid memory leaks. It should be noted that this is often performed automatically and as such calling this function manually by the programmer is really not all that important.

Pass

Nothing

Returns

Nothing

displayepoch

This displayepoch[] function will display the number of seconds and milliseconds since the epoch (1st January 1970).

Pass

Nothing

Returns

Nothing

displaytime

This displaytime[] function will display the timing results recorded by the function recordtime[] along with the process ID. This is very useful for debugging or performance testing.

Pass

Nothing

Returns

Nothing

recordtime

This recordtime[] function record the current execution time upon reaching that point. This is useful for debugging or performance testing, the time records can be displayed via displaytime[].

Pass

Nothing

Returns

Nothing

exit

This exit[] function will cease program execution and return to the operating system. From an implementation point of view, this will return EXIT_SUCCESS.

Pass

Nothing

Returns

Nothing

oscli

This oscli[a] function will pass the command line interface (e.g. Unix or MS DOS) command to the operating system for execution.

Pass

A string representing the command

Returns

Nothing

Example

```
var a:String;
input[a];
oscli[a];
```

The above program is a simple interface, allowing the user to input a command and then passing this to the OS for execution.

quicksortascending

The quicksortascending function will perform quicksort on an array or list of reference records to order them in an ascending manner. For arrays, pass in the array and it will simply do the quicksort with reference to the data. For reference records, pass in the head (1st record), the comparator field and the linking field.

Pass

For an array, just the array. For a reference record, the first record, the comparator field and the linking field.

Returns

Nothing *(the function modified the array/record passed in.)*

Example

```
quicksortascending[head,"startkey","next"];
```

This is an exert of Gadget-2 Mesham domain decomposition. It will quicksort the list (starting with node head), via the startkey, and linking each node using its next member.

quicksortdescending

Same as Quicksortascending except it will order the data descending rather than ascending.

Appendix B

Matrix Multiplication Code Examples

B.1 MPI

```
#include "mpi.h"
#include <stdio.h>
#define NRA 62                  /* number of rows in matrix A */
#define NCA 15                  /* number of columns in matrix A
    */
#define NCB 7                   /* number of columns in matrix B
    */
#define MASTER 0                /* taskid of first task */
#define FROM_MASTER 1           /* setting a message type */
#define FROM_WORKER 2           /* setting a message type */

int main(argc,argv)
int argc;
char *argv[];
{
int     numtasks, taskid, numworkers, source, dest, mtype,rows,
    averow, extra, offset, i, j, k, rc;
double  a[NRA][NCA],            /* matrix A to be multiplied */
```

```
        b[NCA][NCB],           /* matrix B to be multiplied */
        c[NRA][NCB];           /* result matrix C */
MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   ************************** master task ***********************************/
   if (taskid == MASTER)
   {
      for (i=0; i<NRA; i++)
         for (j=0; j<NCA; j++)
            a[i][j]= i+j;
      for (i=0; i<NCA; i++)
         for (j=0; j<NCB; j++)
            b[i][j]= i*j;
      /* send matrix data to the worker tasks */
      averow = NRA/numworkers;
      extra = NRA%numworkers;
      offset = 0;
      mtype = FROM_MASTER;
      for (dest=1; dest<=numworkers; dest++)
      {
         rows = (dest <= extra) ? averow+1 : averow;
         MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,MPI_COMM_WORLD);
         MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
         offset = offset + rows;
        }
        /* wait for results from all worker tasks */
        mtype = FROM_WORKER;
        for (i=1; i<=numworkers; i++)
        {
            source = i;
            MPI_Recv(&offset, 1, MPI_INT, source, mtype,
                MPI_COMM_WORLD, &status);
            MPI_Recv(&rows, 1, MPI_INT, source, mtype,
                MPI_COMM_WORLD, &status);
            MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source,
                mtype, MPI_COMM_WORLD, &status);
        }
}
/*************************** worker task
    ***********************************/
    if (taskid > MASTER)
    {
        mtype = FROM_MASTER;
        MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype,
            MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,
             &status);
        MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype,
            MPI_COMM_WORLD, &status);
        MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype,
            MPI_COMM_WORLD, &status);

        for (k=0; k<NCB; k++)
            for (i=0; i<rows; i++)
            {
                c[i][k] = 0.0;
                for (j=0; j<NCA; j++)
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
            }
        mtype = FROM_WORKER;
        MPI_Send(&offset, 1, MPI_INT, MASTER, mtype,
            MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD)
            ;
        MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype,
            MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Listing B.1: Matrix Multiplication example in C with MPI from [Gusciora1995]

B.2 BSP

```
#include "bsp.h"
#include <stdio.h>
#include <stdlib.h>

#define matrixasize 3
#define matrixbsize 3

......

void addition(int *res, int *left, int *right, int *nbytes)
{
        *res = 0;
        *res = *left + *right;
}

int main(void) {
```

```
bsp_begin(bsp_nprocs());
int mypid=bsp_pid();
int numprocs=bsp_nprocs();

int * matrixa=malloc(sizeof(int) * matrixasize * matrixasize);
int * matrixb=malloc(sizeof(int) * matrixbsize * matrixbsize);
int * matrixanswer=malloc(sizeof(int) * matrixasize *
    matrixbsize);

if (mypid==0) {filla(&matrixa); fillb(&matrixb);}

bsp_push_reg(matrixa,sizeof(int) * matrixasize * matrixasize);
bsp_push_reg(matrixb,sizeof(int) * matrixbsize * matrixbsize);
bsp_push_reg(matrixanswer,sizeof(int) * matrixasize *
    matrixbsize);
bsp_sync();

bsp_get(0,matrixa,0,matrixa,sizeof(int) * matrixasize *
    matrixasize);
bsp_get(0,matrixb,0,matrixb,sizeof(int) * matrixbsize *
    matrixbsize);
bsp_sync();

int * result=malloc(sizeof(int) * matrixasize * matrixbsize);
int i;
for (i=0;i<matrixasize;i++)
{
    int j;
    for (j=0;j<matrixbsize;j++)
    {
      matrixanswer[(i * matrixasize) + j] = matrixa[(i *
         matrixasize) + mypid] * matrixb[(mypid *matrixbsize)
        + j];
```

```
            bsp_fold(addition,&matrixanswer[i * matrixasize + j],&
                result[i * matrixasize + j],sizeof(int));
        }
    }

    bsp_sync();
    bsp_end();
    return 0;
}
```

Listing B.2: Matrix Multiplication example in C with BSP

B.3 Cilk

```
void matrix_multiply_5(matrix_t A, matrix_t B, matrix_t C,
            int i0, int i1, int j0, int j1, int k0, int k1)
{
      int di = i1 - i0;
      int dj = j1 - j0;
      int dk = k1 - k0;
      if (di >= dj && di >= dk && di >= THRESHOLD) {
        int mi = i0 + di / 2;
        cilk_spawn matrix_multiply_5(A, B, C, i0, mi, j0, j1, k0,
            k1);
        matrix_multiply_5(A, B, C, mi, i1, j0, j1, k0, k1);
        cilk_sync;
} else if (dj >= dk && dj >= THRESHOLD) {
    int mj = j0 + dj / 2;
    cilk_spawn matrix_multiply_5(A, B, C, i0, i1, j0, mj, k0, k1);
    matrix_multiply_5(A, B, C, i0, i1, mj, j1, k0, k1);
    cilk_sync;
} else if (dk >= THRESHOLD) {
    int mk = k0 + dk / 2;
// N.B. It's not safe to use a spawn here. Fun exercise: try
    putting
// it in and then running Cilkscreen to detect the resulting
    race.
    matrix_multiply_5(A, B, C, i0, i1, j0, j1, k0, mk);
    matrix_multiply_5(A, B, C, i0, i1, j0, j1, mk, k1);
} else {
// The problem is now small enough that we can just do things
    serially.
for (int i = i0; i < i1; ++i) {
  for (int j = j0; j < j1; ++j) {
    for (int k = k0; k < k1; ++k)
     C[i][j] += A[i][k] * B[k][j];
       }
    }
  }
}
```

Listing B.3: Matrix Multiplication example in Cilk from [Steele2008]

B.4 High Performance Fortran

```
PROGRAM ABmult
IMPLICIT NONE
INTEGER, PARAMETER :: N = 100
INTEGER, DIMENSION (N,N) :: A, B, C
INTEGER :: i, j

!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C

!HPF$ ALIGN A(i,*) WITH C(i,j)
!      replicate copies of row A(i,*)
!       onto processors which compute C(i,j)

!HPF$ ALIGN B(*,j) WITH C(i,j)
!       replicate copies of column B(*,j))
!       onto processors which compute C(i,j)
        A = 1
        B = 2
        C = 0
        DO i = 1, N
          DO j = 1, N
!            All the work is local due to ALIGNs
            C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
          END DO
        END DO
        WRITE(*,*) C
        END
```

Listing B.4: Matrix Multiplication example in HPF from [Luo2002]

B.5 Co Array Fortran

```
real,dimension(n,n)[p,*] :: a,b,c

do k=1,n
  do q=1,p
c(i,j) = c(i,j) + a(i,k)[myP, q]*b(k,j)[q,myQ]
  enddo
enddo
```

Listing B.5: Matrix Multiplication example in CAF from [Numrich2003]

B.6 ZPL

```
program Summa;

config var
  default_size : integer = 4;
  n            : integer = 4;
  iters        : integer = 1;

region
  RA   = [1..m, 1..n];
  RB   = [1..n, 1..p];
  RC   = [1..m, 1..p];
  FCol = [1..m, *];
  FRow = [*, 1..p];

var
  A      : [RA]   double;
  B      : [RB]   double;
  C      : [RC]   double;
  Aflood : [FCol] double;
  Bflood : [FRow] double;

procedure Summa();
var
  i         : integer;
  it        : integer;
[RC]    begin

            for it := 1 to iters do
              C := 0.0;                      -- zero C
              for i := 1 to n do
[FCol]          Aflood := >>[,i] A;          -- flood A col
[FRow]          Bflood := >>[i,] B;          -- flood B row
```

```
            C += (Aflood * Bflood);  -- multiply
          end;
        end;

        if (verbose) then
          writeln("C is:\n",C);
        end;
      end;
```

Listing B.6: Matrix Multiplication example in ZPL from [Chamberlain1998]

B.7 NESL

```
function matrix_multiply(A,B) =
  {{sum({x*y: x in rowA; y in columnB})
     : columnB in transpose(B)}
    : rowA in A} $

A = [[1., .5],[.5, 1.]];
B = [[1., 1.5],[1.5, 1.]];
matrix_multiply(A,B);
```

Listing B.7: Matrix Multiplication example in NESL from [Blelloch1995]

B.8 Titanium

```
public static void matMul( double [2d] a,
                           double [2d] b,
                           double [2d] c ) {
   foreach(ij in c.domain()) {
      double [1d] aRowi = a.slice(1, ij[1]);
      double [1d] bColj = b.slice(2, ij[2]);
      foreach(k in aRowi.domain()) {
            c[ij] += aRowi[k] * bColj[k];
        }
    }
}
```

Listing B.8: Matrix Multiplication example in Titanium from [Yelick2002]

Appendix C

Case Study Codes

C.1 Mandlebrot C-MPI Code

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define hxres 10000
#define hyres 10000
#define itermax 1000
#define magnify 1
void computehystartendpoints(int ,int *, int *);
int main(int  argc,char * argv[])
{
        MPI_Init(&argc,&argv);
        int myrank, processes;
        MPI_Comm_size(MPI_COMM_WORLD,&processes);
        MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
        int hy,hx,hystart,hyend;
        int * startpoints=malloc(sizeof(int) * processes);
        int * endpoints=malloc(sizeof(int) * processes);
        computehystartendpoints(processes,startpoints,endpoints)
            ;
        hystart=startpoints[myrank];
        hyend=endpoints[myrank];
        int * mydata=malloc(sizeof(int) * (hxres + 1) * ((hyend
           - hystart) + 1) * 3);

        for (hy=hystart;hy<=hyend;hy++)
        {
                for (hx=1;hx<=hxres ;hx++)
                {
                        int blue,red,green;
                        blue=0;
                        red=0;
                        green=0;
                        double cx = (((float)hx)/((float)hxres)
                           -0.5)/magnify*3.0-0.7;
                        double cy = (((float)hy)/((float)hyres)
                           -0.5)/magnify*3.0;
                        double x,y;
                        x = 0.0; y = 0.0;
                        int tempit;
                        int iteration;
                        for (iteration=1;iteration<itermax;
                           iteration++)
                        {
                                double xx = x*x-y*y+cx;
                                y = 2.0*x*y+cy;
                                x = xx;
                                if (x*x+y*y>100.0)  {tempit=
                                   iteration; iteration =
                                   999999;}
                        }
                        if (iteration > 999998)
                        {
                                blue=(tempit * 10) + 100;
                                red=(tempit * 3) + 50;
                                green=(tempit * 3)+ 50;
                                if (tempit > 25)
                                {
                                        blue=0;
                                        red=(tempit * 10);
                                        green=(tempit * 5);
                                }
                                if (blue > 255) blue=255;
                                if (red > 255) red=255;
                                if (green > 255) green=255;
                        }
                        mydata[(((hy - hystart) * hxres) + hx)
                           * 3] = red;
                        mydata[(((hy -hystart) * hxres) + hx) *
                           3 + 1] = green;
                        mydata[(((hy -hystart) * hxres) + hx) *
                           3 + 2] = blue;
                }
        }

        int * collecteddata;
        if (myrank==0)
        {
                collecteddata=malloc(sizeof(int) * hxres * hyres
                    * 3);
        }
        int * recvsize=malloc(sizeof (int) * processes);
        int * displacements=malloc(sizeof (int) * processes);
        int i;
        for (i=0;i<processes;i++)
        {
                recvsize[i]=hxres * (endpoints[i]-startpoints[i
                    ]) * 3;
                if (i==0)
                        displacements[i]=0;
                else
                        displacements[i]=displacements[i -1] +
                            recvsize[i - 1];
        }
        MPI_Gatherv(mydata, hxres * (hyend - hystart) * 3,
            MPI_INT, collecteddata, recvsize, displacements,
            MPI_INT, 0, MPI_COMM_WORLD);
        free(mydata);
        if (myrank==0)
        {
                FILE * openfile;
                openfile=fopen("picture.ppm\0","w");
                fprintf(openfile,"P6\n# CREATOR: mandel program\
                    n");
                fprintf(openfile,"%d %d\n255\n",hxres,hyres);
                int hx,hy,eachp;
                for (hy=0;hy<hyres;hy++)
                {
                        for (hx=0;hx<hxres;hx++)
                        {
                                fputc((char) collecteddata[((hy*
                                    hxres) + hx) * 3],openfile);
                                fputc((char) collecteddata[((hy*
                                    hxres) + hx) * 3 + 1],
                                    openfile);
                                fputc((char) collecteddata[((hy*
                                    hxres) + hx) * 3 + 2],
                                    openfile);
                        }
                }
                fclose(openfile);
                free(collecteddata);
        }
        MPI_Finalize();
        return 0;
}

void computehystartendpoints(int size,int * startpoints,int *
   endpoints)
{
        int individualhy=hyres / size;
        int uneven=0;
        if (individualhy * size != hyres) uneven=hyres -
           individualhy * size;
        int i;
        for (i=0;i<size;i++)
        {
                if (i==0)
                        startpoints[i]=0;
                else
                        startpoints[i]=endpoints[i-1]+1;
                int unevendistributer=0;
                if (uneven > 0) {unevendistributer=1; uneven--;}
                endpoints[i]=startpoints[i] + individualhy - 1 +
                    unevendistributer;
        }
}
```

Listing C.1: Mandlebrot C with MPI code

C.2 Mesham NAS-IS benchmark Code

```
%combine c.mesh

/* NUMBER OF PROCESSORS */
var processors:=128;

var bucket:referencerecord["keystart",Int,"keyend",Int,"next",
    bucket,"size",Int,"globalsize",Int,"id",Int];
var numbuckets:Int;
var totalkeys:Int;
var numkeys:Int;
var maxkey:Int;
var totnumbuckets:Int;
var maxiteration:=10;

function void main[]
{
        numbuckets:=1 << numbucketslog2;
        totalkeys:=1 << totalkeyslog2;
        maxkey:=1 << maxkeylog2;
        numkeys:=totalkeys % processors;
        totnumbuckets:=numbuckets + testarraysize;

        var testindexarray:array[Int,testarraysize] :: allocated
            [multiple[]];
        var testrankarray:array[Int,testarraysize] :: allocated[
            multiple[]];

        fillTestArray[testindexarray,testrankarray];
        var numbers:array[Int,numkeys] :: allocated[multiple[]];

        var one:Long;
        var two:Long;
        var three:Long;
        var four:Double;
        var five:Double;
        one:=pid[];
        two:=processors;
        three:=4 * totalkeys;
        four:=314159265.00;
        five:=1220703125.00;
        var theseed:=findseed[one,two,three,four,five];
        createseq[theseed,five,numbers];
        var indexbuckets:array[bucket,totnumbuckets] ::
           allocated[multiple[]];
        var head:bucket :: allocated[multiple[]];
        initBuckets[head,indexbuckets];

        var tempbuffer:array[Int,maxkey] :: allocated[multiple
           []];
        var bucketsizes:array[Int,totnumbuckets] :: allocated[
           multiple[]];
        var globalbucketsizes:array[Int,totnumbuckets] ::
           allocated[multiple[]];
        var keycollection:array[Int,numkeys] :: allocated[
           multiple[]];
        var p;
        par p from 0 to processors - 1
        {

                rank[numbers,testindexarray,testrankarray,1,head
                   ,indexbuckets,tempbuffer,bucketsizes,
                   globalbucketsizes,keycollection]; // free
                   iteration to setup
                if (p==0) recordtime[];
                var i;
                for i from 1 to maxiteration
                {
                        rank[numbers,testindexarray,
                           testrankarray,i,head,indexbuckets,
                           tempbuffer,bucketsizes,
                           globalbucketsizes,keycollection];
                };
                if (p==0)
                {
                        recordtime[];
                        displaytime[];
                };
        };
};
function void rank[var numbers, var testindexarray, var
   testrankarray, var iteration,var head,var indexbuckets,var
   tempbuffer,var bucketsizes,var globalbucketsizes,var
   keycollection]
{
        numbers:array[Int,numkeys] :: allocated[multiple[]];
        testindexarray:array[Int,testarraysize] :: allocated[
           multiple[]];
        testrankarray:array[Int,testarraysize] :: allocated[
           multiple[]];
        iteration:Int :: allocated[multiple[]];
        indexbuckets:array[bucket,totnumbuckets] :: allocated[
           multiple[]];
        head:bucket :: allocated[multiple[]];

        tempbuffer:array[Int,maxkey] :: allocated[multiple[]];
        bucketsizes:array[Int,totnumbuckets] :: allocated[
           multiple[]];
        globalbucketsizes:array[Int,totnumbuckets] :: allocated[
           multiple[]];
        keycollection:array[Int,numkeys] :: allocated[multiple
            []];
        var p:=pid[];
        if (p==0)
        {
                (numbers#iteration):=iteration;
                (numbers#(iteration + maxiteration)):=maxkey -
                    iteration;
        };
        insertTestArray[indexbuckets,numbers,testindexarray,
            testrankarray];

        var bucketinterval:=maxkeylog2 - numbucketslog2;
            // compute a nice interval, which is the key range in
             each bucket
        var i;
        for i from 0 to numkeys - 1
        {
                var thenum:=(numbers#i);
                var bucketnum:=thenum >> bucketinterval;
                ((indexbuckets#bucketnum).size):= ((indexbuckets
                    #bucketnum).size) + 1;
        };
        ((indexbuckets#0).keystart):=0;
        ((indexbuckets#0).keyend):=0;
        for i from 1 to numbuckets - 1
        {
                ((indexbuckets#i).keystart):=((indexbuckets#(i -
                     1)).keystart) + ((indexbuckets#(i - 1)).size
                    );
                ((indexbuckets#i).keyend):=((indexbuckets#i).
                    keystart);
        };

        for i from 0 to numkeys - 1
        {
                var thenum:=(numbers#i);
                var bucketnum:=thenum >> bucketinterval;
                (keycollection#((indexbuckets#bucketnum).keyend)
                   ):=thenum;
                ((indexbuckets#bucketnum).keyend):=((
                   indexbuckets#bucketnum).keyend) + 1;
        };

        computeGlobalBucketSizes[indexbuckets, bucketsizes,
           globalbucketsizes];

        var bucketsumglobal:=0;
        var processor:=0;
        var processorbuckets:array[Int, processors + 1] ::
           allocated[multiple[]];

        (processorbuckets#0):=0;
        for i from 0 to numbuckets - 1
        {
                bucketsumglobal:=bucketsumglobal + ((
                   indexbuckets#i).globalsize);
                if (bucketsumglobal >= ((processor + 1) *
                   numkeys))
                {
                        processor:= processor + 1;
                        (processorbuckets#processor):=(i + 1);
                };
        };
        (processorbuckets#processors):=numbuckets;

```

```

    var collectedsize:array[Int,processors] :: allocated[
        multiple[]];
    var commdsp:array[Int,processors] :: allocated[multiple
        []];
    var senddsp:array[Int,processors] :: allocated[multiple
        []];
    var sendsize:array[Int,processors] :: allocated[multiple
        []];
    (senddsp#0):=0;
    var prevsize:=0;
    var j;
    for j from 0 to processors - 1
    {
        if (j > 0) (senddsp#j):=(senddsp#(j - 1)) + (
            sendsize#(j - 1));
        var firstbucket:=(processorbuckets#j);
        var ptr:=0;
        var qq;
        for qq from firstbucket to (processorbuckets#(j
            + 1)) - 1
        {
            var thehead:=(indexbuckets#qq);
            ptr:=ptr + thehead.size;
            thehead.size:=0;
            if (j < pid[]) prevsize:=prevsize + (
                indexbuckets#qq).globalsize;
        };
        (sendsize#j):=ptr - 1;
    };
    (collectedsize::alltoall[1]):=sendsize;

    var totcolsize:=0;
```

```
        (commdsp#0):=0;
        for j from 0 to processors - 1
        {
                if (j > 0) (commdsp#j):=(commdsp#(j - 1)) + (
                   collectedsize#(j - 1));
                totcolsize:=(collectedsize#j) + totcolsize;
        };

        var collected:array[Int,totcolsize] :: allocated[
           multiple[]];
        (collected::alltoall[sendsize,collectedsize,senddsp,
           commdsp]):=keycollection;

        var myfirstbucketnum:=(processorbuckets#pid[]);
        var mylastbucketnum:=(processorbuckets#(pid[] + 1)) - 1;
        var myfirstbucket:=(indexbuckets#myfirstbucketnum);
        var mylastbucket:=(indexbuckets#mylastbucketnum);

        var minkeyval:=((myfirstbucket).id) << bucketinterval;
        var maxkeyval:=((((mylastbucket).id) + 1) <<
           bucketinterval);
        maxkeyval:=maxkeyval - 1;

        for i from minkeyval to maxkeyval (tempbuffer#i):=0;
           // clear the work array, so can enter population in
           in a min
        var runningsize:=0;
        for i from 0 to processors - 1
        {
                var recvsize:=(collectedsize#i);
                var j;
                for j from runningsize to (runningsize + (
                   recvsize - 1))
                {
                        var thenum:=(collected#j);
                        (tempbuffer#thenum):=(tempbuffer#thenum)
                            + 1;    // set population of keys
                };
                runningsize:=runningsize + recvsize;
        };

        (tempbuffer#minkeyval):=(tempbuffer#minkeyval) +
            prevsize;
        for i from minkeyval to maxkeyval - 1
        {
                (tempbuffer#(i + 1)):= (tempbuffer#(i + 1)) + (
                    tempbuffer#i);
        };

        pV[indexbuckets,minkeyval,maxkeyval,tempbuffer,
            testindexarray,testrankarray,iteration];
};

function void insertTestArray[var indexbuckets,var numbers,var
    testindexarray,var testrankarray]
{
        indexbuckets:array[bucket,numbuckets] :: allocated[
            multiple[]];
        numbers:array[Int,numkeys] :: allocated[multiple[]];
        testindexarray:array[Int,testarraysize] :: allocated[
            multiple[]];
        testrankarray:array[Int,testarraysize] :: allocated[
            multiple[]];
        var i;
        for i from 0 to testarraysize - 1
        {
                if (((testindexarray#i) % numkeys) == pid[])
                {
                        var ie:=mod[(testindexarray#i), numkeys
                           ];
                        (indexbuckets#(numbuckets + i)).size:=(
                           numbers#ie);
                };
        };
};

function void computeGlobalBucketSizes[var indexbuckets,var
   bucketsizes,var globalbucketsizes]
{
        indexbuckets:array[bucket,totnumbuckets] :: allocated[
           multiple[]];
        bucketsizes:array[Int,totnumbuckets] :: allocated[
           multiple[]];
        globalbucketsizes:array[Int,totnumbuckets] :: allocated[
           multiple[]];
        var i;
        for i from 0 to totnumbuckets - 1
        {
                (bucketsizes#i):=(indexbuckets#i).size;
        };
        (globalbucketsizes::allreduce["sum"]):=bucketsizes;
        for i from 0 to totnumbuckets - 1
        {
                (indexbuckets#i).globalsize:=(globalbucketsizes#
                   i);
        };
};

```

```
function void createseq[var seed:Double, var a:Double, var
    numbers]
{
        numbers:array[Int,numkeys] :: allocated[multiple[]];
        var x:Double;
        var k:=maxkey % 4;
        var i;
        for i from 0 to numkeys - 1
        {
                x:=randlc[seed,a];
                x:= x + randlc[seed,a];
                x:= x + randlc[seed,a];
                x:= x + randlc[seed,a];
                (numbers#i):=k * x;
        };
};

function Double findseed[var kn:Long, var np:Long, var nn:Long,
    var s:Double, var a:Double]
{
        var nq:=nn % np;
        var mq:=0;
        while (nq > 1)
        {
                mq := mq + 1;
                nq:=nq % 2;
        };

        var t1:Double;
        t1:=a;
        var t2:Double;
        var t3:Double;
        var i;
        for i from 1 to mq
        {
                t2:=randlc[t1,t1];
        };
        var an:Double;
        an:=t1;
        var kk:Long;
        kk:=kn;
        t1:=s;
        t2:=an;
        var ik:Long;
        for i from 1 to 100
        {
                ik:=kk % 2;
                if ((2 * ik) != kk) t3:=randlc[t1,t2];
                if (ik == 0) break;
                t3:=randlc[t2,t2];
                kk:=ik;
        };
        return t1;
};

function Double randlc[var x:Double, var a:Double]
{
        var r23:Double;
        var t23:Double;
        var r46:Double;
        var t46:Double;
        var ks:Int;
        r23:=1.0;
        t23:=1.0;
        r46:=1.0;
        t46:=1.0;
        var i;
        for i from 1 to 23
        {
                r23:=0.5 * r23;
                t23:=2.0 * t23;
        };

        for i from 1 to 46
        {
                r46:=0.5 * r46;
                t46:=2.0 * t46;
        };

        ks:=1;

        var t1:Double;
        var t2:Double;
        var t3:Double;
        var t4:Double;

        var a1:Double;
        var a2:Double;
        var j:Int;
        t1:=r23 * a;
        j:=t1;
        a1:=j;
        a2:=a - t23 * a1;

        var x1:Double;
        var x2:Double;
        var z:Double;

        t1:= r23 * x;
        j:=t1;
        x1:=j;
        x2:=x - t23 * x1;
        t1:=a1 * x2 ;
        t1:=t1 + a2 * x1;
        j:=r23 * t1;
        t2:=j;
        z:= t1 - t23 * t2;
        t3:=t23 * z;
        t3:=t3 + a2 * x2;
        j:=r46 * t3;
        t4:=j;
        x:=t3 - t46 * t4;
        return r46 * x;
};

function void pV[var indexbuckets,var minkeyval,var maxkeyval,
    var tempbuffer,var testindexarray ,var testrankarray ,var
    iteration]
{
        indexbuckets:array[bucket,totnumbuckets] :: allocated[
            multiple[]];
        minkeyval:Int :: allocated[multiple[]];
        maxkeyval:Int :: allocated[multiple[]];
        iteration:Int ::allocated[multiple[]];
        tempbuffer:array[Int,maxkey] :: allocated[multiple[]];
        testindexarray:array[Int,testarraysize] :: allocated[
            multiple[]];
        testrankarray:array[Int,testarraysize] :: allocated[
            multiple[]];
        var i;
        for i from 0 to testarraysize - 1
        {
```

```
                    var me:=pid[];
                    var k:=(indexbuckets#(numbuckets + i)).
                        globalsize;
                    if (minkeyval <= k && k <= maxkeyval)
                    {
                              classPVtest[i,k,tempbuffer,testrankarray
                                  ,iteration];
                    };
          };
};

function void initBuckets[var head,var indexbuckets]
{
          indexbuckets:array[bucket,totnumbuckets] :: allocated[
              multiple[]];
          head:bucket :: allocated[multiple[]];
          head:=null;
          var i;
          for i from 0 to totnumbuckets - 1
          {
                    var newhead:bucket :: allocated[multiple[]];
                    newhead.next:=head;
                    newhead.keystart:=0;
                    newhead.keyend:=0;
                    newhead.id:=(totnumbuckets - 1) - i;
                    newhead.size:=0;
                    newhead.globalsize:=0;
                    head:=newhead;
                    (indexbuckets#(totnumbuckets - 1) - i):=head;
          };
};
```

Listing C.2: Mesham NAS-IS benchmark code

C.3 FFT uneven data distribution generated C-MPI

```
{
complex * MESHtempvar;
int MESHblockstoprocesses[]={0,1,2,3,4};
int MESHblockatoprocesses[]={103,103,102,102,102};
int MESHblockbtoprocesses[]={512,512,512,512,512};
int MESHblocksizetoprocesses[]={52736,52736,52224,52224,52224};
int MESHblockdistributiontoprocesses
    [5][5]={{103,103,102,102,102},
{103,103,102,102,102},{103,103,102,102,102},{103,103,102,102,102},

{103,103,102,102,102}};
int MESHblocknum;
for(MESHblocknum=0;MESHblocknum<5;MESHblocknum++) {
if(myrank==MESHblockstoprocesses[MESHblocknum]){
MESHtempvar=(complex *) malloc(sizeof(complex) *
MESHblocksizetoprocesses[MESHblocknum]);
int MESHi,MESHj,MESHd;
for(MESHj=0;MESHj<MESHblockbtoprocesses[MESHblocknum];MESHj++){
for(MESHd=0;MESHd<MESHblockatoprocesses[MESHblocknum];MESHd++){
MESHtempvar[(MESHj * MESHblockatoprocesses[MESHblocknum]) +
MESHd]=A[(MESHd * MESHblockbtoprocesses[MESHblocknum]) + MESHj
    ];}}
int MESHelementcount[5];
int MESHloopvar;
for(MESHloopvar=0;MESHloopvar<5;MESHloopvar++)
{MESHelementcount[MESHloopvar]=(MESHblockatoprocesses[
    MESHblocknum] *
MESHblockdistributiontoprocesses[MESHblocknum][MESHloopvar])
    *2;}
int MESHelementdisplacement[5];
```

```
for(MESHloopvar=0;MESHloopvar<5;MESHloopvar++)
{if (MESHloopvar==0){
MESHelementdisplacement[MESHloopvar]=0;
}else{
MESHelementdisplacement[MESHloopvar]=MESHelementdisplacement[
    MESHloopvar
- 1] + (MESHelementcount[MESHloopvar - 1] /2);}}
int MESHsendcounter;
for(MESHsendcounter=0;MESHsendcounter<5;MESHsendcounter++){
MPI_Request MESHreq;
MPI_Isend(&MESHtempvar[MESHelementdisplacement[MESHsendcounter
    ]],MESHelementcount[MESHsendcounter],MPI_FLOAT,
    MESHsendcounter,23,MPI_COMM_WORLD,&MESHreq);
}}}
for(MESHblocknum=0;MESHblocknum<5;MESHblocknum++) {
if(myrank==MESHblockstoprocesses[MESHblocknum]){
int MESHstartpt=0;
complex * MESHtempvar2=(complex * ) malloc(sizeof(complex) *
MESHblocksizetoprocesses[MESHblocknum]);
int MESHrecvcounter;MPI_Request MESHrequestlist[5];
for(MESHrecvcounter=0;MESHrecvcounter<5;MESHrecvcounter++){
MPI_Irecv(&MESHtempvar2[MESHstartpt],(MESHblockatoprocesses[
    myrank]
*MESHblockdistributiontoprocesses[myrank][MESHrecvcounter])
*2,MPI_FLOAT,MESHrecvcounter,23,MPI_COMM_WORLD,&MESHrequestlist[
    MESHrecvcounter]);
MESHstartpt=MESHstartpt+(MESHblockatoprocesses[myrank]
*MESHblockdistributiontoprocesses[myrank][MESHrecvcounter]);
}MPI_Waitall(5,MESHrequestlist,MPI_STATUSES_IGNORE);
int MESHi,MESHd,MESHj;
int MESHoffset=0;
int MESHmc;
int MESHcurrenta[5];
```

```
for (MESHmc=0;MESHmc < 5;MESHmc++) {
if (MESHmc==0) {
MESHcurrenta[0]=0;
} else {
MESHcurrenta[MESHmc]=MESHcurrenta[MESHmc - 1] +
MESHblockdistributiontoprocesses[MESHblocknum][MESHmc - 1];
}}
for (MESHi=0;MESHi<5;MESHi++) {
int MESHthissize=(MESHblockatoprocesses[MESHblocknum] *
MESHblockdistributiontoprocesses[MESHblocknum][MESHi]);
for (MESHd=0;MESHd<MESHblockatoprocesses[MESHblocknum];MESHd++)
    {
for
(MESHj=0;MESHj<MESHblockdistributiontoprocesses[MESHblocknum][
    MESHi];MESHj++)
{
B[(MESHj + MESHcurrenta[MESHi]) + (MESHd *
MESHblockbtoprocesses[MESHblocknum])]=MESHtempvar2[((MESHd *
MESHblockdistributiontoprocesses[MESHblocknum][MESHi]) + MESHj)
    +
MESHoffset]; }}
MESHoffset=MESHoffset + MESHthissize;}
free(MESHtempvar2);}}
}
```

Listing C.3: Part of generated C-MPI Code with uneven data distribution

C.4 Gadget-2 C code PH key finding

```
tatic int quadrants[24][2][2][2] = {
  /* rotx=0, roty=0-3 */
  {{{0, 7}, {1, 6}}, {{3, 4}, {2, 5}}},
  {{{7, 4}, {6, 5}}, {{0, 3}, {1, 2}}},
  {{{4, 3}, {5, 2}}, {{7, 0}, {6, 1}}},
  {{{3, 0}, {2, 1}}, {{4, 7}, {5, 6}}},
  /* rotx=1, roty=0-3 */
  {{{1, 0}, {6, 7}}, {{2, 3}, {5, 4}}},
  {{{0, 3}, {7, 4}}, {{1, 2}, {6, 5}}},
  {{{3, 2}, {4, 5}}, {{0, 1}, {7, 6}}},
  {{{2, 1}, {5, 6}}, {{3, 0}, {4, 7}}},
  /* rotx=2, roty=0-3 */
  {{{6, 1}, {7, 0}}, {{5, 2}, {4, 3}}},
  {{{1, 2}, {0, 3}}, {{6, 5}, {7, 4}}},
  {{{2, 5}, {3, 4}}, {{1, 6}, {0, 7}}},
  {{{5, 6}, {4, 7}}, {{2, 1}, {3, 0}}},
  /* rotx=3, roty=0-3 */
  {{{7, 6}, {0, 1}}, {{4, 5}, {3, 2}}},
  {{{6, 5}, {1, 2}}, {{7, 4}, {0, 3}}},
  {{{5, 4}, {2, 3}}, {{6, 7}, {1, 0}}},
  {{{4, 7}, {3, 0}}, {{5, 6}, {2, 1}}},
  /* rotx=4, roty=0-3 */
  {{{6, 7}, {5, 4}}, {{1, 0}, {2, 3}}},
  {{{7, 0}, {4, 3}}, {{6, 1}, {5, 2}}},
  {{{0, 1}, {3, 2}}, {{7, 6}, {4, 5}}},
  {{{1, 6}, {2, 5}}, {{0, 7}, {3, 4}}},
  /* rotx=5, roty=0-3 */
  {{{2, 3}, {1, 0}}, {{5, 4}, {6, 7}}},
  {{{3, 4}, {0, 7}}, {{2, 5}, {1, 6}}},
  {{{4, 5}, {7, 6}}, {{3, 2}, {0, 1}}},
  {{{5, 2}, {6, 1}}, {{4, 3}, {7, 0}}}
};


static int rotxmap_table[24] = { 4, 5, 6, 7, 8, 9, 10, 11,
  12, 13, 14, 15, 0, 1, 2, 3, 17, 18, 19, 16, 23, 20, 21, 22
};
```

```

static int rotymap_table[24] = { 1, 2, 3, 0, 16, 17, 18, 19,
  11, 8, 9, 10, 22, 23, 20, 21, 14, 15, 12, 13, 4, 5, 6, 7
};

static int rotx_table[8] = { 3, 0, 0, 2, 2, 0, 0, 1 };
static int roty_table[8] = { 0, 1, 1, 2, 2, 3, 3, 0 };

static int sense_table[8] = { -1, -1, -1, +1, +1, -1, -1, -1 };

static int flag_quadrants_inverse = 1;
static char quadrants_inverse_x[24][8];
static char quadrants_inverse_y[24][8];
static char quadrants_inverse_z[24][8];


/*! This function computes a Peano-Hilbert key for an integer
    triplet (x,y,z),
 *  with x,y,z in the range between 0 and 2^bits-1.
 */
peanokey peano_hilbert_key(int x, int y, int z, int bits)
{
  int i, quad, bitx, bity, bitz;
  int mask, rotation, rotx, roty, sense;
  peanokey key;


  mask = 1 << (bits - 1);
  key = 0;
  rotation = 0;
  sense = 1;


  for(i = 0; i < bits; i++, mask >>= 1)
    {
      bitx = (x & mask) ? 1 : 0;
      bity = (y & mask) ? 1 : 0;
      bitz = (z & mask) ? 1 : 0;

      quad = quadrants[rotation][bitx][bity][bitz];

      key <<= 3;
      key += (sense == 1) ? (quad) : (7 - quad);

      rotx = rotx_table[quad];
      roty = roty_table[quad];
      sense *= sense_table[quad];

      while(rotx > 0)
        {
          rotation = rotxmap_table[rotation];
          rotx--;
        }

      while(roty > 0)
        {
          rotation = rotymap_table[rotation];
          roty--;
        }
    }

  return key;
}
```

Listing C.4: Part of Gadget-2 peano hilbert key finding

C.5 Gadget-2 C code BHTree building

```
void domain_topsplit_local(int node, peanokey startkey)
{
  int i, p, sub, bin;

  if(TopNodes[node].Size >= 8)
    {
      TopNodes[node].Daughter = NTopnodes;

      for(i = 0; i < 8; i++)
        {
          if(NTopnodes < MAXTOPNODES)
            {
              sub = TopNodes[node].Daughter + i;
              TopNodes[sub].Size = TopNodes[node].Size / 8;
              TopNodes[sub].Count = 0;
              TopNodes[sub].Daughter = -1;
              TopNodes[sub].StartKey = startkey + i * TopNodes[
                  sub].Size;
              TopNodes[sub].Pstart = TopNodes[node].Pstart;

              NTopnodes++;
            }
          else
            {
              printf("task=%d: We are out of Topnodes.
                  Increasing the constant MAXTOPNODES might help
                  .\n",
                     ThisTask);
              fflush(stdout);
              endrun(13213);
            }
        }

      for(p = TopNodes[node].Pstart; p < TopNodes[node].Pstart +
          TopNodes[node].Count; p++)
        {
          bin = (KeySorted[p] - startkey) / (TopNodes[node].Size
              / 8);

          if(bin < 0 || bin > 7)
            {
              printf("task=%d: something odd has happened here.
                  bin=%d\n", ThisTask, bin);
              fflush(stdout);
              endrun(13123123);
            }

          sub = TopNodes[node].Daughter + bin;

          if(TopNodes[sub].Count == 0)
            TopNodes[sub].Pstart = p;

          TopNodes[sub].Count++;
        }

      for(i = 0; i < 8; i++)
        {
          sub = TopNodes[node].Daughter + i;
          if(TopNodes[sub].Count > All.TotNumPart / (
              TOPNODEFACTOR * NTask * NTask))
            domain_topsplit_local(sub, TopNodes[sub].StartKey);
        }
    }
}
```

Listing C.5: Part of Gadget-2 building BHTree

Appendix D

Tabular Performance Data

D.1 Mandelbrot

Processors	Mesham (secs)	C-MPI (secs)
10	41.89	41.76
16	29.09	29.30
20	23.62	23.59
32	14.62	15.25
64	7.82	8.06

Table D.1: Mandelbrot Timing Results

D.2 NAS-IS Class B

Processors	Mesham (secs)	NAS (secs)
1	8.18	6.49
2	4.29	3.64
4	2.51	2.79
8	1.79	2.21
16	1.55	1.50
32	7.8	8.79
64	31.85	37.18
128	89.64	97.07

Table D.2: NAS-IS Class B Timing Results

D.3 NAS-IS Class C

Processors	Mesham (secs)	NAS (secs)
1	33.38	26.58
2	19.7	16.96
4	12.72	8.76
8	8.82	15.41
16	6.39	16.23
32	13.73	14.36
64	15.04	17.48
128	51.4	62.39

Table D.3: NAS-IS Class C Timing Results

D.4 NAS-IS Total Mop/s

Processors	Mesham-B	NAS-B	Mesham-C	NAS-C
1	41.04	51.68	40.21	50.5
2	78.15	92.18	68.13	79.12
4	133.51	120.41	105.54	153.16
8	187.11	151.83	152.12	87.08
16	216.02	224.19	210.04	82.7
32	43.04	38.19	97.78	93.44
64	10.53	9.03	89.24	76.8
128	3.74	3.46	26.11	21.51

Table D.4: NAS-IS Total Mop/s

D.5 NAS-IS Mop/s per Process

Processors	Mesham-B	NAS-B	Mesham-C	NAS-C
1	41.04	51.68	40.21	50.5
2	39.08	46.09	34.07	39.56
4	33.38	30.1	26.39	38.29
8	23.39	18.98	19.01	10.88
16	13.5	14.01	13.13	5.17
32	1.34	1.19	3.06	2.92
64	0.16	0.14	1.39	1.2
128	0.03	0.03	0.2	0.17

Table D.5: NAS-IS Mop/s per Process

D.6 FFT on 128MB Data

Processors	Mesham	FFTW	Pacheco
1	4.94	4.13	6.59
2	2.28	3.68	4.44
4	1.67	1.97	3.03
8	1.05	1.25	2.76
10	0.94	1.57	-
16	0.86	1.05	3.67
20	0.8	4.19	-
32	0.91	3.49	4.04
64	1.28	4.41	5.82

Table D.6: FFT on 128MB Data

D.7 FFT on 2GB Data

Processors	Mesham	FFTW
2	158.95	81.4
4	126.34	42.08
8	23.2	23.15
10	21.05	43.57
16	16.24	17.68
20	14.12	18.17
32	12.01	13.07
64	9.87	10.06

Table D.7: FFT on 2GB Data