A Service-Oriented Architecture and Language for Abstracted Distributed Algorithms

A Service-Oriented Architecture and Language for Abstracted Distributed Algorithms

Daniel Goodman

Worcester College

Oxford University Computing Laboratory

Submitted for the degree of Doctor of Philosophy

A story for those who

always loved me

Abstract

This thesis describes a new programming model designed to provide an intuitive and efficient way of abstracting the partitioning of distributed input data when programming large, dynamic, distributed, parallel, computing environments. It then describes an implementation of this model as a workflow language (Martlet) and a supporting prototype middleware, as well as providing a range of case studies demonstrating the power of this model. Having introduced this model and implementing language, it concludes by describing how this can be of use in the wider scope of such dynamic environments by using just in time compilers to apply Martlet to mainstream middleware and to improve the utilisation of Condor Pools and Clusters.

Inspired by the inductive constructs of Functional Programming this programming model was designed to address the issue of how to allow users to write well abstracted programs in a so far poorly explored part of a new scenario presented by *Grid Computing*, where the location and partitioning of data is truly dynamic, and not under the control of the user. In this environment, unlike previous computing environments, datasets and computing resources are routinely split across a number of locations and organisations, and the nature of this split can often only be determined at runtime.

While many projects have been quick to embrace the idea of the Grid Computing platform, they have done so using the same basic programming models that they used in more traditional environments. As such they are restricted by assumptions made in these models that are no longer valid. These assumptions prevent programmers using the full potential of this new computing platform. The specific assumption removed through the programming model presented here is that the data is in a known and constant number of pieces when the program is constructed. This solves a common but often unrecognised problem in many eResearch projects, however, this problem must be solved before eResearch can reach its full potential. This realisation can already be seen in the growing number of workshops, conferences, and drives appearing in an attempt to describe programming models for this new computing platform.

Contents

List of Figures

Acknowledgements

Over the course of the years since I started this research I have gained an extensive list of people to whom I owe a huge amount. I shall attempt to thank them here, but to those I fail to name, my apologies and gratitude. Firstly, my supervisors, Andrew Martin and Raphael Hauser for their support, guidance, and patience when I lost my way. Bernard Sufrin and Andrew Simpson for both their friendship and advice. I would also like to thank everyone in Software Engineering, Climate*Prediction*.net and the Computing Laboratory for their help, as well as the Natural Environmental Research Council for funding this work.

Thanks also needs to go to my friends, specifically Evelyn Davies, Josephine Adkin, Douglas Creager, Rebecca Hoath, Kjell Konis, Bruno Oliveira, Sam Rice, Ed Smith, Gail Stevens, Rui Zhang, and the many people in Worcester College and the Gliding Club for their help proof reading my work, their friendship, support and most importantly willingness to provide a means of escape when it all became too much.

Finally I must thank my family, without whom I would have never come close to starting, never mind completing. Thank you for everything.

Chapter 1

Introduction

In this thesis, we describe a new programming model designed to provide an intuitive and efficient way of abstracting the partitioning of distributed input data when programming large, dynamic, distributed, parallel computing environments. We then describe an implementation of this model as a workflow language (Martlet) [59, 60] and a supporting prototype middleware, as well as providing a range of case studies demonstrating the power of this model. Having introduced our model and implementing language, we then describe how this can be of use in the wider scope of such dynamic environments to improve utilisation.

Inspired by the inductive constructs of Functional Programming [12], this programming model was designed to address the issue of how to allow users to write well-abstracted programs in the context of a so far poorly explored part of a new scenario presented by *Grid Computing* [104], where the location and partitioning of data is truly dynamic and not under the control of the user. Grid Computing is an umbrella term used to describe recent projects that aim to enable *utility computing* [103], that is, to make high performance computing a utility like other utilities such as electricity for scientists and engineers to use. In this environment, unlike previous computing environments, datasets and computing resources are routinely split across a number of locations and organisations, and the nature of this split can often only be determined at runtime.

While many projects have been quick to embrace the idea of this new computing platform, they have done so using the same basic programming models that were used in more traditional environments. As such, they are restricted by assumptions made in these models that are no longer valid. These assumptions prevent programmers from using the full potential of this new computing platform. The specific assumption removed through the programming model presented here is that the data is in a known and constant number of pieces when the program is constructed.

Although Martlet is currently only supported by a prototype middleware, it is trivially ported to a wide range of middleware through the use of Just In Time (JIT) compilers. Such extensions not only allow this work to be used with many current and future distributed Grid resources, it also creates opportunities to use this style of programming in order to allow higher resource utilisation when processing large jobs in environments such as the Condor pools [77] used in many Campus Grids [126].

In the rest of this chapter, we will first examine the history and motivation of the Grid Computing environment. Then we will describe in more detail the problem this work is intended to solve and introduce the specific project that was first used to define this problem, along with some use case studies from this project. We then finish the chapter with a summary of the structure of the rest of this thesis.

1.1 A Short History

In the 1950s, it was anticipated that computing, much like other utilities, would be provided by centralised facilities and users would just use a terminal to access these resources. However, the processing power of the machines used as terminals increased at a substantially faster rate than the communication networks required to support this model. This resulted in users being in a position where the best way to get sufficient computing power for the tasks they required, with latencies low enough to make the session interactive, was to use their terminals as standalone computers. However, despite most users having sufficient computing power on their desks, there remained a large class of problems that required larger amounts of processor power and storage. These problems were run on specialised hardware, or supercomputers. As this hardware was specialised, the cost was higher per FLOP than that of the mass-produced computers used by most people. This made such resources the preserve of business, the military, and select groups at research institutions.

In the late 1960s and the 70s, frustrated by the restricted access to such computer resources, the Advance Research Projects Agency (ARPA) constructed the ARPANET to allow people to gain access to remote computers. This was the first step back towards the original utility model envisaged a decade or so before. This network continued to expand, becoming the heterogeneous environment we now know as the Internet. The increasingly common nature of networks linked by the Internet fuelled a new wave of development of distributed computing projects in the decades that followed.

This new wave of research has resulted in the development of a staggering number of technologies to improve the interoperability of heterogeneous systems. By 2000, business and research organisations were starting to look back on the utility model of computing in the light of this new interoperability, and the high-speed networks that were now available. This has led several businesses, including Sun, HP, and IBM, to attempt to construct the scalable, user-friendly technologies required for centrally controlled utility computing centres. These attempts have met with differing levels of success. One of the less anticipated challenges is convincing people to change the way they work to use these resources now they're available. With time, however, their use has started to become more common.

People also started looking at what other resources it is possible to make available as a utility. Such resources include datasets, scientific equipment, specialist data analysis programs, and data storage, to name a few. To conduct experiments using such resources it is necessary to group them together with the required computing power for the experiment. The required complementary resources for experiments may vary greatly from experiment to experiment, as may the resources available to fulfil these requirements. It is this scenario that causes the drive for another model of computing, Grid Computing. Grid Computing is a model for a decentralised, fault tolerant, utility computing service [52, 53, 94]. The main difference between this and other utility computing models is the removal of centralised control and ownership of a system. The intention is that virtual organisations (VOs) can be dynamically constructed through the interoperability of independent systems owned by different real organisations. The hope is that it will enable better collaboration on research through the sharing of computing resources, scientific equipment and data.

This model offers potentially huge benefits to application scientists using the Grid infrastructure to access the resources to perform previously unthinkably complex calculations. The interest to computer scientists lies with the complexities of building such distributed interoperability in a well-abstracted and functional way on an entirely new computational model. To help encourage the development of these technologies, and the benefits they could bring, the UK government, like many others, has under the label of e-Science, started the "UK e-Science Programme". This is a five-year, £250 million program that has funded over one hundred projects. These projects are working on issues such as resource allocation, job scheduling, security and workflow languages, as well as the use of Grid infrastructure such as BOINC [2] to create super clusters. In Chapter 2, we look at the different views that are put forward about how to implement a Grid, and discuss some of the resources that are available for creating Grid computing environments.

1.2 Scenario

The dynamic grouping of resources and the interoperability of such resources in Grid computing creates an interesting new scenario for computing models and the languages for handling them. Historically data from a computation would always end up in a predetermined number of pieces, and furthermore these pieces would normally be within the same computer system, for Grid applications, this is no longer necessarily true. Data for a computation may be split across many resources, both because the computation draws from many separately hosted datasets, and because individual datasets may no longer be stored in a single piece, but span many resources. This situation is further complicated as the size of the datasets is increasing all the time, making it less and less feasible to move all the data to a single location for analysis. Assuming that the dataset is too large to move to a single location, there are three options for performing analysis on such a dataset:

1. The user determines the distribution of the dataset. They then construct analysis functions designed for this distribution, adjusting the functions every time the distribution of the data changes.

2. The user writes code for handling the distribution of the data into their functions, including the code for handling errors. This adds a huge potential for user mistakes, and results in a large increase in the required user understanding of the system. Because of the high level of abstraction reached by most workflow languages, it may not even be possible to describe such functions in a wide range of languages.

3. Make the language and underlying middleware capable of abstracting such classes of problems, so allowing users to construct functions that work with arbitrarily distributed datasets without knowing specific runtime information about the dataset in question.

Clearly the third option is the preferred choice, however, as will be shown in Chapter 2, there are currently no languages in this domain that provide this style of programming model. Further, the papers Workflow Patterns [123] and Workflow Data Patterns [109], which are widely used to rate the level of functionality provided by different workflow languages both fail to identify suitable patterns for performing this style of programming model.

The aim of this project is to design, implement, and test a language and middleware for a handling arbitrarily partitioned data. To demonstrate the effectiveness of the work developed by this project, we are working with the e-Science funded project climate*prediction*.net [116] (CPDN). The CPDN project, inspired by the success of the SETI@home [3] project, is using spare computing power on desktop computers to generate a vast dataset of possible climate predictions. To date, this is the world's largest climate experiment, having modelled over 22 million years of climate across over 200,000 different climate models, producing terabytes of distributed data. This dataset is generated using donated computing power in a similar manner to the analysis performed by SETI@home. Volunteers download and run a client program on their computer. This client program then downloads a climate model and a set of perturbed parameters that CPDN scientists would like to examine the effect of, and runs this model with these perturbed parameters using any spare processor power on the volunteer's computer. At various points during the model's execution the client returns a set of results based on the model's output.

When the results are returned to CPDN, they are stored on one of a set of upload servers. The upload server that the results are stored on is chosen from the set of currently available upload servers at the point that the client returns their results. The choice is made at this point because the is no guarantee that a server chosen at an earlier point will still be available, as in a distributed environment like this it has to be expected that servers will fail and new ones will be added. As such the number of upload servers is not a constant, so when trying to construct any workflow for analysing the returned data a means of handling this changing number of servers is required. This situation is made more complex by the fact that scientists using the returned runs will in general not want to analyse the whole dataset, but will instead want to analyse subsets of the dataset. While these subsets will still be too big to move to a single location, the number of upload servers spanned by these subsets can vary from subset to subset. In addition, as this dataset is intended as a resource untrusted users will have access to, it is necessary that the solution allows for appropriate controls over such access.

As discussed above, the preferred situation here is that the underlying middleware would detect this data partitioning and adjust any submitted functions appropriately. To provide a clearer motivation for this work, we are going to use two functions that CPDN scientists would like to perform as examples through out this thesis. These are the *mean*, and the *singular value decomposition*. While more detail about both of these examples, and other case studies, can be found in Chapter 5, we introduce these two now.

1.2.1 Mean

While the mean of a set of parameters within a set of models is a useful operation in its own right, it is also an important part of many more complex operations such as the North Atlantic Oscillation described in Chapter 5. The mean of a set of runs split into a separate distributed pieces $[x_0 \dots x_{n_1}), [x_{n_1} \dots x_{n_2}), \dots, [x_{n_{a-1}} \dots x_{n_a})$ can be calculated using the equations:[1]

$$\begin{tikzcd}
 & & \boxed{\overline{x} = \frac{\sum_{i=0}^{a-1} y_i}{\sum_{i=0}^{a-1} z_i}} & \\
\boxed{\begin{array}{l} y_0 = \sum_{i=0}^{n_1-1} x_i \\ z_0 = n_1 \end{array}} \arrow[urr, bend left] & \boxed{\begin{array}{l} y_1 = \sum_{i=n_1}^{n_2-1} x_i \\ z_1 = n_2 - n_1 \end{array}} \arrow[ur, bend left] \arrow[rr, dotted, no head] & & \boxed{\begin{array}{l} y_{a-1} = \sum_{i=n_{a-1}}^{n_a-1} x_i \\ z_{a-1} = n_a - n_{a-1} \end{array}} \arrow[ul, bend right]
\end{tikzcd}$$

Here, x_0 through x_{n_a-1} represent the runs distributed across the a servers. The y values then represent the local sums, while the z values keep track of how many runs are included in each y. These partial results are then combined in the final equation to give the overall average.

These can be trivially described and executed across existing technologies and systems, if the value of a is known. However, if the value of a changes, as is the case with CPDN datasets, then this cannot be automatically handled; this leaves it up to the scientist to deal with this on a case by case basis.

1.2.2 Singular Value Decomposition

The Singular Value Decomposition (SVD) is one of the more complex operations that CPDN scientists would like to be able to perform across their dataset. This operation takes a matrix A,

[1] These equations assume that there will be no issues with number overflow.

and factorises it into three matrices U, Σ and V^T such that:

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n}$$

where U and V are orthogonal and Σ is a monotonically decreasing nonnegative diagonal matrix. The importance of this operation is that the diagonal values of Σ are the singular values of A, and the column vectors of V are the corresponding singular vectors. So the first p column vectors of V and values from Σ define the p dimension subspace of $\mathbb{R}^n$ that produces the biggest expansion under the transformation defined by the matrix A. While it would not be realistic to calculate a complete SVD of the dataset, as this would have a significantly larger space requirement than the original dataset, generating just the first few vectors provides a range of the useful information about the nature of the transformation defined by the matrix A. In the case of the CPDN data, this will tell us which combination of the perturbed parameters makes the most significant difference to the climate model and the parameters sensitivity.

Again, it has not been possible with existing rough-grained or workflow languages to describe this function in a way that can be executed in a distributed environment without prior knowledge of the distribution of the dataset. Over the course of this thesis we will show how Martlet overcomes these problems and successfully abstracts them away from the user. More details of the different algorithms and their corresponding Martlet scripts can be found in Chapter 5.

1.3 Thesis Outline

In Chapter 2 we examine existing projects and ideas that are related to this work in the fields of functional programming, parallel computing and distributed computing, especially Service-Oriented Web Service and Grid Service projects.

Then, in Chapter 3 we then revisit how the new dynamic environment creates a style of computation that is different to those met previously, before going on to examine our solution, Martlet. Martlet is a workflow language that implements a novel programming model able to tackle these new issues. It achieves this by using two different classes of data structures and a set of constructs inspired by functional programming. These allow it to adapt at runtime to differently partitioned datasets, creating an environment where users can write functions that will be adapted to work with the partitioning of data on which they are called.

In Chapter 4 we discuss the specification of a Service-Oriented framework to support the execution of Martlet programs over a distributed set of data nodes. This specification is then used to construct a prototype Web Service based middleware. The specification and construction of this middleware illustrates the need to address issues handled by other middleware systems, and so leads to the discussion in Chapter 6 on how existing projects can be extended to handle this new programming model.

Chapter 5 examines a range of real world case studies, demonstrating both how existing algorithms can be converted to work on the Martlet programming model, and showing how, when this is not possible, it is often possible to generate new algorithms to perform the required functions. This opens up the interesting possibility of a whole class of algorithms just waiting to be discovered once people start thinking in terms of this new programming model. In addition to demonstrating the descriptive power of Martlet, this chapter allows us to examine its performance on a distributed test bed of computers.

Then in Chapter 6 evaluates this work against the other technologies discussed in Chapter 2 and discusses how Martlet might be extended to support a wider range of existing and future systems. This discussion includes the possibility of extending the classes of constructs to allow more complex interactions, and the construction of a partial evaluator from JIT compilers. Such a partial evaluator would allow this middleware and language to be used in conjunction with other Grid and Web Service middleware. We also examine the possibility of using this model to allow better utilisation of heterogeneous computing resources where the number of resources changes.

Finally in Chapter 7 we conclude this thesis by reiterating our main contributions are. We then think about how these contributions fit into the area covered both by “Grid Computing”, and parallel computing in general. We then finish with some final thoughts on how this work is one of the steps required if distributed utility computing, under its many labels, is to break free of the programming models used in more conventional environments and to reach its full potential.

Chapter 2

Related Work

In this chapter, we will examine some of the existing technologies that are relevant to this thesis. While they are discussed here, and some comments are made as to their suitability or otherwise for this project, we will defer details about which technologies are used to inspire and implement Martlet and its supporting middleware until the relevant chapters.

2.1 Functional Programming

Functional programming [12] as a programming model was first created in the 1950s, building on work on Lambda Calculus, done by Alonzo Church in the 1930s. In this model programs are pure mathematical functions, these take their input as arguments and call other pure mathematical functions on parts of this to construct their output. This is repeated until the functions become simple enough that they can return base types. As these functions are purely mathematical, they do not alter *state*, as is the case with imperative programming; instead all values are immutable. As there is no mutable state, there is no assignment. This also means constructs such as `while` statements in imperative programming are not possible in functional programming. Instead such functionality is provided through the use of recursion.

So far this discussion has listed the things that functional programming does not have, we will now discuss what functional programming is able to gain through the loss of these features.

Because functions cannot modify any state, they, unlike functions in imperative languages, functions in functional programming languages cannot have any side effects, as such their output is deterministic. This occurs because there is a guarantee that no value used by a function can change after it has been calculated. This means all values used in function calls with the same inputs are the same as all externally referenced values are immutable. Imperative languages are unable to offer this guarantee.

As there is a guarantee that there are no side effects and no state modification, the output of the program is not dependant on the order that the program is executed in. This allows programs to be evaluated using techniques such as lazy evaluation [70], where functions are only computed at the point that their result is required. This also makes it possible to guarantee that programs are thread safe.

Another distinct property of functional programs is that functions are treated as first class variables. This means it is possible to pass functions into other functions as arguments, allowing the construction of very generic functions.

Collectively these features allow this style of programming to describe some extremely powerful programs. This is often achieved using only a few lines of code. So for instance a function to reverse a list can be written in Haskell [12] as;

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

This uses pattern matching to determine which of the two clauses to execute. If the function is passed an empty list denoted by `[]`, this is its base case and it will return an empty list. If, on the other hand, the function is passed an element, `x`, concatenated to a list, `xs`, it will recursively call `reverse` on the list `xs`, and concatenate the original element `x` to the end of the reversed list. An example that takes advantage of the first class nature of functions is `map`. This is a function that applies another function, `f`, to every element of a list, and can be written as;

```
map f [] = []
map f (x:xs) = (f x) :  map f xs
```

again using pattern matching to determine between the base and inductive cases.

The properties of this programming model, including specifically recursion over lists and functions as first class variables, inspire the data partitioning independent workflow language (Martlet) and its associated programming model described in this thesis.

2.2 Parallel Computing

Parallel computing [129] is one way of providing increased processing power, and refers to computing platforms that are able to perform more than one instruction at a time. This is distinct from the various forms of multi-threaded platform where the user is given the impression of being able to run processes in parallel, but the underlying hardware is in fact only allowing threads to take turns using the computing resources. There are many different types of parallel computing platform, varying from multi-core processors, to multiple processors sharing a single memory, to multiple processors with their own resources connected by a range of transport mechanisms. In all cases, there is a constant, known number of processors that can be synchronised to differing degrees. In the latter case the transport mechanism can vary from interlinking inside specialist hardware, to low latency switches in Beowulf clusters [117], with connectivity structures described by many different patterns including trees, arrays, hypercubes, and rings. These different architectures have distinctly different properties, with inter-processor communication often being a bottleneck. As such there are many different algorithms for solving a given problem in order to minimise the effect of the restrictions on data transfer. For example, [10, 21, 22, 23, 131, 133] are all papers describing different implementations of the same technique for solving the Singular Value Decomposition (SVD) problem on different parallel platforms.

2.2.1 Parallel Imperative Languages

These different and restricted platforms have led to the construction of platform-specific languages that aim to make the use of such platforms both more intuitive and more efficient. These languages can achieve this by allowing the user to split the operations into distinct types that the platform will perform. Such types typically would include:

parallel These are steps which are executed on each processor independently. For instance, when multiplying a matrix by a vector, if there are sufficient processors, then each processor will perform just one dot product. Therefore, each processor will run the same code and the same steps, just on different data.

inter-processor These steps allow the transfer of information between processes in a user specified manner. These steps usually occur as a synchronized communication, as is the case in the SVD examples. This can be done via an API such as Message Passing Interface (MPI) [113].

reduction These are steps included explicitly within the programming languages for reducing a set of values into a single value. They tend to do this using techniques such as the parallel prefix method [74]. However, due to the fine-grained nature of such languages, these steps are normally restricted to just primitive operations.

Programming languages for non-distributed parallel platforms are capable of describing a large class of algorithms, but their focus is, and will remain, performance within a very well-defined domain. As such, these languages remain fine-grained, and rely on a detailed knowledge of the platform they will run on. For instance the MasPar [80] language, which runs on an array of processors, requires that every processor in the array has a unique, sequential integer, and that the programmer includes as an attribute in the program stating the number of processors in the array. To execute this code on a machine with a different size array requires the attribute to be updated and the code recompiled.

Languages such as OCCAM [71, 73] and PCN [49] provide a more abstract way of describing parallel problems. These are in many ways inspired by the Communicating Sequential Processes language algebra (CSP) [108] and as such also look much like CSP. These languages, like their functional programming counterparts, provide a means by which users can build deterministic programs. This is done through the use of provably deterministic constructs to compose smaller deterministic components into larger deterministic components. OCCAM was designed for programming transputer microprocessors, though there where compilers produced for other languages, because of its domain specific nature it includes the ability to describe components within the language. PCN, on the other hand, is intended as a higher-level language that is used for chaining together components written in other languages such as C and Fortran. In both cases, compiled programs are architecture specific, and need recompiling and potentially updating when the target machine changes. From the point of view of these languages and the aim of this project, a different partitioning of the data would constitute a change of architecture.

Yet another attempt to improve parallel programming is Sun's latest language for scientific computing, Fortress [1]. Fortress is designed with the intention of allowing the user to write programs in terms of more conventional mathematical notation. This notation will then be converted into parallel code where possible. To help facilitate this, the compiler, libraries and language provide fast thread spawning and mapping as an implicit construct.

2.2.2 Data Parallel Languages

First developed in the 1980s as a way of programming vector machines, Data-Parallel Computing [13] is an alternative view on the construction of parallel programs, where the parallelisation is derived from the data structures instead of being inserted through explicit instruction from the user. This offers a lot of promise for making programming of parallel machines easier because data is often naturally parallel, whereas in many cases code has to be carefully crafted to make it parallel. Early languages implementing this model of programming include NESL [14], which has gone on to inspire other projects such as Data Parallel Haskell [29]. In these the data structures used to derive this parallelism are described in nested structures where each element in the structure can include some parallel properties. These structures can then be automatically flattened to produce efficient parallel code.

2.2.3 Constraint Logic Programming

Constraint Logic Programming (CLP) is implemented by languages such as Prolog. Under this model, the programmer provides constraints describing the relationship between elements in a data structure. From these constraints, new constraints that provide useful information can be derived. These new constraints are found by searching the space described by the original constraints and any intermediately discovered constraints.

This searching can be split up into individual parts that can be run in parallel. There is work on solving such problems within a distributed heterogeneous environment [83], and methods for communicating partial results. This type of language is thus far poorly addressed by the community, and there may be potential here to describe problems with a high level of abstraction, thereby allowing them to adjust to the available environment with ease. However, even on a single processor, this model does not perform well when handling numerical problems such as the ones we intend to address in this thesis. As such languages using this style of programming are out of scope and will not be examined further.

2.2.4 Coordination Languages

Coordination languages [56] are a means of separating concerns between the communication and coordination parts of a program, and the data processing parts. There are two ways they can be implemented.

One is to take the form of a programming language in its own right. Such a language merely makes a cleaner separation between the code that the programmer needs to change if the underlying architecture changes, and the code that performs the actual algorithm. Workflow languages such as those discussed in Section 2.5 are good examples of such coordination languages, as are the languages used to control programs in Virtual Shared Memory systems.

The other approach is to created parameterised skeletons [39, 40] of common patterns. These can then be parameterised with the appropriate code for the process, allowing the compiler to merge the two parts to create a program for the given architecture. Then if the program needs porting to a different architecture, the compiler can substitute in different code for the skeleton, so yielding truly portable code. This also allows for the construction of accurate timing profiles for each architecture. This of course comes at the cost of the level of description available when compared with the first option.

2.2.5 Virtual Shared Memories

Virtual Shared Memories (VSM) are a means by which parallel programs can be constructed without having to be aware of the structure of the hardware that they are going to be executed on. Instead, the programs just add and remove key value pairs from the VSM, and the middleware/compiler-constructed code deals with the data transfer. This allows for easier construction and management of distributed data and distributed data structures, at the expense of having code that is hand crafted for the architecture that it is going to run on. This can be overcome with differing degrees of success by clever compilers, but there is potential for serious performance issues.

With such programming environments, the language used is normally a language such as C, Java, Fortran, or Python, with additional constructs added to handle the passing of data. Because these languages all implement more traditional programming models and the VSM hides entirely the distribution of the data, using this model it is very hard to construct programs that will adjust to the distribution of an unknown number of pieces of data.

Examples of the coordination languages used to harness VSMs are Linda [111] and Paradise [110], which then run on top of middlewares such as NetWorkSpaces [112] or Piranha [110], when operating in an environment such as a network of computers, otherwise they use architecture specific compilers to add the code to handle these additional requirements.

2.3 Distributed Computing Models

The ability to take advantage of potentially many remote computing resources and datasets in addition to any local resources, coupled with the growing speed of networks, makes distributed computing an appealing aim for many large projects [19]. In this section we will examine a range of different models for distributed computing before examining some of the technologies available to implement these different models in the next section.

Remote Procedure Calls The simplest form of distributed computing first started in the 1980s with Remote Procedure Calls (RPC) [35], and these are still the underlying basis of many other middleware, implementing different styles of distributed computation. With RPC, the raw data is passed in full as a message to a procedure on a remote computing resource. The result is later returned in the same fashion. This has advantages in that it is simple to implement, and is practical for small datasets where the latency of the communication medium is a significant part of the communication time. However, as the size of the dataset increases, this method of distributed computing becomes impractical. Examples where this is used include actions such as sending email, and HTTP requests.

Distributed Computing with References An alternative approach that alleviates this problem is to pass a reference to the data instead, as supported by WS-Addressing [20]. This can then be used to reference the data and retrieve directly just the required parts. In its simplest form, this reference only points to the location where the data can be found; however, the reference can be extended into a handle that contains metadata as well. Adding handles to the processing side of the computation makes it possible to keep track of the data as a process acts it on. This removes the need to perform a synchronisation every time the data is altered. This approach not only helps overcome some of the issues related to data size, it also allows the construction of Object Brokers such as CORBA [89] and RMI [119]. Object Brokers allow the programming models used in Object Oriented Programming to be extended to work over distributed computing resources and are used for applications ranging from telecommunications to distributed databases to booking systems.

Service-Oriented Architecture While middleware implementing Object Brokers can be very well engineered, and have on occasion been very successful, experience has shown that often they encourage the user to create programs that are too closely coupled, often with disastrous effects when components within the system fail. This has led to Service Oriented Architecture (SOA) or Message Base Architecture to be put forward as an alternative model. This is the style implemented by databases using SQL for remote queries. Under this model, it is preferable to keep state in the messages themselves, instead of relying on stateful client server connections. This means that if a service becomes unavailable, it can be replaced dynamically with another service, and the execution continues as before. The key differences between this and RPC are the conceptual view about where state should occur, and that while RPC is only a request response model, SOA is able to have a far more diverse and powerful set of interactions between resources. This is used to chain together separate systems such as stock control and the back-ends of many web sites.

Grid and Utility Computing One newly popular style of distributed computing, using variations on an SOA model, is utility computing [103]. In this model the user submits a job to a system, then all the allocation of computing power, storage, and other resources, will be done dynamically at runtime and their results will be returned to them at a point in the future. All they will have to do is pay for the resources that they use, in a similar manner to other utilities. This leads to the vision of Grid Computing as a decentralised fault tolerant utility computing service [52, 53, 94]. The main difference between the Grid and other utility computing models is the removal of centralised control and ownership of a system. The intention is that virtual organisations (VOs) can be dynamically constructed through the interoperability of independent systems. The aim is to facilitate better collaboration on research through the sharing of computing resources, scientific equipment and data.

The implementation of such a system creates a new set of challenges, which makes the Grid starkly different from other existing distributed computing models. These challenges include creating a set of standardised specifications that can be implemented by different organisations to enable inter-organisation job submissions and security, and decentralised resource brokering and scheduling. A range of models have been put forward for to implement such a system, along with some middleware attempting to implement these models. Exactly how far this model can go is currently the subject of much research and discussion with projects ranging from monitoring jet engines to high energy physics to the humanities.

2.4 Implementing Distributed Computing

To implement the wide range of different models for distributed computing there is an equally wide range supporting technologies, and in this section we will examine some of these.

2.4.1 Representational State Transfer

Representational State Transfer [46] (REST) was first developed between October 1994 and August 1995. Originally referred to as the “HTTP Object Model”, it was later renamed to avoid confusion with implementation models for Web servers. REST then continued to be iteratively refined over the next five years. Fundamentally, it is a set of architectural constraints that attempt to allow the construction of low latency, highly distributed, reliable systems. The biggest example of a REST system is the World Wide Web.

REST does not constrain the components within the system; instead, it constrains the connections that are possible between resources. So for example the four verbs in HTTP that are implemented by all components are *GET*, *POST*, *PUT* and *DELETE*. Resources are then identified by Universal Resource Identifiers (URIs), and it is up to the implementer to determine what their responses will be to each of the four options. This freedom allows an otherwise extremely restrictive system to have many heterogeneous components. Such systems vary from servers returning static content, to scientific equipment returning data, to functions performing analysis on data. While the resources can have state, the client server connections themselves are stateless. This greatly simplifies reasoning about any event in the system.

The main advantage of REST as a style of communication is its simplicity and proven scalability, allowing for the creation of large heterogeneous systems. In addition, as it does not mandate the use of verbose technologies like XML, it can have a much smaller overhead than other more prescriptive technologies. The disadvantage of this model is that, as an interaction should consist of only a single operation, accessing a resource requires all the relevant information to be encoded into a URI, making it hard to define clean interfaces to allow the loose coupling of components, this often results in fragile constructions where components cannot be easily changed.

2.4.2 Web Services

Web Services are one of the latest ideas for distributed computing. As with all popular buzzwords, exactly what a Web Service is depends greatly on who you ask, with answers varying from its just a remote procedure call made using SOAP, to resources made available on the Internet using a stateless SOA [52]. There is much debate as to whether Web Services should be stateless [38, 69, 96], and what exactly it means for a service to be stateless.

For this work, we take the view that a Web Service is any resource made available using SOAP. In addition, we take the view that a Web Service application should be constructed in a loosely coupled manner in line with the SOA principles. We will also, where possible, ensure that our services are stateless; however, if it becomes impractical to maintain this stance due to the data intensive applications such as the ones presented in this project we will be prepared to weaken it.

Using the wide variety of developing technologies, many models have been suggested for constructing Grid applications through the use of Web Services, these include the Open Grid Service Architecture [52, 100] used in the Globus Toolkit 3, the Web Services Resource Framework [38] used in Globus Toolkit 4, the Web Services Grid Application Framework [96], and Asynchronous Service Access Protocol (ASAP) [106]. In addition to these models, there are groupings such as Web Services Interoperability (WS-I) [91] and WS-I+ [8] that define sets of Web Service protocols such as WS-Context [26] and WS-Security [7] that they argue all service which want to be fully interoperable should implement.

We will now examine some of the core technologies used to implement Web Services, then in later sections we will examine some of the mentioned models for building on these.

SOAP

SOAP [125] is a simple, extendable message protocol described using XML. Because of its XML construction and its simple, extendable structure it is very popular for platform independent distributed computing such as Web Services. SOAP messages consist of distinct parts including:

- The SOAP envelope. This consists of a header and a body to contain the message. The body contains the data to be transmitted. It is possible to store in the body nested additional SOAP messages as well as more traditional content. The header describes what the receiving party should do with the message, including the following two items.
- A reference to the definition of the encoding used to serialise this message data into SOAP.
- A reference to the RPC definitions for the case that the receiving party is expected to perform an RPC on the message.

Because of SOAP's extensibility and popularity among distributed computing projects, especially Web Services, a wide range of extensions providing additional functionality for many different applications have been developed [7, 26, 64, 78, 114]. SOAP is extremely descriptive, however, its XML base results in the major drawback that the messages are very verbose. While not a serious issue for small messages, where network latency is a more significant component of the total communication time, for transporting large amounts of data the encoding, decoding and transportation of such messages can become unfeasible [33].

Web Service Description Language

Web Service Description Language [34] (WSDL) is an attempt to formalise an XML-based interface description language to describe the inputs and outputs of Web Services. WSDL is a good first attempt, and has allowed the creation of a wide range of tools that take a WSDL document and produce a stub for binding to Web Services. In addition, a range of tools have been developed to automatically generate WSDL from the code used to support Web Services, or from other mediums such as UML [66]. This makes the construction of WSDL fairly painless for the simpler Web Services.

While WSDL has unquestionably made it easier to bind to Web Services, and has opened up the possibility of automatic binding, it does have two key weaknesses. First, it is unable to describe interaction patterns that are more complex than a simple request-response. Secondly, WSDL is unable to describe all of the structures that can be sent using SOAP. This occurs because, while it is able to describe abstract types, WSDL does not support multiple super types. So while it is possible to write encoders, decoders and SOAP for transmitting structures built using design patterns such as the *composite design* pattern [44], it is not possible to describe this in WSDL if any type in the structure are required to extend two separate super types. In addition, any type that is derived from a class implementing an interface has to have its interface replaced by an abstract class, containing just abstract methods in order to be described using WSDL. While these weaknesses can be overcome by unpleasant hacks, this situation is far from ideal for a technology whose major selling point is the transparency it creates between different components.

SOAP Service Description Language

The SOAP Service Description Language [92, 93] (SSDL) was suggested in 2005 as an alternative to the WSDL, with the aim of solving some of the problems with WSDL as well as encouraging a message orientated approach to Web Service interactions. Unlike WSDL, SSDL is built with the explicit assumption that it is describing interaction carried out using SOAP as the medium. As such, instead of describing the interface that the message will be sent to, it describes the SOAP structure of the message its self. In addition to this change, it also allows for message interactions to be described using a range of higher level descriptions including a small subset of the Communicating Sequential Processes language [108] (CSP). These changes allow the documents to be both clearer and more descriptive, however at the current time there are very few tools supporting SSDL, but it is too early to tell if it will be widely used, or if WSDL is just too dominant.

Universal Description, Discovery and Integration

The Universal Description, Discovery and Integration protocol [86] (UDDI) is the current standard used by registries of Web Services. It is used to provide a level of indirection between the services and the client wishing to use the services. This can be thought of as analogous to the role of DNS within the Internet. It is, however, more powerful than DNS, because instead of just allowing a client to resolve a name to an endpoint, it allows clients to search for a set of endpoints that offer equivalent services.

UDDI simplifies the development of production code, as the only change required when the code goes live is the UDDI server that the service uses. In addition, it is another step in the direction of automatic discovery and linking of Web Services [28].

Axis and Tomcat

Apache Axis [5] and Apache Tomcat [6] are open-source projects run by the Apache Software Foundation. Tomcat is the Servlet container used in the official releases of Java Servlet and Java Servlet Pages. Axis is developed from the Apache SOAP Project, and runs as an application in a Tomcat container, providing a SOAP container for the deployment of Web Services. Together, these two projects aim to provide the best-of-breed Java Web Service container, and have thus far proved to be a highly stable and well-featured platform on which to deploy Web Services. Supported features include the ability to auto generate WSDL, client side stubs, and the ability to chain *Handlers* within the container such that all messages to the service must pass along this chain of Handlers. This allows features such as security implementations, the format of messages, and the transport mechanisms to be adjusted without altering the core service. This, coupled with its open source nature, has resulted in this platform being used by a vast number of both Web Service and Grid projects.

.NET

.NET [36] is Microsoft's multi-language platform, which among its properties has a trivial means to deploy C# functions as Web Services. It also supports all of the complementary functions that you would expect from a full-featured platform. Its main drawback is that currently the only fully supported implementation is produced by Microsoft, and is therefore tied to their operating system and business model. There is some support for .NET on other platforms, most notably the Mono [82] project, which has developed powerful tools for supporting .NET, somewhat alleviating such concerns. However, they are still unable to offer all of the library functions required for true interoperability.

2.4.3 Asynchronous Service Access Protocol

Asynchronous Service Access Protocol(ASAP) [106] is one model for handling the potentially large times between tasks being started and completing, without having problems caused by connections being interrupted or timing out. Under this model, everything is referenced through the use of URIs and all service calls are asynchronous. First, the user calls a factory service to construct an instance of the service they would like to execute, this returns a URI to the constructed service. Once the service is constructed, the user then makes another asynchronous service call to start this service. This second call includes an observer URI that references a user service that will be called when the invoked service terminates. In addition to this, there is another URI that the user can use if they wish to poll their job. This provides a very neat and effective way of managing a long running remote service; however, as it only addresses this issue, it needs complementing with other protocols to represent a complete solution.

2.4.4 Web Service Grid Application Framework

Web Service Grid Application Framework [96] (WS-GAF) is the name of a philosophy put forward in a paper in 2003 by the North East Regional e-Science Centre (NERESC). This had the aim of opening up discussion on the direction the Grid community was taking and whether it could better align with the direction already taken by the Web Services community. The ideas introduced in this paper were then followed up by another paper [95] in 2004, presenting a range of suggested design patterns for security and interoperability.

Their argument is that there is a huge amount of work being done by the Web Services community to produce protocols and supporting tooling for Service Oriented Architectures. The Grid community then uses these underlying technologies to take advantage of existing resources. However, the community failed to observe where the Web Services community is heading. As a result, they fail to think about future interoperability with Web Services. This has led them to augment specifications such as WSDL with additional tags that are required for Grid Services, resulting in the tooling constructed by the Web Services community for these technologies becoming incompatible with the Grid versions.

In addition, while Web Services are stateless, used by many people, and tend to be deployed for extended periods. Many Grid Services had publicly visible state and these where being deployed and disposed of frequently. This was due to an alternative view where each service is used by specific groups of people. Therefore, each different group would have their own instantiation of the service and state. It is argued that this state and associated ability for the user to create and dispose of services, while not forcing an object-oriented style of programming, does encourages such a style. An object-oriented style can then lead to a too closely coupled construction that hinders efforts to allow Grid applications to be deployed with ease across different organisations.

NERESC suggested solution to these problems is instead of modifying the technologies produced for Web Services, to build an application framework on top of these technologies. In doing so, the binary choice between Grid and Web Services will be removed. This will then allow the creation of new services that use take advantage of both Grid and Web Services using workflow languages like BPEL [4].

Probably the most contentious element of this philosophy is the removal of state from service connections. While it is simpler and more stable not to have state within the service connections, and they offer a couple of scenarios demonstrating alternatives to stateful connections it is not clear whether it is feasible to extend these to all data intensive grid problems, or whether you will just end up making the state in connections harder to spot and, there in, harder to reason about.

2.4.5 Web Service Resource Framework

Web Service Resource Framework [38] (WS-RF) design was published in 2004 in response to the WS-GAF paper offering a different view on how Grid Services can be constructed in an interoperable way with Web Services. This paper argues that by refactoring the existing Grid Service infrastructure it is possible to have state without breaking the Web Service model. They aim to achieve this through the construction of separate entities called *Web Service Resources* which hold the state of the system. These will be created, used, and then disposed of. This moves the previous creation and disposal of Grid Services into a separate entity, creating a much cleaner separation of concerns. However, it is arguable that this still encourages the wrong style of programming to be adopted for such an environment.

The paper proposes to achieve this aim by integrating a range of proposed protocols and technologies, including WS-ResourceProperties [62], WS-ResourceLifetime [114], WS-RenewableReferences, WS-ServiceGroup [63], WS-BaseFault [78] and WS-Notification [64]. Among the projects which are aiming to implement this model are the WEDS [37] project, aimed at enabling distributed simulations, and the Globus Toolkit 4 (GT4) project. The GT4 project is being implemented to replace Globus Toolkit 3, which was one of the main projects targeted for criticisms by the WS-GAF paper.

2.4.6 Globus

The Globus Toolkit [47] (GT) is probably the biggest publicly available Grid middleware project, with an impressive set of sponsors including DARPA, the U.S. Department of Energy, NASA, the UK e-Science program, Microsoft and IBM. The Globus Toolkit aims to provide base middleware for a wide range of high performance Grid computing projects. Their attempts to produce this middleware have resulted in a range of slightly different architectures [38, 48, 52, 53] over the last decade. The latest release, GT4, is based on WS-RF and built on top of a range of existing open-source products including Jakarta Tomcat, Apache Axis and OGSA-DAI, using where possible existing Web Service technologies. This, however, has not always been the case, with the earlier versions, while containing core elements of other technologies, they where incompatible with these other technologies. This caused Globus to be forced to develop separately and it was this type of incompatibility that prompted NERESC to write the WS-GAF paper. These incompatibilities have also existed between different versions of the toolkit making it hard for projects developed using the Globus toolkit to develop as the Globus toolkit does.

To support their aim of providing generic middleware for such a wide range of projects, Globus have attempted to construct a wide range of solutions to almost all of the different issues facing Grid middleware [51]. As a result, it is arguable that Globus's biggest problem is that they have tried to do too much to quickly, so have been unable to effectively observe the different solutions in order to determine how best to amalgamate then into a single package. This has resulted in their middleware in the past becoming unsustainably heavy and complex. This, in turn, has required that they review and in many ways start over again periodically, resulting in many versions. While this has almost certainly resulted in the quality of the Globus toolkits improving immensely, it has also resulted in lots of projects that have tried to use a given toolkit as a supporting middleware suddenly discovering that the toolkit they are using and dependant upon is no longer being developed or maintained.

At the time that this project started, WS-GAF had just been published and the replacement of GT3 had been announced. However, with no implementation of GT4 and no concrete indication that this version would be any more long-lived than previous versions, it was not possible to use Globus. The first release of GT4 was made in April 2005, with two more incremental releases since. This version would appear to be more mature than previous versions, with better interoperability with Web Services. Hopefully, this will give it the strength to stand the test of time and reduce the risks for people deciding to use it.

2.4.7 Open Middleware Infrastructure Institute

Started in 2004, the Open Middleware Infrastructure Institute [90] (OMII) is a group based at Southampton University which works in collaboration with a large number of groups in the UK e-Science program, including groups at the National e-Science Centre and Manchester University. They have three main aims:

- To provide a single repository containing all the middleware produced by the UK e-Science program. This middleware will be evaluated and comprehensively documented in relation to its function, usability, and reliability.
- To provide a point of focus from which to guide the UK e-Science program such that groups can work together to produce middleware that has a greater degree of interoperability.
- To provide a tested and well-maintained package of middleware drawn from the work of the UK e-Science program. This should consist of reliable components that are packaged such that they are easy to install and use.

It is the third aim that is of most interest to this project. They are currently on their second major release, which appears to be stable and usable. As such, people are starting to use this middleware for more advanced grid projects. The software stack, like so many others, is built on top of Jakarta Tomcat and Apache Axis. On top of this, they have a range of pluggable modules that provide services such as file transfer, security, dynamic deployment of processes, support for workflow languages such as BPEL, and user management. All of these are based on middleware constructed elsewhere and integrated in. There is a risk, as happened with the Globus Toolkit, that this work will become too heavyweight and lack interoperability. While some have raised concerns over the size of the OMII package, thus far OMII seem to be avoiding this pitfall.

When we started, the OMII had not reached their first release yet. As such, it was not possible to build on their middleware. However, the work described in this thesis deliberately maintains conceptual compatibility to ensure that we have not closed off the option of migrating to the OMII middleware at a suitable moment in the future.

2.4.8 Berkeley Open Infrastructure for Network Computing

Berkeley Open Infrastructure for Network Computing [2] (BOINC) is an open source software package designed by the team behind SETI@home [3]. Taking their experience with volunteer computing gained through the SETI@home project, they have produced a pluggable infrastructure that handles job and client management, data transfer, scoreboards, server installations, and client software for projects that conform to the same model as SETI@home. All a project wishing to take advantage of volunteered computers has to do is produce plugins that encapsulate their jobs and any required graphics eg. a screensaver, and the code they would like to run on the computers. This makes using such a model very easy, and currently there are about a dozen projects, including CPDN, using this platform.

2.4.9 MONET

Launched on the 1st of April 2002, the MONET [121] project was a two-year investigation into the use of the Semantic Web to provide mathematical Web Services. The aim of the project was to attempt apply the latest ideas for creating a Semantic Web to the world of mathematical software. This was attempted using sophisticated algorithms to match the characteristics of a problem to a combination of the advertised capabilities of available services, and then invoking the chosen services through a standard mechanism. The hope was that the resulting framework would be powerful, flexible, and dynamic, putting state-of-the-art mathematical resources at the disposal of users anywhere in the world.

To achieve this aim, it was decided that they would try to build the middleware framework out of existing and emerging technologies such as SOAP, WSDL, OWL, and UDDI. The middleware framework for this project is constructed from three main pieces.

- The Web Services that provide the mathematical functions. These are augmented with an XML document that describes the service, using the Mathematical Service Description Language [27] (MSDL). This document provides sufficient information to allow the broker to match appropriate services to clients requirements.

- The MONET Broker [32] keeps track of the available services and handles requests for functions from clients. When it receives these requests it composes available mathematical services to construct a service to perform the requested function.

- The Mathematical Object Manager [79] which is responsible to storing the data structures required for the computations. These data structures are indexed with URIs. Both the clients and the services are able to use the object manager.

A typical use case for these components is as follows:

1. Mathematical services register their availability with the broker, using MSDL to describe the services they provide.
2. A client wishing to perform a function locates the broker using UDDI or some other out-of-band technology.
3. The client registers any required data structures with the object manager, receiving URIs with which to reference them.
4. The client submits a query. The broker then constructs a range of possible ways that this query can be performed. These are returned to the client.
5. The client chooses the way they would like to have the query executed, which the broker then performs by contacting the required services registered with it.

The middleware produced by the MONET project is aimed at exploring the possibilities of the Semantic Web, using data structures that are stored in a single piece. As such the middleware produced is of limited use to CPDN. However, the architecture is designed to support the construction of dynamic functions at runtime, as a result it has the potential to provide a significant insight into the type of architecture that may be require to solve the data analysis problems faced by projects like CPDN.

2.4.10 Condor

The Condor research project [77] has been running at the University of Wisconsin-Madison for over eighteen years now and is one of the most successful and widely used distributed computing projects to date. The project was inspired by the observation that most desktop computers use only a fraction of their computing power, and as desktop computers became more and more powerful, this wasted computing resource was becoming ever more valuable. Taking these observations on board, the Condor project aims to enable users to take advantage of this wasted processor time. They note that in such an environment the potential for high *performance* computing is extremely limited, and instead the important focus is high *throughput* computing. That is that the important feature of any system running is such an environment is that it keeps all the processors busy as much of the time as possible. For these reasons, Condor has been nicknamed “the cycle scavenger”.

To take advantage of this spare computing power the Condor team has constructed clients to run on a range of machines and operating systems with potential spare processing power. These clients detect when a machine is idle and advertise using a ClassAd [102] for compute jobs for the machine to do during this idle time. Machines running clients are grouped into a pool with a job submission server for users to submit jobs to. These jobs each also support a ClassAd which describes the resources needed for that job. A *Matchmaker* then compares these adverts with the adverts from the resources and submits possible pairings to both the job and the resource. If the pairing is mutually acceptable, they both inform the matchmaker that they have been claimed. The job will then be sent, complete with any data files that are not natively available to the claiming machine.

Condor supports a wide range of features, including the ability for jobs to take regular check points to protect against failure of the host machine, the rerouting of I/O to the computer the job was submitted from, and the ability to span many networks. If a machine ceases to be idle, then the current job will create a checkpoint and either, be shifted to another available machine in the pool, or back onto the scheduling server to wait for another machine to become available.

An important feature of the Condor architecture is that the client and job decide if they are appropriate for each other, so the a large chunk of the scheduling computation occurs on the claiming resource. This is important because it means that every time a new machine is added to the pool, it brings with it the computing power required for the scheduling. This improves the scalability, allowing pools to contain many thousands of machines without overloading the matchmaker. In addition it allows machines and jobs to choose matches that best suit them, so improving the throughput.

Condor is for the most part a very sophisticated batch system capable of managing resources that range from Unix clusters to office workstations running Windows. As Grid computing has become more popular, they have also extended their work with the Condor-G [54] project. Condor-G is Condor augmented with the required handles to allow a Condor pool to register with an instance of the Globus Toolkit as a Grid resource.

Condor is an extremely powerful and well-developed distributed computing solution used to great effect by a large number of organisations. However, its design is not suited to CPDN data analysis for two main reasons. Firstly, it transfers the data to the computing resource, instead of, where possible, choosing the compute resource to minimise the requirement for data transfer. This does not fit very well with the CPDN model where the data is widely distributed, and in many cases, the analysis is expected to be data oriented. Secondly, as it is designed to handle jobs where the data is in a known number of pieces, it does not contain any means of adjusting functions to handle arbitrarily distributed data at runtime. This means that such information would have to be included on a case-by-case basis into any submitted workflows. It would not be possible to include this information within the individual applications as it this would result in the Condor attempting to bring the entire dataset to a single location. The impossibility of this was the reason that the dataset was originally distributed. Because of the mature and well featured nature of Condor, and the fundamental level at which some of these problems occur, adjusting it to overcome these problems would almost certainly require too much knowledge of the internal workings of the system.

2.4.11 Styx Based Projects

The Styx protocol [16] is a communications protocol used by operating systems such as Inferno [97]. Originally developed at Bell Labs, Styx is a variant of the 9P protocol that was developed for the Plan 9 operating system. In many ways Styx is just like a network file system; however, the critical difference is that with Styx all resources, not just files, are represented using the interface for files. This is somewhat similar to the ideas put forward in REST to make binding heterogeneous elements together to construct distributed systems much simpler. Using Styx, it is possible to mount resources from remote systems as if they were a local resource and then read from these resources as required.

The ability to mount resources in this way has been used by a group at the Reading e-Science Centre to produced middlewares based both on Inferno [17], and directly on Styx [98]. These middlewares are able to take workflows constructed with Taverna [87] and execute them using Styx components wrapped in Web Services. The Styx components allow data to be seamlessly streamed between services as required. The action of streaming data in this manner allows data to be transferred efficiently in its native format, and allows the potential for components later in the workflow to start processing the data before components before them have completed.

This middleware looks to be extremely powerful, and Styx is likely to become an extremely useful tool in the future of distributed computing. However, the current middleware is not suitable for CPDN style projects for a range of reasons, some of which have come up many times before. It has no support to handle natively operations where the data is split into an unknown number of pieces. It makes all data structures immutable after creation; this would result in the data structures becoming too big, as a complete copy is required no matter how small the change is, and finally, as with many of the other projects examined, these middlewares move data to the computation instead of moving the computation to the data.

2.4.12 Self Adapting Middleware

While we have talked about many projects that lack the ability to adapt to their input, there is a range of middleware systems that offer some ability to adapt the way that they are going to run at runtime. These middlewares have mostly been developed by numerical analysis projects and are designed to adjust to squeeze best performance out of the heterogonous hardware that they can be deployed on.

To do this they use a range of techniques; one of these is to include additional system parameters in function calls. These system parameters control the way that the function executes so for instance changing the block size in some numerical methods to ensure that the blocks fit in the processor cache. The value of these parameters can be determined through a suite of tests at install time, as is done with ATLAS [128], ATCC [122], BeBOP [124], and LFC [31]. Additionally, the middleware can keep information from previous invocations and use this to determine the best values for the parameters, as ATCC, BeBOP, and LFC do.

Another technique used by the NetSolve and GridSolve [130] projects is to, as with Web and Grid services, have a directory of services that perform a particular task. Then, through monitoring the performance of these services with different types of input data, it is possible to dynamically decide which of the available services is best to use for a calculation on the basis of the input data in addition to all the usual parameters used for scheduling.

Dynasoar [127] is a prototype middleware, developed to test the dynamic deployment of services at runtime, in a Service Oriented Architecture. This, unlike the other techniques mentioned, was not done to improve the performance of numerical analysis algorithms, but to decouple the connection between the production of code for services, and the computing resources used to execute these services. This makes the architecture far more dynamic, and allows for scenarios such as researchers to placing online code for services they have developed but cannot support, their code can then be run on generic computing resources provided by someone else, elsewhere. This complements the more common scenario of reducing the amount of time cluster nodes spend installing new software, and allowing services to be placed closer to the data they are using.

2.4.13 Other Web Service and Grid Projects

The e-Science initiative, both in the UK and worldwide, has produced a huge number of projects, making it impossible to enumerate them all. However, to give some insight into some of the styles of project, we are going to conclude this section by examining a selection of them.

- One common class of project aims to federate existing computing and storage resources so that they can be accessed through a single interface. By doing this, they hope to make such resources better utilised and more widely available. This style of interface would also make it easier for research bodies to provide large amounts of computing power for their members. Examples of such projects operating on different scales and in different areas of both science and the world include DataGrid [55], EGEE [43], TeraGrid [84], PlanetLab [72, 99], the UK National Grid Service [85] and GridPhyN [9].

- Mirroring the above, are projects that aim to make large, distributed, heterogeneous datasets available to the wider scientific community. An example of such a project is the NERC Data Grid [75]. To achieve this aim these projects aim to interface with new and existing datasets, and to then provide a single interface on top of them all. This should allow users ranging from people analysing small datasets on a single machine, to people taking advantage of Grids of many clusters to use this data without having to worry about different data stores and their individual interfaces.

- Projects like ^{my}Grid [101] and ISPIDER [11] represent the union of these two styles of Grid project. These aim to simplify the construction of rough grained analysis of information split across many datasets, using computing resources provided in many locations, providing helpful interfaces and features missing from many of the current scripted solutions such as *province* [25]

2.5 Workflow Languages

Workflow Languages and their supporting middleware are the means by which users are able to extend the functionality of existing services without submitting native code to servers. They allow the users to combine both existing local and remote services to produce new services with extended functionality. There is a huge number of workflow languages in existence created by a wide range of projects, with a wide range of focuses from the control of scientific simulations to the integration of business Web Services. These languages are designed to work with a wide variety of computation resources, data resources and scientific equipment.

Although there is a large selection of languages capable of interacting with a wide range of middleware, they all address a similar programming model. This restricted scope is highlighted by the pair of papers, “Workflow Patterns” [123] and “Workflow Data Patterns” [109]. These papers define a set of patterns of workflow constructs and data interactions, which are widely used by the workflow community to measure the level of functionality provided by different languages. However, with all the patterns presented, the assumption is made that the data is going to be in a single piece. As a result, this programming model, and these patterns do not work with the data and environment presented by CPDN, where it is not known until runtime how many resources the data is split across. This shortcoming in views is further shown by another widely used paper to define problems with workflows, “Web Service Composition - current solutions and open problems” [115], which again fails to observe this newly arising issue with such dynamic environments. Because of this, none of the currently available workflow languages are suitable in their current forms to satisfy the requirements of CPDN, or any other project that removes the static constraint on the partitioning of data that existed in more traditional distributed computing.

Of the extensive range of workflow languages available, most of these came about because so many projects started at the same time in response to e-Science programs started both in the UK and abroad. As a result, most of these workflow languages were constructed for projects that required a basic workflow language at a time when there were no fully functional and supported languages available. While this has been great for the generation of ideas, it has also resulted in a lot of ground being covered many times by different projects. This will inevitably mean that a large number of these languages will fall by the wayside, leaving a small number of better-supported languages like the ones discussed in this section.

2.5.1 Business Process Execution Language

Business Process Execution Language [4] (BPEL) is an extremely well-featured and popular workflow language designed, as the name suggests, for integrating business services to create new services and features. BPEL engines are included in many packages, such as OMII and GT4, and there is a large family of languages that extend BPEL with new features, like the ability to include snippets of Java code within BPEL scripts [15]. Scripts are written in XML and this, coupled with the level of feature, leaves BPEL open to criticism for being too verbose and hard to use without powerful tooling. In addition, while very useful for creating workflows involving a few hundred services, which is ample for most business applications, there can be issues of scalability for some scientific workflows. However, this is probably more a result of the business focus of the BPEL engines and the parsing of the verbose scripts than the language itself.

2.5.2 Abstract Grid Workflow Language

Abstract Grid Workflow Language (AGWL) [45] is an XML-based workflow language developed to provide a set of high-level compositions ranging from `parallel` and `sequence` to `switch`, `for` and `parallelForEach`, which can be used to hide from the user the underlying complexities of operating in a Grid environment. To achieve this, the user writes their workflow in AGWL using the provided compositions to compose atomic calls to, underlying services, and other AGWL workflows. The compositions and the atomic services used contain the name and type of the task, along with abstract identifiers for any input and output of the operation. At runtime this is compiled into a workflow described in Concrete Grid Workflow Language (CGWL), automatically adding in the information that was missing form the AGWL workflow, such as the bindings of the variables, the transfer of data, and the resources that the task should be submitted to. The CGWL workflow now contains all the information that was missing from the abstract workflow and can be executed on an underlying workflow engine. As the user never sees the CGWL workflow, all the low-level work is effectively abstracted from the user, leaving them just to think about the workflow they would like to construct.

2.5.3 Virtual Data Language

The Virtual Data Language [65] (VDL) is another XML-based language. Its construction was started by the Grid Physics Network [9] project as a workflow language where the user is not required to include the locations of the files and operations they wish to describe. Instead, they just use abstract names for these values. These abstract names are then resolved at runtime by a workflow engine catalogue such as the GriPhyN Virtual Data System [50] (VDS), formerly Chimera. This package includes the Pegasus workflow scheduler [42]. Pegasus is able to take a VDL workflow with resolved abstract names and schedule it into an XML Acyclic Directed Graph (DAG) that can then be passed to Grid management tools such as Condor's DAG-Man. These tools are then able to execute the function described by the DAG before returning the results to the user. The many layers of indirection allow large amounts of this middleware to be used in a wide range of projects, with just the lower layers being adjusted to suit the different environments. This has allowed it to not only be used to run workflows on a large number of Condor pools, but also with projects such as TeraGrid [84].

2.5.4 Taverna and Simple Conceptual Unified Flow Language

Simple Conceptual Unified Flow Language (SCUFL) and Taverna [87] are a language and tool for constructing functions in this language. They were developed by the ^{my}Grid project [101] in response to the lack of open source tooling for graphical construction of workflows appropriate for BioInfomatics, and are now used with a range of middleware. Together they provide a user-friendly front-end for scientists, capable of handling a wide range of computing resources and data stores. There are a range of workflow engines capable of handling SCUFL workflows, including Freefluo produced by ^{my}Grid and Styx based projects produced by the Reading e-Science centre.

2.5.5 Triana

The Triana project [120] at Cardiff University is one of two test-bed applications for the EU-funded GridLab. It started out as a workflow-based graphical problem-solving environment developed to provide interactive analysis tools for analysing gravitational wave data. It now consists of a set of pluggable components that are independent of any specific problem domain. These can be used to construct graphical clients supporting both existing workflow languages and new workflow languages as they are produced. These components are capable of interacting with both Grid resources and Web services, and have already been used to construct clients for a wide range of languages including VDL for use with PPARC and clients for ^{my}Grid and the Styx-based projects mentioned earlier.

2.6 Google and MapReduce

Google is one of, if not *the*, world's biggest civilian user of distributed computing power. In order to make their operation more efficient, they have constructed a range of in-house technologies. These technologies have benefited over other projects from working in the more restricted scope provided by Google, and as a result, have achieved some impressive results. Of these technologies, one that is particularly related to this thesis is the MapReduce framework [41].

MapReduce was created independently at the same time as Martlet, with the aim of simplifying programming on clusters. This programming model shares the observation that it is possible to dramatically reduce the complexity of parallel programming through the use of functional programming. The reduction in complexity is achieved by using the constructs provided within the framework to hide all the error handling and data transfer code, in much the same way that workflows do. However, in addition to hiding this code, it also hides the code for splitting the data from a single resource across many resources, process replication to analyse this data on the different resources, and for merging these pieces of data back again once the computation has completed.

Under this programming model, users first specify a function called *Map* to be mapped onto the input data. This function takes a list, and outputs a list of key-value pairs. They also specify a function called *Reduce* that is mapped onto the sorted output from the first mapped function, merging values from this output that share the same intermediate key into a single value. The types of these two operations are `(value1) -> (key2, value2)` and `(key2, [value2]) -> value3` respectively, with the supporting infrastructure doing the transformation between the type of the *Map* function output and the *Reduce* function input. It should be noted that the user specified *Map* function used here is different to the *Map* function in functional programming that is used to apply the function to the input data.

In addition to these two functions, the user must also specify the natural numbers `M` and `R` and a function *Split* to define how output from the *Map* functions is to be split into the inputs for the *Reduce* functions. The choice of values for `M`, `R` and *Split* depend on a range of issues including how many machines the analysis is going to occur on, the nature of the data and the ease with which the computations can be separated.

When invoked, a program goes through the following steps, an outline of which can be seen in Figure 2.1.

1. The required input data is split into `M` lists of values.

2. For each of the `M` lists of values a *Map* function is scheduled to be run on a machine on the cluster. This can be thought of in functional programming terms as the *Map* function being mapped over a set of data. While it is possible to have enough machines so that there is only one job per machine, having more jobs than machines is recommended as it allows more efficient scheduling of jobs both in heterogeneous environments and when tasks have varying execution times.

3. The results are then distributed using the function *Split*into `R` pieces on the basis of their key values. An example of a typical *Split* function is `key mod R`. Once this stage has completed all, key value pairs with the same key will have been migrated to a single point for reduction, regardless of how many individual map functions generated them.

4. Each of the `R` sets of intermediate results then has the user-specified *Reduce* function run on it independently. In functional programming terms, this is another map. This does mean that it is not possible to perform parallel-prefix style reductions [74].

5. The `R` pieces are then concatenated back together to produce the final result.

For this entire process the user is required only to specify `M`, `R`, the *Map* function, the *Reduce* function, and the *Split* function. At no point is the user required to know anything about distributed programming, as the underlying framework handles all this. The current framework used by Google is built using a C++ API, on which users call methods to set the operating parameters and execute. In addition to this framework, there are open-source APIs written in other languages available. Google's framework works in conjunction with the Google File System [57] (GFS) which supports data stored across many thousands of machines in a fault tolerant way, while providing an abstract single point of access. The integration of the MapReduce scheduler and GFS allows most *Map* functions to be scheduled in such a way that they are

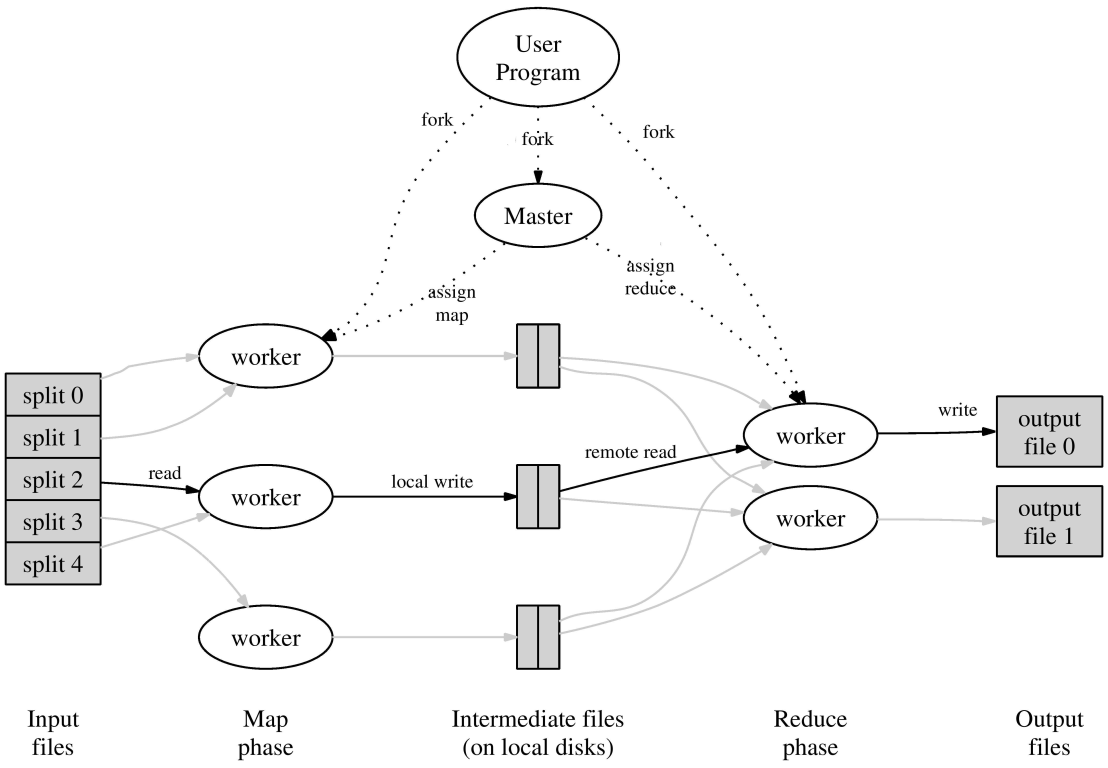


Figure 2.1: Outline of a MapReduce using three workers for the *Map* phase and two workers for the *Reduce* phase.

executed on the computer holding the data, and the output from *Reduce* functions to be added to the file system without moving data. Using this technique, it has been possible to write in under 50 lines of code functions that can *Grep* a terabyte of data in two and a half minutes, and sort a terabyte of 100 byte records in less than 15 minutes, using just commodity hardware [41].

While this work is impressive, there are issues, which do not affect the clusters of computers used by Google programmers, but do make such an approach inappropriate for the scenario presented by the wider field of distributed utility computing.

- This model of computation is unable to support algorithms for performing functions such as the singular value decomposition used in our case study. This restriction is caused because instead of the second phase being a true reduction such as a parallel prefix, it is another map; while it is possible to chain together many MapReduce's to achieve the desired result, the abstraction of the complexity is being weakened by such a change. As such, it would be better to have a more descriptive model to start with. The lack of a more extensive model is probably the result of the restricted scope within which Google work.
- The existing model is limited by the bandwidth of its supporting network, even in an environment such as a Google data centre. This is a situation that will only get worse in the more distributed, loosely-coupled environments we are working with.
- Users are submitting C++ code to be executed. This is fine if the users are sufficiently trusted and limited by immutable data structures or some kind of sandbox. However, for resources that are intended for much larger audiences the potential for abuse with native code becomes much higher. As such, it is desirable to have a workflow language wrapping known functions to prevent such abuse.

2.7 Summary

This chapter has introduced a range of existing and developing technologies for distributed and parallel computing in its many forms. These collectively provide both inspiration and a foundation for the work in the rest of this thesis, as well as demonstrating the difficulties currently faced when trying to use existing models to address the problem introduced and solved in this thesis.

In Chapter 6 we will compare some of the more relevant technologies mentioned here with the programming model, language and prototype middleware described by this thesis. We will then consider how best to integrate the work presented in this thesis into existing projects.

Chapter 3

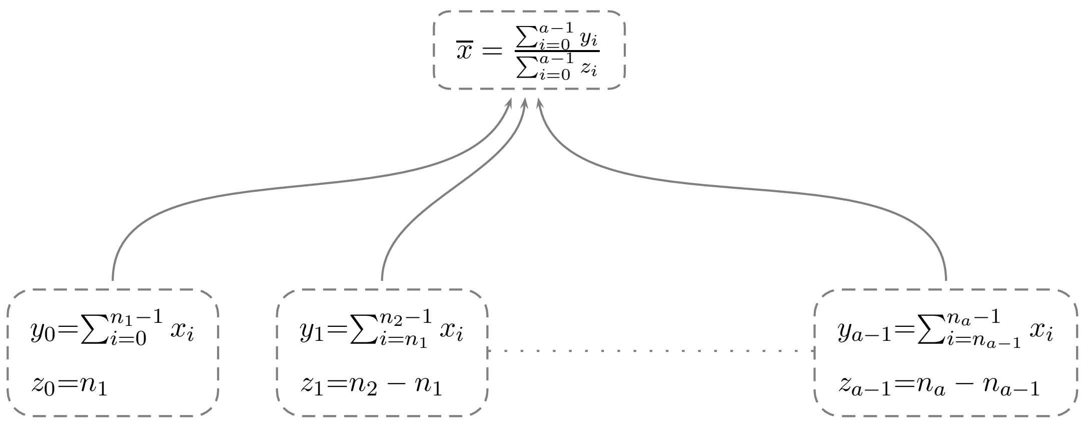
A Distributed Programming Model

In this chapter we will first reiterate the problem, and why existing languages are unable to address this problem. We will then introduce and examine Martlet [59, 60], a rough-grained workflow language that implements a programming model able to overcome this problem and which has the potential to provide a more intuitive way of describing parallel programs.

In Martlet, like many other systems, the execution site and the location of the data are abstracted from the function. However, in Martlet, the splitting of the data is also abstracted. This additional abstraction allows Martlet programs to adjust as the distribution of the dataset changes without user intervention.

3.1 Example Problem

As discussed in the introduction, the average temperature of a given set of returned models is an example of where the level of abstraction described in this thesis is required. If this data spans a servers, the calculation can be described in a way that could be used for distributed computing as:


$\overline{x} = \frac{\sum_{i=0}^{a-1} y_i}{\sum_{i=0}^{a-1} z_i}$
$y_0=\sum_{i=0}^{n_1-1} x_i$
$z_0=n_1$
$y_1=\sum_{i=n_1}^{n_2-1} x_i$
$z_1=n_2-n_1$
$y_{a-1}=\sum_{i=n_{a-1}}^{n_a-1} x_i$
$z_{a-1}=n_a-n_{a-1}$

Where x_0 through x_{n-1} represent the runs distributed across the a servers in the pieces denoted $[x_0 \ldots x_{n_1})$, $[x_{n_1} \ldots x_{n_2})$, $\ldots$, $[x_{n_{a-1}} \ldots x_{n_a})$. The y values then represent the local sums, and the z values keep track of how many runs are included in each y. These partial results are then combined in the final equation to give the overall average. Each of these computations could occur on a different computing resource.

To write this such that it is properly executed in parallel with an existing workflow language, the user first has to determine the number of servers spanned by their data subset; only once this value is known can the workflow be written, and if the value of a changes, the workflow must be rewritten. The only alternative is for the user to construct code to handle restructuring the calculation to match the segregated data. It is not a good idea to ask this of the user, since it adds complexity to the system that the user does not want and may not be able to deal with, as well as adding a much greater potential for the insertion of errors into the process. In addition, workflow languages are not usually sufficiently descriptive for a user to be able to describe what to do with an unknown number of inputs, meaning it is not possible just to produce a library for most languages. This problem is removed with the model implemented by Martlet, as it makes such abstractions a fundamental part of the language.

3.2 Programming Model

To explain this programming model in detail we are going to examine Martlet, a language produced to implement this new model. Martlet supports most of the common constructs of existing workflow languages; in addition to these constructs, it also has constructs inspired by inductive constructs of functional programming languages [12]. These additional constructs, coupled with a new class of data structure, implement a programming model where functions are submitted in an abstract form, and are only converted into a concrete function that can be executed when provided with the partitioned data structures at runtime. This hides from the user the parallel nature of the execution and the distribution of the data they wish to analyse.

We chose to design a new language rather than extend an existing one to implement this model, because the widely used languages are already sufficiently complex that an extension for our purposes would quickly obfuscate the features we are aiming to explore. Moreover, at the time the decision was taken, there were no suitable open-source workflow language implementations to adapt. It is hoped that in due course the ideas developed in this language will be added into other languages.

3.2.1 Data Structures and Functions

The abstraction that the functionally inspired programming constructs provide is built on two classes of data structure: *local* and *distributed*. Local data structures are just like data structures that appear in standard languages. While they can be moved between machines as needed, they are always stored in a single piece at a single location. Distributed data structures, on the other hand, are split into a number of pieces spanning a number of machines, where the partitioning and distribution remains hidden from the user at all times. Abstractly distributed data structures can be thought of as a list of local data structures that, with a suitable concatenation function, can be converted to a single local data structure. Obviously, the concatenation function will change from situation to situation. For example, a distributed matrix is a list of matrices, which, if concatenated together in the order they appear in the list, would produce a single matrix equivalent to the distributed matrix. This is very similar to the structures now being created for the Data Parallel Haskell project [29]; however, as discussed in Chapter 6, Data Parallel Haskell is aimed at programming more conventional parallel machines.

The functional constructs then use this abstraction to construct workflows where the number of pieces that the data is partitioned into is not specified. This is achieved by constructing base cases and inductive cases that can be applied to local data structures. Then at runtime, when the partitioning of the data structures is finally known, the abstractions can be replaced with constructions based on these cases for local data structures. This provides a concrete workflow that is appropriate for that particular partitioning of the data.

Using this model, the distributed average problem looked at in Section 3.1 can be written in Martlet as the program in Figure 3.1, taking the distributed matrix `A` and returning the average

```
// Declare URI abbreviations to improve the script readability
define
{
   uri1 = baseFunction:system:http://cpdn.net:8080/Martlet;
}

proc(A,B)
{
// Declare the required local variables for the computation. Y
// and Z are used to represent the two sets of values Yi and
// Zi in the example equations. ZTotal will hold the sum of
// all the Zi's.
   Y = new DisMatrix(A);
   Z = new DisInteger(A);
   ZTotal = new Integer(B);

// The base case where each Yi and Zi is calculated, and
// recorded in Y and Z respectively. The map construct results
// in each Zi and Yi being calculated independently and in
// parallel.
   map
   {
      matrixSum:$uri1$(A,Y);
      matrixCardinality:$uri1$(A,Z);
   }

// The inductive case, where we sum together the distributed
// Yi's and Zi's into B and ZTotal respectively.
   tree((YL,YR)\Y -> B, (ZL,ZR)\Z -> ZTotal)
   {
      matrixSumToVector:$uri1$(YL,YR,B);
      IntegerSum:$uri1$(ZL,ZR,ZTotal);
   }
// Finally we divide through B with ZTotal to finish computing
// the average of A storing the result in B.
   matrixDivide:$uri1$(B,ZTotal,B);
}
```

Figure 3.1: Function for computing the average of a matrix *A* split across an unknown number of servers.

in a column vector `B`.

3.3 Syntax and Semantics of Martlet

In this section, we examine the syntax and semantics of Martlet with examples, a complete copy of Martlet's grammar can be found in Appendix A. Since this language was developed for large scale distributed computing on huge datasets, the data is passed by reference. In addition to data, functions are also passed by reference. This means that functions are first class values that can be passed into and used in other functions, allowing the workflows to be more generic.

URIs are used to allow the global referencing of both data and functions. The inclusion of these in scripts would make them very hard to read and would increase the potential for user errors. These problems are overcome using two techniques. First, local names for variables in the procedure are used, so the URIs for data and passed functions need to be entered only when the procedure is invoked. This means that in the procedure itself all variable names are short, and can be made relevant to the data they represent. Second, a `define` block is included at the top of each procedure where the programmer can add abbreviations for the parts of the URIs of function calls. This works because the URIs have a logical pattern set by whom the function or data belongs to and the server it exists on. As a result, the URIs in a given process are likely to have much in common. When these abbreviations are used later in the script they have to have a dollar sign on either side. This is done to prevent unintentional substitutions.

The description of the process itself starts with the key word "`proc`", followed by a list of arguments that are passed to the procedure, of which there must be at least one due to the stateless nature of processes. Finally, there is a list of statements in between a pair of curly braces, much like C. These statements are executed sequentially when the program is run.

3.3.1 Statements

All statements exist within a scope. At the start of the function this scope contains just the variables that are passed into the function as arguments. This scope can then be augmented with additional values when new variables are created. These new values will remain in scope until the function execution leaves the statement encompassing the create statement. As a result, it is not necessary to pass arguments from one statement to another, as statements are able to draw their input and output parameters from the scope they are executed in. This is the same as in most imperative languages.

There are two types of statement: normal statements and expandable statements. The difference between the two types of statements is the way they behave when the process is executed. At runtime an *expand* call is made to the data structure representing the abstract syntax tree. This makes it adjust its shape to suit the set of concrete data structure references it has been passed. Normal statements only propagate the *expand* call through to any children they have, where as expandable statements will adjust the structure of the tree to match the specific dataset it is required to operate on.

3.3.2 Normal Statements

As the language currently stands, there are seven types of normal statement: sequential composition, asynchronous composition, if-else, while, temporary variable creation, declaration of constants, and process calls. The set of normal statements was deliberately restricted to just those required for chaining together rough-grained functions, such as those provided by the Climate Data Analysis Toolkit [24], that the project chooses to provide. This was done, as the intention was to make the CPDN data available to the public. As a result, the submission of finer-grained code would not be appropriate; however, there is a discussion in Section 6.2.3 about how to weaken this restriction if Martlet where to be deployed in a more trusted environment.

Sequential Composition is marked by the key word `seq` signalling the start of a list of statements that need to be called sequentially. Although the `seq` keyword can be used at any point where a statement would be expected, in most places sequential composition is implicit. The only location where this construct is really required is when the user wants to create a function in which a set of sequential lists of statements are run concurrently by an asynchronous composition. An example of this is shown in Figure 3.2.

Asynchronous Composition is marked by the key word `async` and encompasses a set of statements. When this is executed, each statement in the set is started concurrently. The asynchronous statement terminates once all the sub-statements have returned.

In order to prevent race conditions and the associated undesirable behaviour, it is necessary that no process in an asynchronous statement uses variables concurrently with another process

```
async{
      seq{
          f1(A,B,C);
          f2(A,B);
          f3(B,C);
      }

      seq{
          f4(D,E);
          f1(D,E,F);
          f3(E,F);
      }
}
```

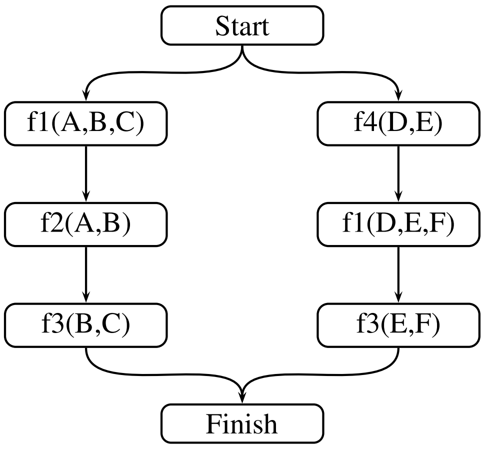


Figure 3.2: On the left, Martlet code using `seq` statements to run two sequential lists of operations asynchronously. On the right, a diagram representing the workflow described by the code opersite.

that writes to them. This would normally be enforced by the middleware at either compile time or runtime depending on when it is first detectable.

if-else and while are represented and behave the same as they would in any other procedural language. There is a test followed by a list of statements. As there are currently no boolean values in Martlet, they evaluate one or more tests on ordered sets, and then use boolean logic to merge these into a single result.

Temporary Variables can be created by statements that look like

```
identifier = new type (identifier);
```

The identifier on the left hand side of the equality is the name of the new variable, The type on the right is the type of the new variable, and the identifier on the right is a currently existing data structure used to determine the level of parallelisation required for the new variable. For example if the statement was

```
A = new DisMatrix(B);
```

this would create a distributed matrix `A` that is split into the same number of pieces as `B`. The type field is required as there is no constraint that the type of `A` is the same as the type of `B`. This freedom is required as there is no guarantee that a data structure of the right type is going to appear at this stage in the procedure. This was the case in the average function in Figure 3.1.

```
seq{
    seq{
        f1(A,B);
        A = new Matrix(B);
        f2(B,A);
        f3(A,C);
    }
    f4(A,C);
}
```

Figure 3.3: In this example `f1` and `f4` use the identifier `A` that is in scope at the start of the function. `f2` and `f3` on the other hand use the identifier `A` that is created after `f1` has executed.

Constants are always local, as it is not possible at submission time to know how to split a distributed constant. This is not a significant restriction, as they can just be included inside expandable statements when a copy is needed for each part of a distributed data structure. Their syntax is very similar to that of temporary variables, as shown.

```
   identifier = const type (description);
```

The main difference is that instead of taking another data structure as a parameter, it takes a description. This description is the same as the description used when instructing the middleware to construct a data structure outside of a function. So for example to construct an integer the description would just be the value that the integer is going to take.

Scope of Variables and Constants Variables and constants created in this way will be in scope for all statements proceeding the construction statement and still within the statement encompassing the construction. If the identifier `A` has the same value as an identifier already in scope for the function, then the newly created identifier will hide the previous identifier from all statements after the construction and still within the statement encompassing the construction. An example of this can be seen in Figure 3.3.

Process calls fall into one of two categories: those statically named in the function and those passed into scope by a reference at runtime. Both appear as an identifier and a list of arguments. However in the static case, this identifier must be a URI, whereas in the case of a function passed into scope, it must merely match an identifier in the functions scope.

3.3.3 Expandable Statements

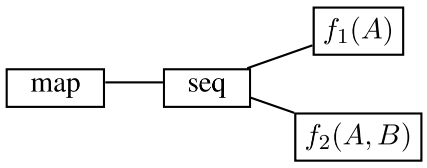
There are four expandable statements: `map`, `foldr`, `foldl` and `tree`. These fall into two categories. The first category just contains the `map` statement, which is used to apply a function independently to each of the separate local data structures contained within a distributed data structure. The other category contains reduction operations, which describe functions that take a set of distributed data structures as input and produce a set of local data structures as output. This includes both folds, which work with accumulating parameters, and `tree`, which performs a parallel prefix style reduction [74]. Each statement has an obvious functional programming equivalent. As expandable statements do not propagate the call to *expand* through to their children, and they must have been expanded before the function can be computed. This means that on any given path between the root and a leaf of the abstract syntax tree, there must be at most one expandable statement.

map is equivalent to `map` in functional programming, it takes a function `f` and a list, and applies this function to every element in the list. This is defined in Haskell below:

```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

An example of `map`, and the corresponding unexpanded abstract syntax tree is shown at the end of this paragraph. Map in Martlet encompasses a list of statements and takes no arguments. This is because unlike Haskell where map is a function, in Martlet it is just a statement. This allows the passing of information to be handled differently. In the Martlet `map` statement, `f1` and `f2` are implicitly joined in a sequential composition to create the function that will be applied to each local data structure in the distributed data structure. The Haskell definition represents this by the passed value `f`. In Haskell, because values are immutable it is necessary to explicitly pass values in and out of functions; however, as they are statements in Martlet, this is not necessary, as variables are implicitly passed as a result of the statement's scope. This allows the list to be represented by the distributed values `A` and `B`. This means, while in Haskell it would be necessary to wrap the lists into a list of tuples to pass in and out of a map function, Martlet is able to hide this complexity.

```
map
{
  f1(A);
  f2(A,B);
}
```



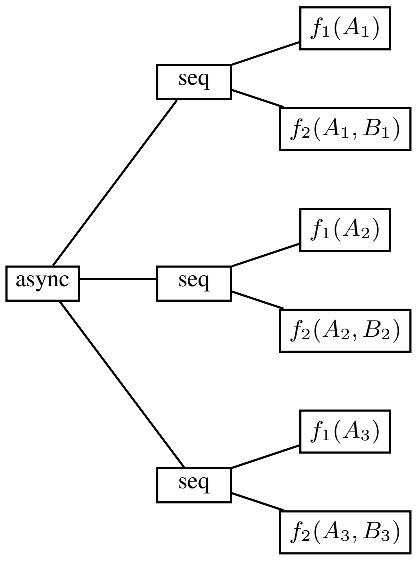


Figure 3.4: The abstract syntax tree for the example map statement after expand has been called setting $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$.

When this is expanded, it examines the distributed data structures it has been passed and creates a copy of these statements to run independently on each local data structure in the distributed data structure as shown in Figure 3.4.

Due to the use of an asynchronous statement in this transformation, no local value that is passed into the `map` statement can be written to. However, local values created within the `map` statement can be written to, as an independent copy of each of these will be constructed for each instance of the function.

foldr is a reduction operation that takes a list, a function `f` and an accumulator parameter. Starting from the right hand side of the list and working left it applies `f` to each element of the list and the accumulator parameter, storing the result in the accumulator parameter. This is defined in Haskell as:

```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

This means that the elements of a list `xs = [1,2,3,4,5]` can be summed by the statement;

```
foldr (+) 0 xs
```

which evaluates to

```
1+(2+(3+(4+(5+0))))
```

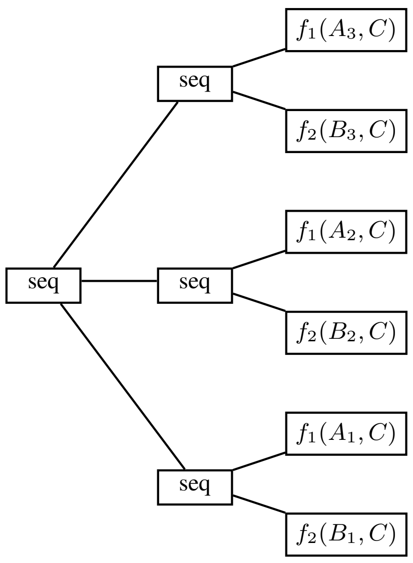
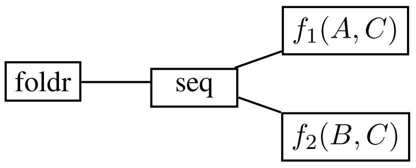


Figure 3.5: The abstract syntax tree after the foldr example has been expanded, with the distributed data structures $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$, and the local data structure C as an accumulator parameter.

Foldr statements are constructed from the `foldr` keyword followed by a list of one or more statements which represent `f`. An example is shown below with its corresponding abstract syntax tree. Like with `map`, because in Martlet `foldr` is a statement and not a function, it is not necessary to explicitly pass in arguments, or wrap multiple arguments in tuples, as arguments are implicitly drawn from the statement's scope.

```
foldr
{
  f1(A,C);
  f2(B,C);
}
```



When this statement is expanded this is replaced by a sequential statement that keeps any non-distributed arguments constant and calls `f` repeatedly on each piece of the distributed arguments as shown in Figure 3.5.

foldl is the mirror image of foldr, evaluating from the left hand side of the list first. It is defined in Haskell as follows:

```
foldl f e [] = e

foldl f e (x:xs) = foldl f (f e x) xs
```

This means that the example from foldr translates to:

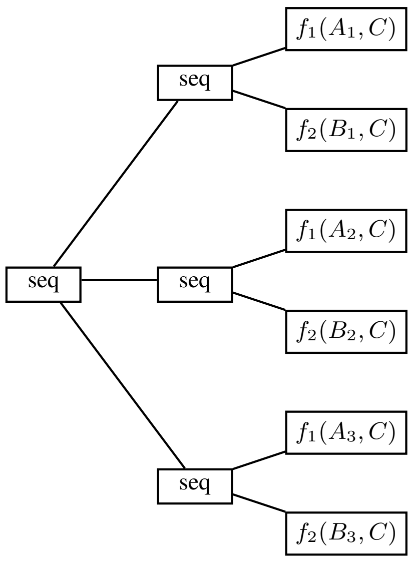


Figure 3.6: The abstract syntax tree after the foldl example has been expanded, with the distributed data structures $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$, and the local data structure C as an accumulator parameter. Note the reversal of the ordering of the distributed arguments compared with foldr.

```
foldl (+) 0 xs
```

which evaluates to

```
((((0+1)+2)+3)+4)+5
```

This is expanded in almost exactly the same manner as `foldr`. The only difference is the ordering of the function calls from the sequential statement. This can be seen in Figure 3.6. The only time that there is any reason to choose between `foldl` and `foldr` is when `f` is not associative.

tree is a more complex statement type. Unlike `foldr` and `foldl`, which can be described by a sequential statement, `tree` constructs a binary tree with a part of the distributed data structure at each leaf, and the function `f` at each node. This means that unlike the folds, it is able to take advantage of the potential for parallel execution when it is expanded and evaluated. A Haskell equivalent is:

```
tree f [x] = x
tree f (x:y:ys) = f (tree f xs') (tree f ys')
                  where (xs',ys') = split (x:y:ys)
```

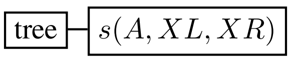
`split` is not defined here since the shape of the tree is not part of the specification. It will however always split the list so that neither is empty.

Unlike the other expandable statements, each node in a tree takes $2n$ inputs from n distributed data structures, and produces n outputs, in addition to the constants and functions used. This means there is insufficient information in the structure to construct the mappings of values between nodes within the tree. In the Haskell definition, this extra information is encoded into the function body. While neat, this requires that the value of n is fixed at the point that the function is created. Since in Martlet `tree` is a statement that is part of the language, rather than a function written in it, this is not practical. Instead, Martlet syntax requires the distributed input arguments and the output argument that the statement uses are declared in brackets above the function to provide the required additional information.

The relationship between distributed inputs and the outputs of `f` needs to be encoded in the format `(XLeft,XRight)\X->A`, where `XLeft` and `XRight` are two arguments drawn from the distributed input `X` that `f` will use as input. The output will then be placed in `A`. This allows the correct placement of inputs, and the identification of the corresponding outputs so that they can be fed into the next level of the tree. Non-distributed inputs and functions used in `f` can be simply drawn from the scope without issue, so are not required in the brackets.

As an example, consider a statement that uses a function `s` to reduce two values to one. This statement takes a distributed argument `X` as input, and outputs the result to the non-distributed argument `A`. This can be written as:

```
tree((XL,XR)\X -> A)
{
   s(A,XL,XR);
}
```



When this is expanded, it uses sequential, asynchronous, and temporary variables in order to construct the tree as shown in Figure 3.7. In order to get a better idea of the tree this describes, a diagram showing just the data flow is included in Figure 3.8. Because of the use of asynchronous statements, any value that is written to must be passed in as either an input or an output.

3.3.4 Recursion

Martlet is not currently directly capable of recursion, as functions are not named by the programmer, but instead are named by the middleware. As a result, it is not possible to include a function's name within itself. However, it is still possible to construct recursion by using the property that functions are first class variables. Therefore if a function `x` takes a function refer-

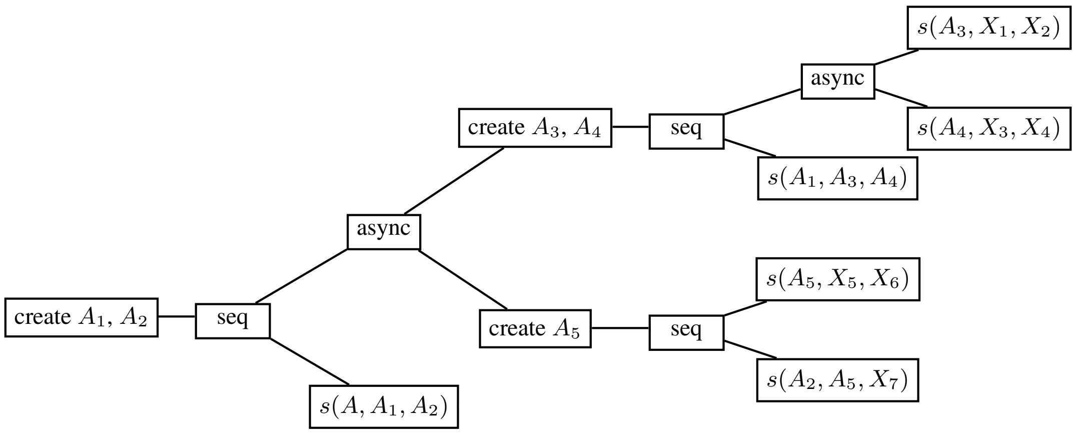


Figure 3.7: If X is set $X = [X_1, X_2, X_3, X_4, X_5, X_6, X_7]$, then this is one of the possible trees that could be generated from the tree function on page 52.

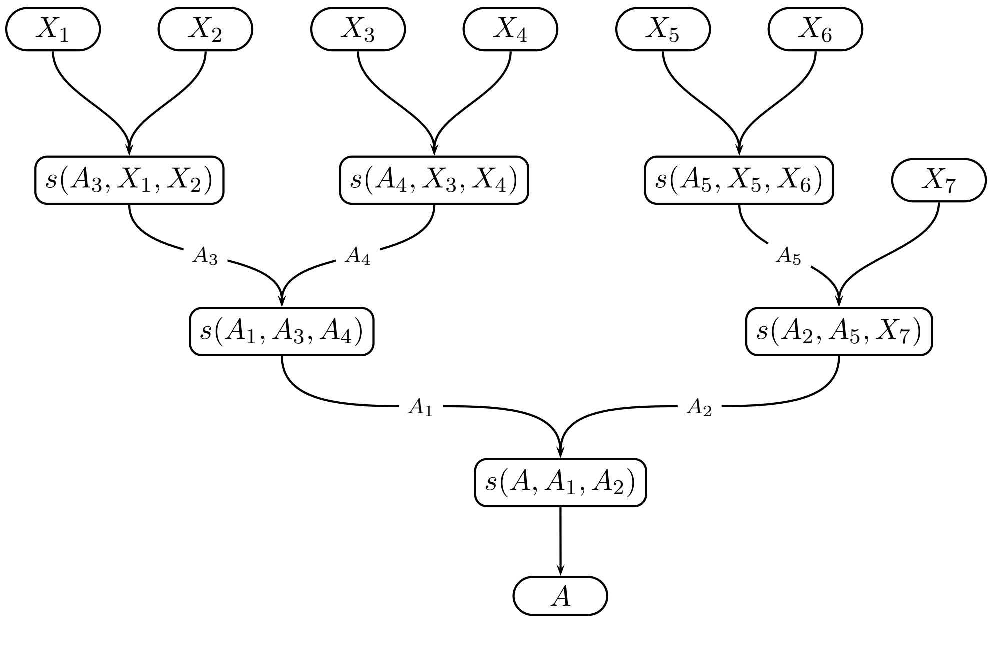


Figure 3.8: A view of the execution described by the tree in Figure 3.7, tracing out just the data flow.

```
proc(f,A,B)
{
  \\Some code......

  \\The recursive step
  f(f,A,B);

  \\Some more code......
}
```

Figure 3.9: An example of how recursion can be included in Martlet workflows.

ence `f` to call, recursion can be introduced by setting `f` equal to `x` when the function is called. This can be seen in Figure 3.9: in this example `f` will be called as a passed function argument, if the value of `f` at runtime *is* the URI of this function, the function will now recurse.

This could be simplified by the addition of an identifier such as `this` that the function can use to identify and therein call itself. However, recursion is not necessarily a good feature to have in a workflow language, as including it makes scheduling more complex and can result in a loss of performance through not having the full tree defined when the workflow is first initiated at runtime. This has resulted in a large class of middleware that is not able to support such functionality; therefore, languages that include recursion have a more restricted set of the middleware on which it is possible to execute their workflows.

3.3.5 Function Constraints

As the prototype version of Martlet stands, there is very little in the way of extra constraints to make functions easier to read and less likely to contain errors. There are two types of constraints that it would be constructive to add, these are type information and input/output information in the function headers. While the addition of both of these would make functions easier to read and help prevent functions being partially evaluated before failing, they where not included in the prototype because the naive versions would limit the general nature of the language, and more advanced systems would both require more knowledge about how the language will be used, and require too high a level of work for a prototype. Once more experience has been gained using this programming model with different users, and in different problem domains, it to be it will be appropriate to revisit this issue.

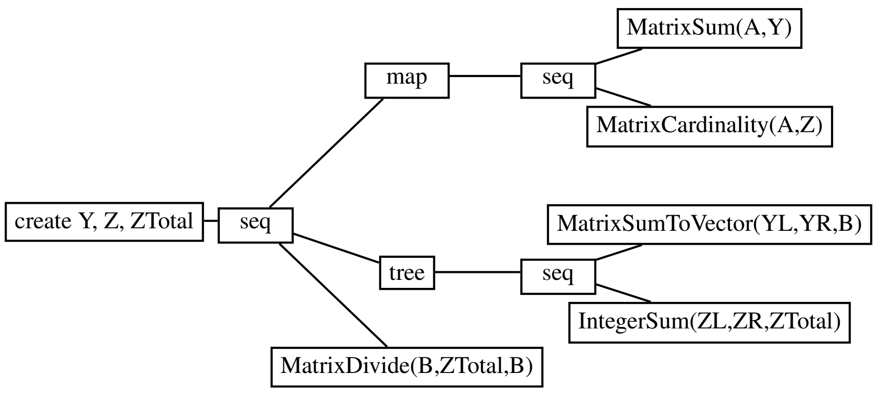


Figure 3.10: The abstract syntax tree representing the generic work-flow to compute the function in Figure 3.1.

3.4 Example Program Execution

In this section, we will take the Martlet program in Figure 3.1 (the example process to calculate averages) and show the stages that it goes through between being submitted by a user to being executed on a concrete dataset.

When the initial program is submitted, an abstract syntax tree that directly represents the abstract function is generated. This is shown in Figure 3.10. If this is invoked with arguments `A` and `B`, where $A = [A_1, A_2, A_3]$, the tree is copied and the copy expanded to match these concrete data structures. The result of the expansion of the tree is shown in Figure 3.11.

Once the tree has a concrete number of arguments for each of the distributed data structures and no longer contains expandable constructs, it can be broken up and executed on appropriate computing resources. At the moment this is done through the use of a supporting middleware that has been developed as part of this project [61]. However, at this point, there are no constructs used that are not provided in a wide range of other workflow languages. This means it would now be possible to use the expanded tree to produce workflows that can be used on other middleware. This is looked at in more detail in Section 6.2.1.

3.5 Conclusion

In this chapter we have introduced a novel programming model, and implemented it as a rough-grained workflow language capable of abstracting from the user the partitioning of data used in an algorithm. This is done without simply transferring all the data silently to a single location

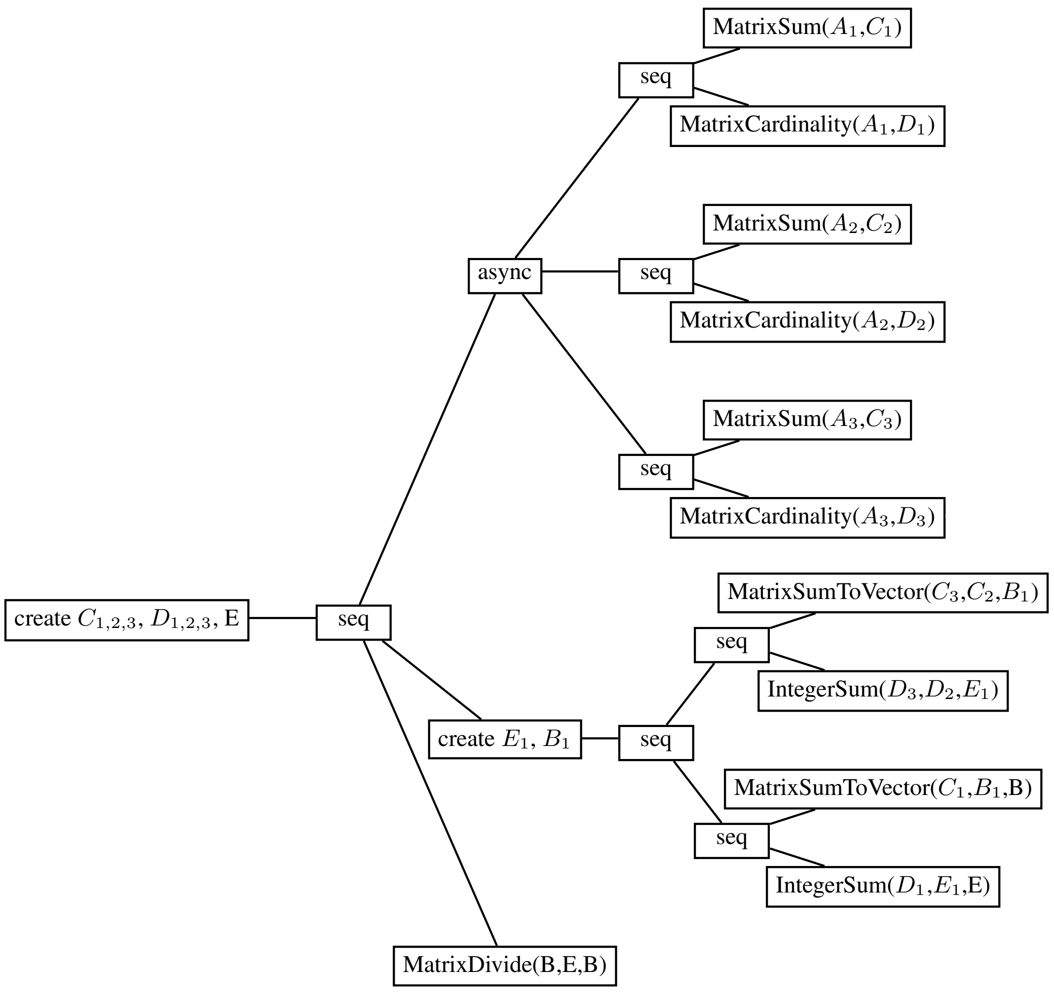

create $C_{1,2,3}$, $D_{1,2,3}$, E
seq
async
seq
MatrixSum(A_1,C_1)
MatrixCardinality(A_1,D_1)
seq
MatrixSum(A_2,C_2)
MatrixCardinality(A_2,D_2)
seq
MatrixSum(A_3,C_3)
MatrixCardinality(A_3,D_3)
create E_1, B_1
seq
seq
MatrixSumToVector(C_3,C_2,B_1)
IntegerSum(D_3,D_2,E_1)
seq
MatrixSumToVector(C_1,B_1,B)
IntegerSum(D_1,E_1,E)
MatrixDivide(B,E,B)

Figure 3.11: The abstract syntax tree representing the function in Figure 3.1 after it has been expanded to fit the concrete data structures $A = [A_1, A_2, A_3]$ and B.

in the background, so avoiding the limiting bottleneck in such a solution. Instead, this has been achieved through the use of abstract statements in the language inspired by functional programming and two complementary classes of data structure. These constructs also facilitate an alternative and potentially more intuitive way of thinking about parallel programs in terms of just base cases and inductive cases.

In the next chapter, we will introduce a prototype middleware that has allowed the evaluation of this programming model and language, before we test and evaluate the complete system in Chapter 5 and Chapter 6

Chapter 4

Middleware

The prototype middleware described in this chapter was developed to construct a specification of the style of architecture required to support our programming model in a heterogeneous distributed environment. This then provides insight into the system support required by such a programming model, in addition to providing a platform for experimentation with Martlet.

In Chapter 2 we saw that there is a range of existing middleware supporting Service-Oriented architectures for utility-style computing. However, the more advanced of these fell into two categories: they were either locked quite firmly into the more traditional, and inappropriate for Martlet, computing models, or they were still very much in development and so were not at a stage that was stable enough to extend to support this project. As such, it was necessary to construct a large amount of custom middleware to support Martlet, this was done by building on lower level, well-established middleware. As large parts of the infrastructure were going to have to be constructed from scratch, it was decided at the start of this project that we would construct it all from scratch. This allows us to examine in an unbiased way the requirements for such a system, making it easier to see both which middleware it can be integrated with later, and where else such a pattern could be instantiated. In addition it insulated us from events in an immature field, such as the abrupt end of the Globus Toolkit 3 project.

4.1 Introduction

The Middleware is built on top of an Apache Axis [5] server running in an Apache/Jakarta Tomcat [6] servlet container. This choice was made because was is one of the best supported, widely used and functional SOAP containers and was marked to become the corner stone of the

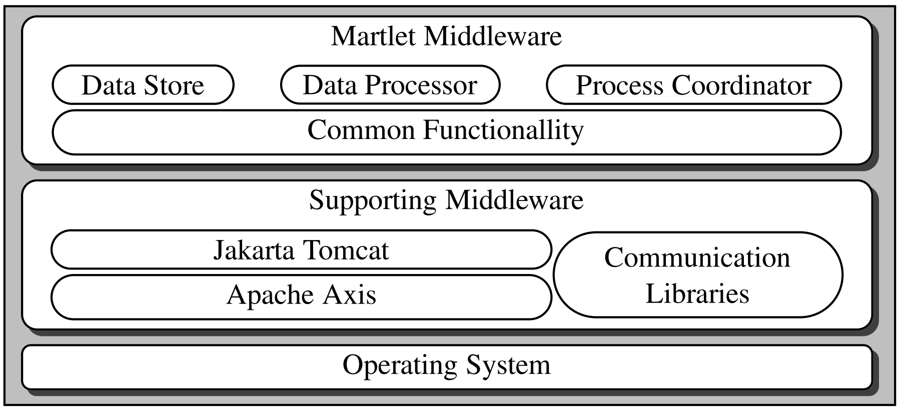


Figure 4.1: Diagram outlining the stack used by the prototype middleware.

middleware produced by the OMII [90]. This means that, while we had a very stable platform to work with, we are not segregating ourselves from other work that is going on in this field. In due course, we may migrate to the OMII middleware, in order to gain use of all the additional functionality supported by the OMII stack, such as WS-Security [7], but at the start of this project, there were no OMII releases available. An outline of the middleware stack used can be seen in Figure 4.1. The middleware is inspired by a range of existing work including ASAP [106], MONET [121], WS-GAF [96], and WS-RF [38]; this has resulted in a factoring of components similar to that used by the MONET project, the use of factories to construct instances of invoked services, and the use of handles and URIs to manage resources. However, it should be noted that the middleware is not constructed with a strict adherence to any one set of ideas or principles.

The middleware is constructed such that the different conceptual parts of the system are separated into different concrete parts. The structure of the communication between these parts and the use of XML documents posted by each node on the Web means that nodes can be added and removed at will, without having to inform the whole system of the change.

Data is passed by reference, which has two main advantages. First, the data is only moved when it is required, allowing the references to be passed through many services without having to move very large amounts of data through with them. Second, the use of references means that data transfer can be instigated by a more efficient method than SOAP [33]. As a result, the data transfer model is abstracted so that the protocol used to transfer the data is separate from the implementation of the rest of the middleware, with lazy evaluation [70] being used to minimise the data transfer overhead.

In addition to the use of references to pass data, references are used to pass functions. This allows functions to be first class values that can be passed into and used in other functions. This applies both to functions that have been submitted to the middleware using the Martlet language and to the base functions that are deployed to the system by administrators.

The middleware also has a comprehensive locking, provenance and data transfer model that allows multiple functions to run at the same time, and functions to be queued without having to worry about forgetting data or generating race conditions. The system that performs locking is extended to keep records so that a provenance system is built into the middleware, thus keeping a complete record of the life of each piece of data.

4.2 Topology

The implementation is divided into three logical units: data stores, data processors, and process coordinators.

Data Stores provide a set of methods for accessing the data stored at a given location. This unit is deliberately lightweight and only capable of generating a data structure from stored data.

Data Processors ingest, store, and run expanded Martlet abstract syntax trees on datasets which they either have locally, or retrieve from another data processor or data store.

Process Coordinators are the components that users interact with. They handle access to the rest of the middleware. This is consequently where most of the complexity of the project occurs. This is where the abstract trees that represent submitted functions are adjusted to fit the arguments on which the function has been called, and then are broken up and scheduled across the data processors. This is the only component to have any knowledge of other nodes in the system.

All three components share common functionality for transferring, discarding, and publishing data. They can be grouped together at will on servers, so it is possible to have both a data processor and a data store on the same machine. A possible configuration of five servers is shown in Figure 4.2. It is worth noting that many different process coordinators can use each data processor and data store concurrently, and each process coordinator does not need to be aware of all the other nodes in the system.

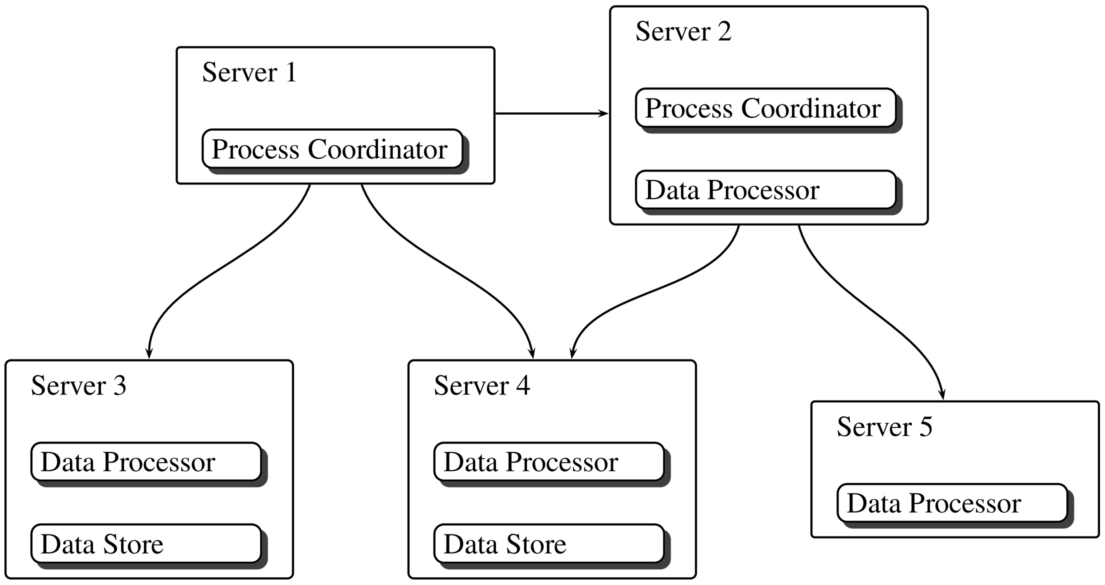


Figure 4.2: An example of how five servers could be configured. Note that more than one Process Coordinator can use each Data Store and Data Processor and the Process Coordinator does not need to know about all available Data Processors.

This topology has similarities with the architecture of the MONET project [121], with the process coordinator mapping to the MONET *broker* and the data store mapping to the MONET *mathematical object store*. Although elements of the architecture are similar, this work serves an entirely different purpose. MONET was aimed at automatically combining services for mathematical functions, not at the abstraction of parallelisation from analysis of distributed data.

4.3 Shared Features

Before examining the implementation of the logical units, we will examine some of the design issues relating to the implementation of their shared functionality.

4.3.1 Resource Lookup

All data and functions are held in handles stored in a lookup table. This lookup table is indexed by URIs that are created when handles are added to the lookup table. The use of handles facilitates the seamless storage and passing of metadata about the data structure or function. This metadata consists of both object-specific information and implementation-specific information.

Examples of this data include the number of processes currently using a data structure, whether a data structure is locked to prevent race conditions, and when the data structure's existence can no longer be guaranteed.[1] The inclusion of resource lifetimes is similar to the ideas in the WS-RF [38] specification, where resources are given an updateable life time. This means that if a data structure is lost because of either the failure of a server or the forgetfulness of the user, the resources used will be recovered automatically.

4.3.2 URIs

A range of information is encoded within the URIs used for referencing handles in the lookup table. This information includes the URL of the holding server, a unique identifier within the domain of the holding server, the type of the item referenced, and the group it belongs to. The structure of these URIs is:

`Martlet:`*`unique id`*`:`*`type referenced`*`:`*`group`*`:`*`server url`*

so an example URI would look like:

`Martlet:matrixAverage:Function:all:http://cpdn.net:8080/CPDN`

Through the combination of the unique identifier and the server url, all URIs are guaranteed to be globally unique. The encoding of the URL into the URI also allows servers to dynamically discover other servers as and when required, thus reducing the amount of state required in the system.

We chose to encode this much data into the URI because the desired system architecture required this information to be provided. The only other sensible alternative currently available would be the use of WS-Addressing [20], in which this information is encoded into an XML document that is used instead of the URI. We felt that this added unnecessary overhead parsing the document; however, a suitable level of abstraction has been included so that this decision can be changed later if needed for reasons such as interoperability.

4.3.3 Security

Security falls into two categories;

1. Preventing users from running malicious code that is harmful to this or other systems.

[1]If there is space to continue storing a data structure there is no reason to actively destroy it. Instead, this can be delayed until the garbage collector is invoked.

2. Preventing users from accessing or modifying data without permission.

The first-category is handled by Martlet restricting the user to chaining together rough-grained functions provided by the system administrator. This allows the system administrator to provide a comprehensive but safe set of functions, thus preventing any malicious code from being executed.

To ensure that the security model for the second category remains orthogonal to the work on the underlying design, we decided that this should be implemented by having a security handler on every server. Messages have to pass through this handler before reaching the underlying framework. If the messages are allowed through then it is assumed that the request is legal and it will be executed. It is possible for the handler to check a range of issues, including the source of the message and the permissions of the sender. This fits in well with the model that Apache Axis [5] uses for service construction, in which services are made from chains of handlers, with each handler responsible for a specific task.

If the URIs are changed so that the security handler lets a request through, the request will fail to map to a handle in the resource lookup, and will fail before it can do anything harmful. By doing this, the security can be abstracted away from the rest of the middleware while still being taken into account in the design. The group field in the URIs was added to enable users and sets of handles to be grouped, thereby simplifying the administration of the security model.

4.3.4 Sending Messages

The communication of messages to other components of the middleware is handled by a set of classes, one for each different type of component. This abstracts the details of communication away from the workings of the middleware to a single point. Coupled with the use of Axis handlers, this allows for the handling of errors, thus providing a reliable means of communication that can automatically handle a range of errors. Unrecoverable errors will propagate through for the calling infrastructure to handle. These will however, either have been thrown by the receiving service, or because the communication remains impossible after the communication class has tried all available options.

4.3.5 Data Transfer

Data transfer is initiated by a service provider that requires a specific piece of data making a request to the provider that holds the data. There are two types of data retrieval request a service can make. For data types that are small, such as integers, it just requests that the value is sent as the response to the request. For potentially large data types such as matrices it requests that the data is sent by an out of band method. This is done because SOAP is not a good format for sending large datasets due to its verbose nature [33].

In order to save on communication times data transfer occurs if, and only if, a service is handed a reference to a handle on a different server and it tries to read the data from it. This is instigated by the service first recognising that it the URI is not local, and therefore creating a lazy [70] handle. This lazy handle has all the information it requires to transfer the data, but only executes this operation if `get` is called on the handle to retrieve the data it holds. If a `set` call is made, the lazy handle will remove the old copy of the data from the remote server, making the local version the only copy. Otherwise, when the function terminates the local copy is removed in order to ensure that only one copy of the data exists at any time. The decisions about which copy of the data to remove were made in order to keep the amount of network traffic to a minimum.

If copied data is unmodified, the copy of the data will be deleted after the server has finished with it, so the user URI will remain unchanged. However, if data is modified, the original is deleted leaving only the copy. This process invalidates the original URI, leaving just the new URI which the user is informed of when the modifying function terminates. As the original URI no longer points to anything it is not safe to share the URIs of mutable data with other users, as there is no guarantee that it will not be changed leaving no resource referenced by the shared URI. This restriction can be removed by taking a copy of the data to be shared and continuing any further work with the original.

We will now examine out of band requests in more detail. These requests contain both the URI of the required data and the location where the service would like the data to be placed. The data is encoded into an efficient format and transferred, after which an asynchronous call is made that informs the requesting service provider that the data has been placed in the requested location. Currently the data is transferred by SFTP. This choice was made for convenience, and is deliberately abstracted so it can be changed to a different protocol at a later stage if desired.

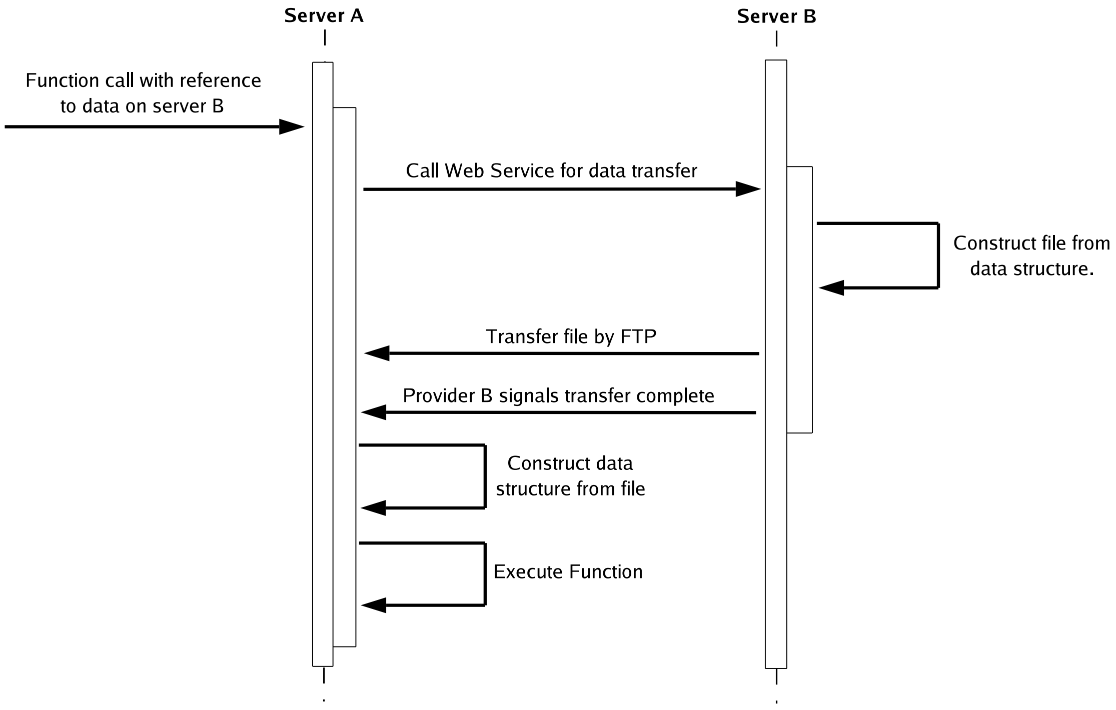

Server A
Server B
Function call with reference to data on server B
Call Web Service for data transfer
Construct file from data structure.
Transfer file by FTP
Provider B signals transfer complete
Construct data structure from file
Execute Function

Figure 4.3: Process diagram of non-local reference retrieval.

Since the receiving server initiates the data transfer, the overall model for transferring the data is a pull model, though the underlying protocol to transfer the data uses a push model, in which the sending processor initiates the transfer. This is done because it reduces the amount of inter-node signalling that is required to organise an effective cleanup of transient data structures. All message calls are asynchronous to avoid timeout issues occurring while the server is marshalling the potentially very large amounts of data. Figure 4.3 shows a generic outline of the sequence of events for data transfer.[2]

Each data type requires an encoder and decoder. These can be constructed by extending a provided class that implements the base functionality. For example, matrices are encoded using the netCDF [105] file format. The choice to use netCDF for matrix transfer was made for several reasons: it is one of the formats that is supported by the NERC DataGrid project [75, 76]; it is supported by libraries in a wide range of languages; and it is the native format of the returned climate data.

[2]The function call is not restricted to being from a process coordinator.

4.3.6 Resource Locking

As the environment will require multiple threads in order to provide acceptable performance, data locking is required to prevent race conditions and ensure only valid answers are produced. One solution would be like in functional programming, to make all data immutable after creation. However, since there may be many operations that just make small changes to large data structures, this could result in large time and space complexities, making it an impractical solution. Instead, the chosen design was the addition of information to the handles that can only be altered by a single monitor. This enables large numbers of handles to be locked atomically, thereby protecting against internal deadlock. Deadlock during inter-server communication is a problem since atomic locking across all servers is neither practical nor possible. To prevent deadlock occurring, it is necessary to restrict the user in such a way that if a function is going to write to a reference, while that function is being executed, all other functions that use the same reference must be submitted through the same process coordinator.

The monitor is constructed using a static synchronised method to try and lock an array of handles as an atomic action, else return the handle that could not be locked. Each handle has a method that the monitor can call in order to wait on until the handle becomes free, at which point the monitor can try to get a lock again. Each lock can be either *unlocked*, *read only*, or *write*. The type of lock that each handle requires to perform a given action is read in from the called function before any attempt is made to get a lock. This is abstracted away from the programmer by an abstract class called a *function object*, which is extended by all objects that read or write to data structures. This class contains a Boolean array stating which arguments are *read only* and which are *write*, along with four methods for controlling access to the arguments; *ready()*, *finished()*, *invoke()* and *run()*. These methods perform the following tasks:

Ready returns with the service provider in a state where all handles held by the object are locked by the object making it safe to proceed with the function's execution.

Finished unlocks all the handles, notifying any function objects waiting on them that they are now free. It then signals any processes listening for the completion of the function before returning.

Invoke is the abstract method that must be extended with the function a class is going to perform.

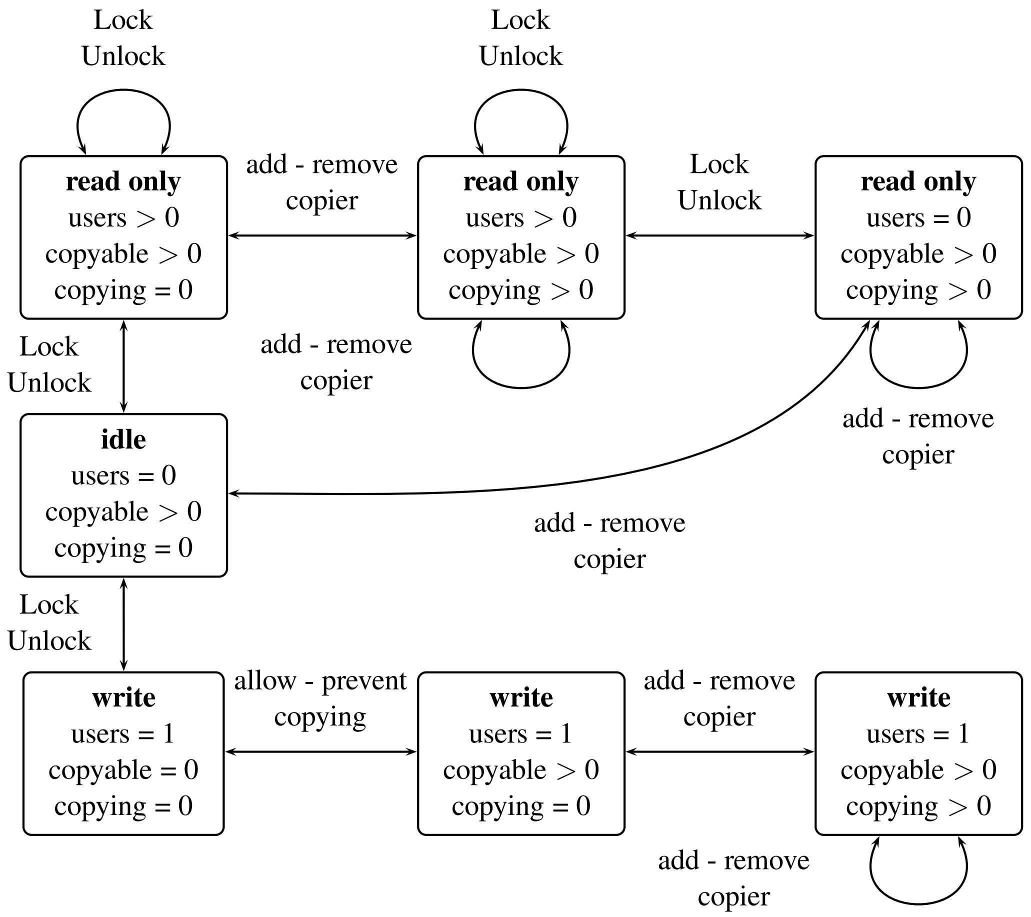


Figure 4.4: The state diagram for resource locking.

Run spawns a new thread, therein providing the required asynchronous nature. This method calls *ready()*, *invoke()* and finally *finish()*.

Additionally, handles hold locking information to control when the data they hold can be copied safely. Since data transfer only reads from a single handle, it is sufficient to make each handle a monitor for its own lock. The necessity for the second lock is created by the scenario where a composite function calls a function on a different node, passing references to local data. The data is not in a final state at this point, so it must not be used by other functions, but the data transfer request must not be locked out. A complete state diagram for the locking can be seen in Figure 4.4.

4.3.7 Provenance

Provenance [25] is one of the pieces of metadata held in handles. It is of importance to this project since the reproducibility of results is vital to any scientific work and, with the ability to publish URIs online, it is not necessarily sufficient for the user to record what they have done. Moreover, it is possible for such a system to record information about any operations performed on the data with greater detail, accuracy and fidelity than a user could manage.

The provenance data is stored as a directed acyclic graph, in which each node represents a function call and the vertices from the node lead to the nodes that represent the inputs to the function. Each handle holds the node that represents the last operation to happen to its data. As such, each handle starts with a node listing the dataset it was constructed from. When an operation is performed on this handle the node is replaced with an operation node that has the old node as one of its children. From this graph, it is then possible to construct a list of operations that have been applied to the queried data structure to transform it into its current state. The next node is constructed by the functions in the abstract class function object. Through the adjustment of this class, it is possible to include the storage of a wide range of information, such as execution time, middleware overhead, the server the execution occurred on, the date, and so on.

4.3.8 Publishing

All nodes have a *publish* method which takes a reference to a data structure, and returns the URL of a Web page where it has been published. For a local data structure this is done by encoding the data structure into a file and placing it into the directory structure of a Web server along with a corresponding Web page. For distributed data structures, the publish call is propagated to all the nodes holding parts of the structure. The returned URLs are then added to a Web page constructed on the server holding the distributed handle, and the URL of this page is returned to the user.

4.4 Data Store

The Data Store is the logical unit that retrieves data from the stored datasets and manipulates it into a data structure for analysis. It adds just one method to the base functionality. This method takes a dataset descriptor and a range descriptor. It uses these to retrieve the data described by the range descriptor from the dataset described by the dataset descriptor. From this data it constructs a data structure and places it in a handle, returning its URI reference. For example, the range descriptor may state that "only the temperature of Europe is required" and the dataset descriptor may state that "this should only be taken from returned models that had pressure at mean sea level set to 1013mb when the model was started, and that were returned before the 1st of May 2007".

The process coordinator constructs the range descriptor from a description submitted by the user when requesting the construction of the data structure. The dataset descriptor, on the other hand, must be constructed earlier by a separate call to the process coordinator. This is required because the global dataset is constantly growing as more model runs are returned and, if two subsets are going to be compared it will be necessary for them to be taken from the same dataset i.e. data structure 1 must be constructed from the same returned models as data structure 2. By the use of the dataset descriptor, a fixed point in time is created from which comparable data structures can be created. When the dataset descriptor is created from the database of returned runs, it not only contains the returned results from which the data structure is going to be built, but also the servers these model runs are stored on. As a result, the process coordinator needs only to know the location of the database for the project, not the location of every data store used by the project.

4.5 Data Processor

Data processors perform analysis on the data structures generated by data stores. To do this, they store a set of *base functions*, that have been approved for use. Base functions are constructed by wrapping a function through the extension of the abstract class *function object*, and building a factory to construct these wrapped functions through the extension of the class *function constructor*. The use of wrapping like this is important as it allows for the integration of legacy code into the set of base functions. The information about these functions is added to an XML document and placed on a supporting website so that any process coordinator that wants to use the data processor is aware of the supported functions. The decision to host the XML locally and not use an advertisement elsewhere was made because the additional level of indirection is not required and restrictions on the structure of the advertisements would restrict the design if an inappropriate choice were made. However, a sufficient level of abstraction has been maintained so that this decision can be changed later if desired.

The data processor has three additional core methods that extend the base functionality:

Function Invocation takes a reference for the function and an array of references for arguments. It retrieves the handle containing the function constructor and uses it to construct an object that will perform the function on the arguments provided. Creating *function constructors* as factories for *function objects* simplifies the management of multiple concurrent calls to the same function, leaving the function object to deal with locking the handles and executing the individual request as discussed in the section on resource locking.

Function Submission allows for the submission of new functions described with a Martlet abstract syntax tree. When submitted, the abstract syntax tree is parsed into a tree of *function constructors* that can be placed in a handle and used to construct *function objects*. This method does not affect the published XML documents, since the submitted methods are not base functions but parts of a function currently being executed by a process coordinator. As such, they are not guaranteed to be self-contained, may have a very short lifetime, and are not appropriate for use in other functions.

Creating New Handles is a simple method that creates empty handles of a given type for storing the output of computations. It returns a corresponding URI after creating the handle.

4.6 Process Coordinator

The process coordinator is the part of the system with which the user will interact, and the only part to handle unexpanded Martlet programs. As has already been mentioned, and will be discussed further in Section 6.2.1, some parts of this prototype middleware could potentially be replaced with elements drawn from existing middleware. However, the Process Coordinator has large amounts of functionality that is not currently supported by existing middleware. It adjusts Martlet functions to match the splitting of data, locates servers, and schedules jobs to compute the adjusted functions. It also coordinates requests across all other parts of the system, handling the construction of both functions and distributed data structures. We will now examine in more depth these pieces of functionality.

Data Structures As discussed in Section 3.2.1, there are two classes of data structure, *local* and *distributed*, that need to be supported by the middleware. Local data structures always reside on a single server, and are the same as the data structures you would use with any other workflow language. All servers in the middleware support these. Distributed data structures are abstractly a list of local data structures. This means that, when a new data structure is added, its distributed counterpart is automatically generated. Unlike local data structures, only process coordinators can handle distributed data structures. These encode the list of local structures using an array of URIs referencing the local structures. This limiting is possible because the Process Coordinator is the point where abstract functions are expanded to concrete functions, and concrete functions only use local data structures.

4.6.1 Locating Servers

As a process coordinator is not simply provided with a list of servers, it is necessary to locate servers. This is performed in two separate ways.

The simpler of these is the method used to locate data stores. In this case, the process coordinator is given the address of a Web Service that provides an interface to a database storing the location of the returned runs. Through querying this Web Service, it is possible to get a list of all the servers that are required to supply data for a given computation, along with a description of what data they must supply.

The second case is locating data processors, and keeping track of what services they provide. This is done by, maintaining a set of URLs of active data processors. For each URL, the process coordinator periodically queries the XML document published by the data processor to see what functions it supports. From these documents, the process coordinator is able to maintain for each base function, a list of servers that support it. Each of these lists is referenced by a URI to enable users to use the base function in Martlet programs. When such a program is run, the URIs will be substituted with URIs from the lists they reference. A diagram showing the linking for three servers supporting two functions and their corresponding handles is shown in Figure 4.5.

If a URI is used that references a different process coordinator, then the referenced process coordinator can be queried about its set of data processors, thereby increasing the number of data processors each process coordinator can use.

If a server is removed in a controlled way, it will empty its XML document of functions a

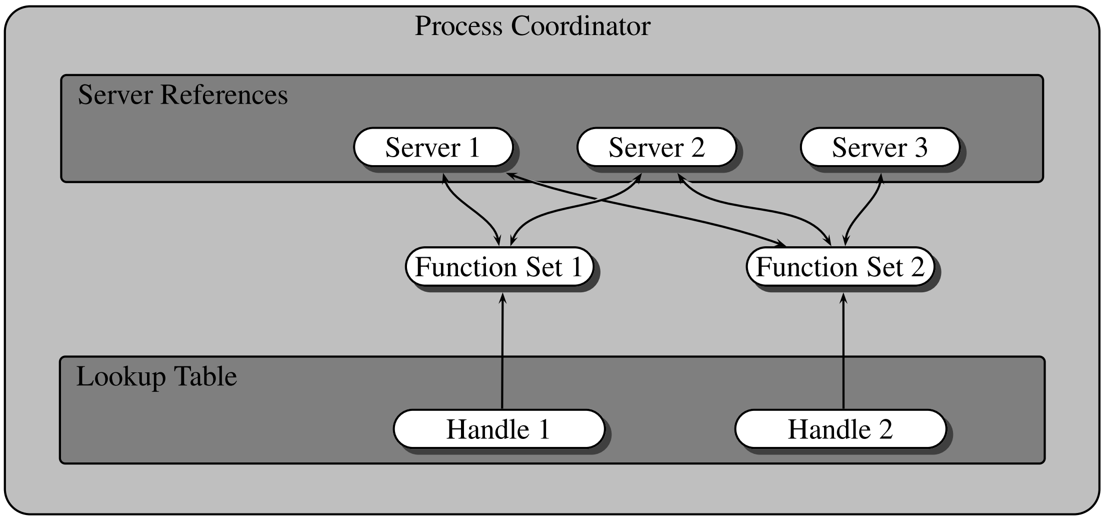


Figure 4.5: The objects and linking for three data processors, which collectively support two functions. Note that Sever 3 only supports one function.

predefined time before it goes offline, allowing process coordinators to update their records. If, on the other hand, it fails, this will be detected and reported when attempting to invoke functions on it, triggering its removal and the rescheduling of jobs.

4.6.2 Program Submission and Execution

When Martlet programs are submitted, they are converted into abstract syntax trees and stored, since at this point there is insufficient information to do anything more. However, when they are executed with a full set of arguments, all the required information is present, allowing the following chain of events to take place:

1. The handles of the arguments are locked and a copy of the Martlet program abstract syntax tree is made to protect the structure of the original. Copying the abstract syntax tree leaves the original available for use in other concurrent and future function invocations.

2. The abstract syntax trees of any function calls to other Martlet programs are retrieved and recursively integrated into the tree, so that the only remaining function calls are calls to base functions. This step does mean that while Martlet can potentially support recursion, the middleware currently does not support recursion.

3. By examining the distributed data structures, the tree is expanded to remove the expandable constructs, replacing them with a set of normal constructs that represents their expandable counterparts for the given distributed data structures.

4. Since the full tree is now present, the base functions are scheduled to run on specific data processors chosen from the function sets stored on the process coordinator.

5. With the tree now fully quantified, it is automatically broken into pieces and distributed to the appropriate data processors.

6. The root node of the function is then invoked.

7. Once the execution completes, the sub-trees are disposed of, leaving just the returned results.

4.6.3 Deadlock Prevention

To ensure that deadlock does not occur and to guarantee simultaneously that race conditions do not prevent the generation of correct results, it is necessary for all execution requests that write to a given data structure concurrently to be submitted to the same process coordinator. For example, suppose there is a process `f` that adds the value in its first argument to its second argument, and the call `f(a,b)` is made to a process coordinator, then other requests using `b` can be made. However, until `f` completes, these requests are bound to being submitted to the same process coordinator, even if the second call only reads `b`. This occurs because, while it is possible to use locking on a single server to protect jobs, it is not possible to do this across a potentially unknown set of servers. This is not a severe restriction on the user as there is no advantage gained by using multiple process coordinators.

4.6.4 Scheduling

With the wide range of schedulers already in existence for workflow languages [18, 30, 88, 118], and the potential for vastly different operating environments, the development of an advanced scheduler was not one of the aims of this project. As such, the current scheduler is very rudimentary. To facilitate the construction and use of more advanced or appropriate schedulers, an interface is provided for schedulers to implement to allow them to interact with the process coordinator. In addition to this a *visitor* interface is provided allowing the construction of custom visitors for traversing the expanded abstract syntax tree. As the tree is capable of handling an arbitrary number of visitors, and the visitor controls its movement through the tree, a huge degree of freedom is afforded to schedulers wishing to analyse and manipulate these trees.

4.7 Conclusion

This chapter has introduced a high level specification for a Service-Oriented middleware and a Web Service based prototype implemention to support Martlet workflows. Collectively, these provide a fairly pure insight into what is needed to run such workflows, and while some parts of the implementation could now be replaced by existing middleware, other pieces cannot be so easily replaced. As such, this work provides a checklist when considering integrating or modifying other middleware package to support Martlet, or when adding in ideas derived from Martlet into other middleware.

In Chapter 5, we use this middleware to test the case studies introduced in the Chapter 1 along with some of the case studies used in Dean and Ghemawat's MapReduce paper [41]. We then in Chapter 6 examine in more depth how this middleware can be further integrated into other solutions. This discussion continues the principles this project was started on of not segregating this work from the rest of the field.

Chapter 5

Case Studies

In this chapter we examine four case studies: North Atlantic Oscillation, Word Count, Grep and Singular Value Decomposition (SVD). These four case studies were chosen for a range of reasons. The North Atlantic Oscillation makes use of the average function that has been used to motivate this work so far, so this will take this example to completion. The Word Count and Grep are used in Dean and Ghemawat's MapReduce paper [41] as examples, so it is useful to show them here in comparison. Finally, the SVD demonstrates the ability to construct more complex operations.

5.1 North Atlantic Oscillation

The North Atlantic Oscillation (NAO) is a climatic phenomenon where the sea-level pressure between the Icelandic low and the Azores high in the North Atlantic Ocean fluctuates. This fluctuation controls the strength and direction of westerly winds and storm tracks across the North Atlantic. When measuring this, pressures are taken at these two points and the difference is computed. If this needed to be computed this over a set of models, this would be done by then taking the mean of the results. We consider this example important because the basic style of the computation encompasses a huge number of the operations that are desirable to perform on an ensemble of data.

In Figure 5.1, we have the code for this operation, which includes a call to the provided base function in Figure 5.2 for computing the mean. This base function is also described in Martlet, once again calling to other base functions. If the mean had not been provided as a base function, the user could first submit it as a user defined function, and then used the `define` block at the

```
define
{
  base = BaseFunction:all:http://userpc55.ox.ac.uk:8080/Martlet;
}

proc(icelandicPressure, azoresPressure, result)
{
  difference = new DisMatrix(icelandicPressure);

  map matrixSubtract:$base$(icelandicPressure, azoresPressure, difference);

  matrixMean:$base$(difference, result);

}
```

Figure 5.1: Martlet code to compute the North Atlantic Oscillation.

top of their second function to rename the returned URI so keeping the function readable. The observant reader may notice that the mean function used here is different to the mean used in the earlier examples. This difference occurs because the earlier example was an idealised version that assumed that number overflow would not be a problem. This version, while slightly longer, so not as good for use in the earlier examples, does not make this unrealistic assumption.

5.1.1 Analysis

To demonstrate the operating characteristics of Martlet, a series of tests were run on a test bed of ten servers storing some 45,000 model runs. To enable these tests to be carried out on the largest available dataset, we decided to apply the mean function directly so as not to restrict ourselves to just the two values per run used in the NAO. In these tests, the following information was recorded from each server: time spent executing base functions; time spent adjusting abstract syntax trees and scheduling functions; time spent communicating between servers; and out of the time spent communicating, how much of that was sending data.

In the first test, the number of values averaged per run was steadily increased, allowing us to observe the performance characteristics of the test bed as the size of the input data increases. The total times for this test can be seen in the following table and are shown graphically in Figure 5.3. The times for each individual server can be seen in Appendix Section C.1.

```
define
{
  base = BaseFunction:all:http://userpc55.ox.ac.uk:8080/Martlet;
}

proc(input, output)
{
  partialAverage = new DisMatrix(input);
  noRuns = new DisInteger(input);

  map
    async
    {
      matrixAverage:$base$(input, partialAverage);
      matrixCardinality:$base$(input, noRuns);
    }

  total = new Integer(output);

  tree((left,right)\noRuns->total)
    numberSum:$base$(left, right, total);

  map
  {
    scale = Real(input);
    numberDivide:$base$(noRuns, total, scale);
    matrixScale:$base$(partialAverage, scale, partialAverage);
  }

  tree((left, right)\partialAverage->output)
    matrixSum:$base$(left, right, output);
}
```

Figure 5.2: Martlet code to compute the mean of a matrix without problems with number overflow.

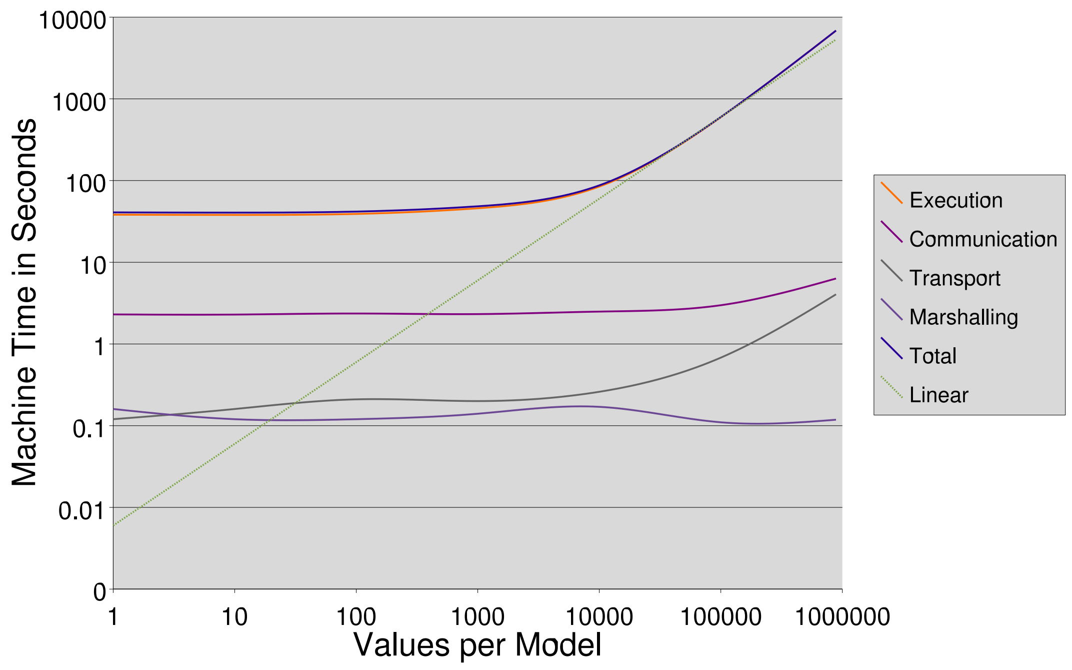


Figure 5.3: A graph of the time spent doing different tasks as the size of the input dataset is varied. “Linear” shows the theoretical growth rate of the analysis function alone.

Global Totals	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	38.2	37.95	39.03	45.77	84.42	595.79	7858.33
Communication	2.3	2.29	2.36	2.32	2.49	2.98	6.65
Transport	0.12	0.16	0.21	0.2	0.26	0.68	4.52
Marshalling	0.16	0.12	0.12	0.14	0.17	0.11	0.12
Total	40.78	40.52	41.72	48.43	87.34	599.56	7869.62

This shows that, as expected, while the overhead times remain constant, the execution times and transport times rise linearly with the size of the input data. The initial plateau in both the execution and transportation times is caused by the cost of constructing the function objects and supporting data structures. This time for very small datasets is a significant part of the total time spent on these tasks, however all the overheads are very small in respect to the overall computation for all datasets bigger than a couple of megabytes per machine.

The second test measured the time required to run the same calculation across a differing number of machines. To do this we took 9000 model runs, each containing 1,000,000 values, and calculated their average multiple times while steadily increasing the number of machines from 2 through to 10. The results from these tests can be seen in the following table, and the

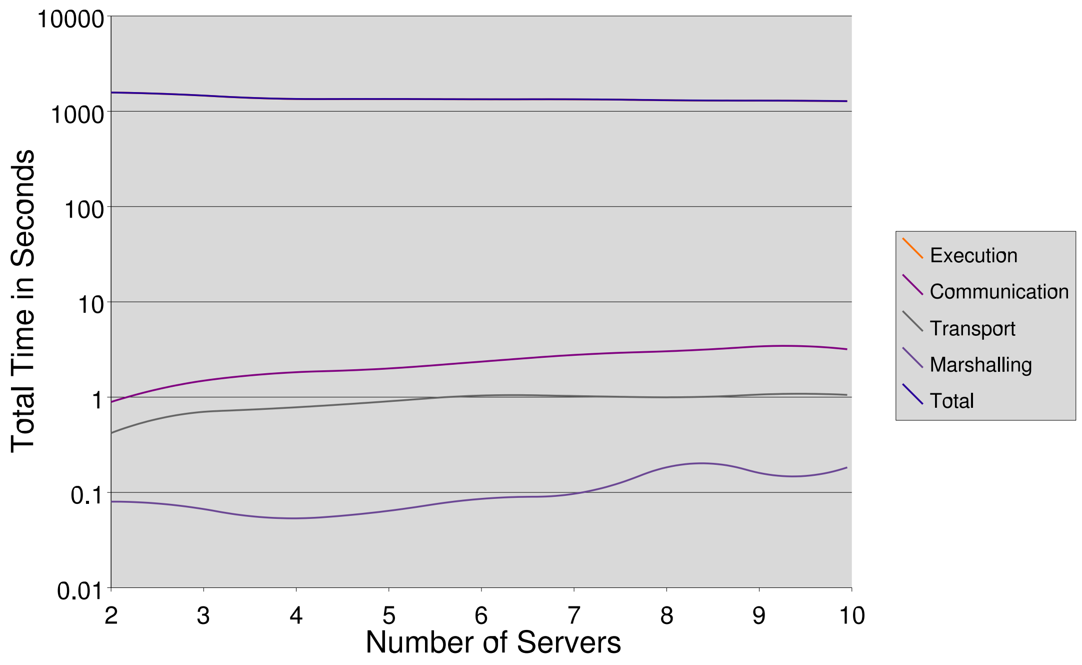


Figure 5.4: A graph of the time spent doing different tasks as the number of the data processors is varied. Note that while the marshalling time varies greatly, it is still very small, and the discrepancy is probably due to inaccuracies in the measurements.

full results can be found in Appendix Section C.2. A graphical representation of these results can be seen in Figure 5.4.

Global Totals	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	1572.7	1540.8	1335.4	1346.5	1331.2	1336.2	1290.2	1291.0	1271.6
Communication	0.89	1.33	1.81	1.93	2.36	2.86	3.07	3.64	3.14
Transport	0.42	0.68	0.75	0.88	1.07	1.02	0.98	1.12	1.05
Marshalling	0.08	0.08	0.05	0.06	0.09	0.09	0.25	0.12	0.19
Total	1573.2	1541.8	1336.9	1348.2	1333.4	1338.9	1293.2	1294.5	1274.7

This shows that while the marshalling and communication costs increase slightly as the number of servers increases, the total execution costs remain approximately constant. This means that expanding this model out to many servers is highly practical. In fact, the data shows a decrease in the overall execution time; we believe that this is due to a range of causes, including that, because the execution is being run with garbage collection, as the number of servers increases, the number of garbage collections decreases due to the increasing overall heap size.

Summary

As the middleware that these tests were run on was constructed as a prototype, it is worth bearing in mind that it is not the absolute values that are important here, but the relative trends of the values. Collectively, these tests show that there is very little overhead cost incurred when adding more nodes or expanding expandable statements at runtime. As such, this programming model is both highly scalable, and as efficient as the existing less-dynamic languages and supporting middlewares. While it can be argued for the second test that the cost of the reduction is small at each step, the test shows that the surrounding costs of building such reductions is very small; as such, the expense of more complex reductions can easily be predicted using standard reasoning about the algorithm being performed.

5.2 Word Count and Grep

Word Count and Grep [107, pp. 79–80] are two of the case studies constructed in C++ in Dean and Ghemawat's MapReduce paper [41]. Here we show how, with the appropriate rough-grained functions, they can also be described in Martlet. In Section 6.2.3 we discussed future work could include the ability to embed other languages into a Martlet workflow. In this case, it may be possible to construct components such as these without the need to have specific functions deployed within the framework. However in a framework intended for analysing large volumes of text, it is not unreasonable to expect such functions to be available.

Figure 5.5 shows the code for a distributed Grep. This code uses a fold for the reduction operation to limit the parallelisation; this is done because the transport costs will be much higher than the computation costs. In Figure 5.6, we have the code for a distributed word count, taking a distributed string of words as an input, and returning a string of words and a vector as outputs.

5.2.1 Analysis

Due to the highly data dependent nature of these functions, empirical tests would provide a minimal additional insight over that already gained through the previous case study. As such, we are only going to consider the time complexities under the assumption of sensible implementations of the called lower-level functions. Making these assumptions, Grep has an average time complexity for the base case of $O(n)$ and a worst case of $O(n^2)$. For the inductive case, Grep has a time complexity of $O(1)$. Word Count has an $O(n)$ time complexity for both the base

```
// Declare URI abbreviations to improve the script readability
define
{
   uri1 = BaseFunction:system:http://cpdn.net:8080/Martlet;
}

proc(input, pattern, output)
{
// Declare the required local variable for the computation. This will store
// the result of grep on each local data structure.
   localGrep = new DisString(input);

// Grep the local data structures,
   map grep:$uri1$(input, pattern, localGrep);

// and concaternate the results into the output.
   foldr stringConcatenate:$uri1$(output, localGrep, output);
}
```

Figure 5.5: The Martlet code for a distributed Grep. Foldr is used because transport times will be larger than the computation times.

```
// Declare URI abbreviations to improve the script readability
define
{
   uri1 = BaseFunction:system:http://cpdn.net:8080/Martlet;
}

proc(input, outputWords, outputFrequency)
{
// Declare the required local variables for the computation. These will
// store the results from each local data structure.
   localWords = new DisString(input);
   localFrequency = new disinteger(input);

// calculate the word count for each local piece,
   map
      wordCount:$uri1$(input, localWords, localFrequency);

// and merge the results.
   tree((words1, words2)\localWords -> outputWords,
        (frequency1, frequency2)\localFrequency -> outputFrequency)
      wordFrequencyMerge:$uri1$(words1,
                                words2,
                                outputWords,
                                frequency1,
                                frequency2,
                                outputFrequency);
}
```

Figure 5.6: The Martlet code for a distributed word count, taking a distributed String of words as an input. It returns a String of words and a column vector holding the frequency of each word in the order it appears. Note that each word will appear in the output vector only once.

case and the inductive case. However, in the inductive case, n is the size of the data from the previous step, not the size of the original input. With this in mind, while it would be expected to see higher data transfer times and execution times for these functions, it is not expected that the profile of the other times would differ much for that seen in the North Atlantic Oscillation.

5.3 Singular Value Decomposition

An example of the more advanced analysis techniques we would like to perform, which requires more than a single pass over the dataset, is the Singular Value Decomposition (SVD) of a matrix. The SVD of a matrix A is a factorisation that breaks the matrix down into three matrices, U, Σ and V^T, such that:

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n}$$

Here, Σ is a nonnegative, diagonal matrix, containing the singular values of A, and with the constraint on that all entries $\sigma_{i,i} \geq \sigma_{i+1,i+1}$. U and V are orthogonal, with the column vectors of V containing the corresponding singular vectors of A, and U containing the normalisation of their images under A.

This is of interest because, the property that V contains the singular vectors means that the maximum size change of a p dimensional subspace under the mapping defined by A can be read from Σ and the column vectors in V. This property is used in image processing, signal processing, and gene recognition techniques to remove less significant information from the mapping. In these cases, the SVD is computed, and the smaller entries of Σ are zeroed. The factorised matrices are then multiplied back together to produce a low-rank matrix that contains the most accurate possible space transformation defined with the reduced number of dimensions.

The SVD is of interest to climate*prediction*.net because the leading vectors in V give a linear combination of the model runs that affect the output most. This can then be indexed back against the original perturbed condition from which the models were run to determine the perturbations that make the biggest difference to the model.

The are several algorithms for calculating the SVD of a matrix. We first consider the Golub-Kahan method and the Lanczos method and evaluate them with respect to their applicability both to the loosely coupled distributed target environment provided, and to the programming model provided by Martlet. Then we examine a method proposed by the author and Hauser [68], this method is specifically designed with large highly distributed datasets in mind.

5.3.1 Golub-Kahan Method

The classic way to produce an SVD is the Golub-Kahan method [58, pp. 452–457]. This uses Givens rotations, and potentially Householder matrices [58, pp. 208–221] to stepwise construct the factorisation.

Givens Rotations

A Givens Rotation $G(i, k, \theta)$ is a counter-clockwise rotation by an angle of θ in the (i, k) coordinate plane. This can be used to zero individual elements within a matrix by carefully choosing the value of θ. A Givens Rotation defines an orthogonal matrix as follows.

$$G(i,k,\theta) = \begin{array}{cc} \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} & \begin{array}{c} \\ \\ i \\ \\ k \\ \\ \\ \end{array} \\ \begin{array}{ccccccc} & & i & & k & & \end{array} & \end{array}$$

where

$$c = cos\theta$$

$$s = sin\theta$$

Householder matrices are similar; however, they perform a reflection that allows all but one entry in a column or row to be zeroed in a single step, instead of the repeated application required when using Givens Rotations. We will only mention Givens Rotations here though, as they are more appropriate to parallel applications.

Using Givens Rotations to Compute the SVD

The monolithic way of computing an SVD is to use Givens rotations or Householder matrices to produce a bidiagonal matrix B such that $A = UBV^T$, where U and V are orthogonal matrices [58, pp. 251–252]. Then, through an iterative sequence of Givens rotations, the off-diagonal elements are pushed towards zero until they are small enough to be deemed to be zero. Calculating appropriate permutation matrices to ensure the diagonal entries are monotonically decreasing completes the SVD. This method has some promise for parallelisation, because each Givens rotation only affects two rows and two columns.

Existing research into parallelisation of the SVD using Givens rotations is mostly aimed at real-time signal processing on closely coupled processors [10, 21, 22, 23, 131, 132, 133]. However, some of the ideas may extend to less closely coupled machines.

The factorisation shown in these papers is not quite the same SVD as described, as it has the requirement that $m \geq n$ and Σ is square. This means U is no longer explicitly square. As Σ is diagonal, the non-calculated part of U does not affect the factorisation, and is just restricted to being the orthogonal complement of the calculated part of U. Therefore, it can be calculated separately if it is required. This alternative factorisation of the SVD is sometimes called the "thin SVD".

This technique first creates an orthogonal matrix V such that

$$AV = W$$

where W has orthogonal columns. It then factorises W such that

$$U\Sigma = W$$

where each column in U has unit norm. This is done by iteratively building V as follows. Initially

$$V^0 = I$$

and

$$A^0 = A$$

Then for every pair of columns in A, a plane (Givens) rotation is applied to make them orthogonal to each other. This entire process is iterated until all the columns are effectively orthogonal. Each transformation that is applied to A has its inverse applied to V. This increments k and preserves the relationship

$$A^k(V^k)^T = A$$

As all of the transformations are orthogonal, the orthogonal nature of V is also preserved.

Summary

This method is not practical for the proposed environment, because the whole of the SVD must be calculated and this has a space requirement of $O(mn)$, which in the case of CPDN, equates to somewhere between 4.5TB and 14TB of storage. If each iteration is taken to be when every vector in A has been normalised against every other vector in A, then for each iteration, every vector in A is at some point on the same computing resource as every other vector in A in order to have a rotation applied. Moving vectors around to achieve this creates a communication complexity of $O(mn)$ also equating to somewhere between 4.5TB and 14TB for CPDN per iteration, of which there may be many.

5.3.2 Lanczos Method

Another technique is the Lanczos method [58, pp. 495–496]. This technique makes it possible to calculate just the leading part of the SVD for $A \in \mathbb{R}^{m\times n}$, thereby reducing the storage, communication and time complexity of the problem. It does this by incrementally calculating a bidiagonal matrix B such that $B = U^TAV$, where $U^TU = I$ and $V^TV = I$, and using the fact that the singular values of B are the same as the singular values of A. Describing U, V, and B as follows:

$$U = [u_1, \ldots, u_m]$$

$$V = [v_1, \ldots, v_n]$$

and

$$B = \begin{bmatrix} \alpha_1 & \beta_1 & & \cdots & 0 \\ 0 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{n-1} \\ 0 & \cdots & & 0 & \alpha_n \end{bmatrix}$$

and considering the equations $AV = UB$ and $A^T U = VB^T$, we obtain for k in the range $[1..n]$:

$$\begin{aligned} Av_k &= \alpha_k u_k + \beta_{k-1} u_{k-1} \\ A^T u_k &= \alpha_k v_k + \beta_k v_{k+1} \\ \beta_0 &\equiv 0 \\ \beta_n v_{n+1} &\equiv 0 \end{aligned}$$

From these it is possible to create a loop and initialisation condition to calculate all the values of B, U and V. This is done by defining

$$\begin{aligned} r_k &= Av_k - \beta_{k-1} u_{k-1} \\ p_k &= A^T u_k - \alpha_k u_k \end{aligned}$$

giving the following equations

$$\begin{aligned} \alpha_k &= \pm \|r_k\|_2 \\ u_k &= \frac{r_k}{\alpha_k} \\ \beta_k &= \pm \|p_k\|_2 \\ v_{k+1} &= \frac{p_k}{\beta_k} \end{aligned}$$

Summary

This technique is good with sparse matrices because it does not alter A, and is used extensively in search engines because of this property. It has good space, time, and communication complexities; however, it requires vector multiplication by A and A^T during every iteration. This means every node in the system is interlocked at every iteration. For a distributed system, such a level of interlocking is inappropriate, as communication latency will become a limiting factor, and the reliability of nodes is not guaranteed. This makes this method inappropriate for distributed systems, as well as impossible to implement in Martlet.

5.3.3 A Variation on the Block-Lanczos Method

In [68] we propose a technique based on the Lanczos method, that provides the loose coupling required for such environments. This technique works by separately computing the p leading entries of the SVD for the data stored locally on each node. The SVDs of two parts of the data structure can then be combined and used as a new local dataset. This combined dataset is a fraction of the size of the original two, so overcoming the problems of data size. This process can then be iterated, creating a loosely coupled tree of execution, allowing for the computation of arbitrarily large datasets. An example of this occurring for a tree containing three leaves is shown in Figure 5.7. If required, the result can be taken and used to seed another iteration, so ensuring a high level of accuracy is in the final result.

Both the branch and leaf nodes of the tree have the same core function, and only differ in the mappings used to map from the input and output of this core function to the input and output of the node itself.

Core Function

The core calculation just requires a single input matrix $A \in \mathbb{R}^{m \times n}$, and returns two matrices W and V equivalent to $U\Sigma$ and V in the SVD of A. This calculation relies extensively on a factorisation called the QR factorisation [58]. In a QR factorisation the input matrix is factorised into two matrices Q and R where Q is an orthogonal matrix and R is an upper triangular matrix.

A matrix Q is initialised as a seed by the following equality to provide an orthogonal matrix draw form the initial input data.

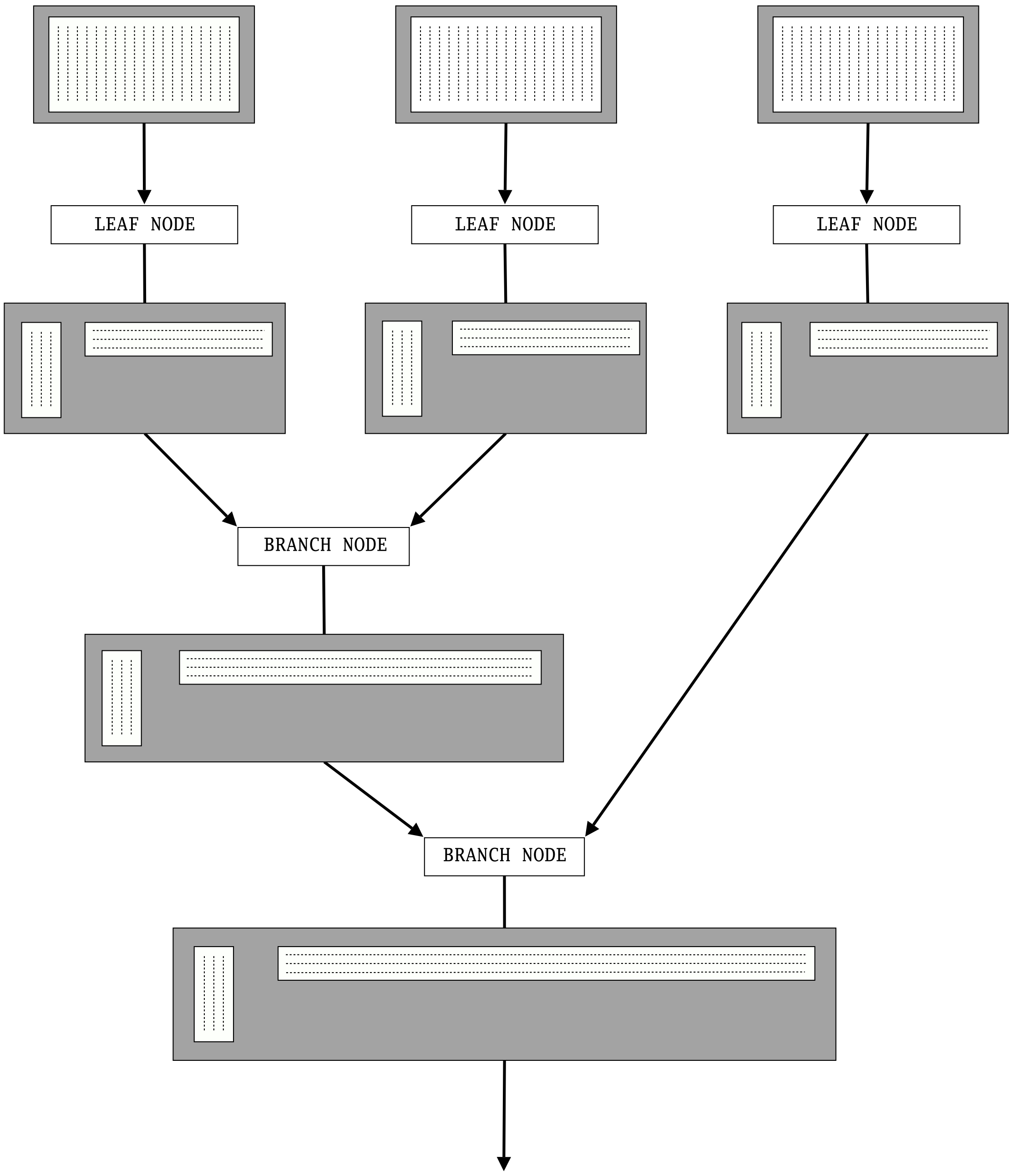


Figure 5.7: An outline of the flow of data for a tree with three leaf nodes.

$$QR = qr\left(\begin{bmatrix} A_{1,1} & \cdots & A_{1,2p} \\ \vdots & \ddots & \vdots \\ A_{m,1} & \cdots & A_{m,2p} \end{bmatrix}^T\right)$$

The following assignments are then iterated until the normalised change to Σ is less than some error ξ.

$$\begin{aligned}
U\Sigma V^T &= svd(AQ) \\
X &= Q\begin{bmatrix} V_{1,1} & \cdots & V_{1,p} \\ \vdots & \ddots & \vdots \\ V_{2p,1} & \cdots & V_{2p,p} \end{bmatrix} \\
Q'R' &= qr\left(A^T\begin{bmatrix} U_{1,1} & \cdots & U_{1,p} \\ \vdots & \ddots & \vdots \\ U_{m,1} & \cdots & U_{m,p} \end{bmatrix}\right) \\
QR &= qr\left(\left[\begin{array}{cc} X & \begin{bmatrix} Q'_{1,1} & \cdots & Q'_{1,p} \\ \vdots & \ddots & \vdots \\ Q'_{m,1} & \cdots & Q'_{m,p} \end{bmatrix} \end{array}\right]\right)
\end{aligned}$$

This method does use an invocation of another singular value decomposition internally, however this invocation is on the matrix AQ, which is $n \times 2p$, so far smaller than the original matrix A. The complexity of this call can be reduced further because of the knowledge we posses about the shape of the input matrix. First we construct U' and P' such that;

$$U'P' = qr(AQ)$$

then we compute the SVD of P' such that;

$$U''\Sigma V^T = svd(P')$$

as P' is a $2p \times 2p$ matrix this SVD is very quick. We then complete the SVD by the constructing U with the statement $U = U'U''$. This gives an overall complexity for the internal SVD of

$O(mp^2)$.

Once Σ has converged in the core iteration, the matrices to return are generated by the assignments

$$\begin{aligned} V &= \begin{bmatrix} Q_{1,1} & \cdots & Q_{1,p} \\ \vdots & \ddots & \vdots \\ Q_{m,1} & \cdots & Q_{m,p} \end{bmatrix} \\ W &= A\begin{bmatrix} Q_{1,1} & \cdots & Q_{1,p} \\ \vdots & \ddots & \vdots \\ Q_{m,1} & \cdots & Q_{m,p} \end{bmatrix} \end{aligned}$$

Leaf Nodes

For leaf nodes as A is the local dataset, A can be used directly and, V and W require no additional work so are returned as they are. This is shown in Figure 5.8.

Branch Nodes

Branch nodes can have an arbitrary number of children, but for the sake of this example we will assume that they only have two children. This means they will receive the values W_1, W_2 and V_1, V_2 as input. A is then defined as;

$$A = \begin{bmatrix} W1 & W2 \end{bmatrix}$$

After the core function is applied to A the returned values W_c and V_c need mapping to W and V ready to be returned. W does not require a transformation, but V_c needs the information about how the columns in W map to the columns in the original dataset. This is defined by the relationship;

$$V = \begin{bmatrix} V_1 & 0 \\ 0 & V_2 \end{bmatrix} V_c$$

This data flow can be seen in Figure 5.9.

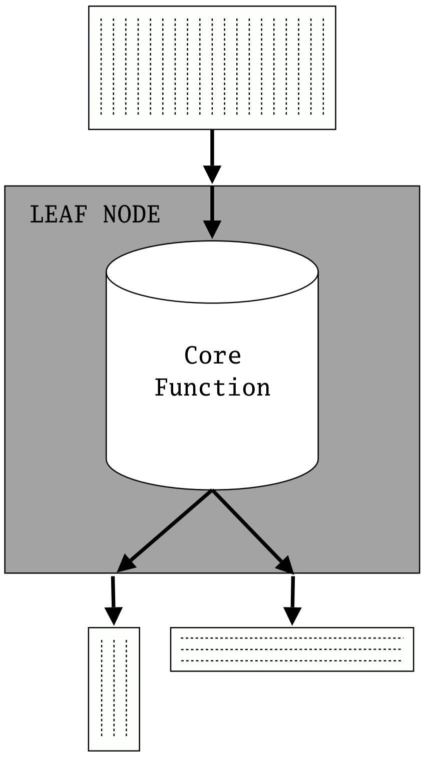


Figure 5.8: An outline of the flow of data within a leaf node.

The Final Step

Once the tree has been evaluated, and the matrices W and V have been produced, it is necessary to construct U and Σ. Σ is produced by taking the Σ from the last call of the core function, and sub-setting it to a $p \times p$ matrix containing the first p values. U is then produced by the statement $U = W\Sigma^{-1}$. Thus completing the SVD of the leading p vectors and values.

Summary

This method provides the required loose coupling to implement in the environment presented by CPDN, and also demonstrates some other very appealing properties. These properties and its performance are discussed in Section 5.3.6 after we have introduced a Martlet implementation in the next section and in Appendix Section D.2 a Matlab [81] implementation that was used in the analysis.

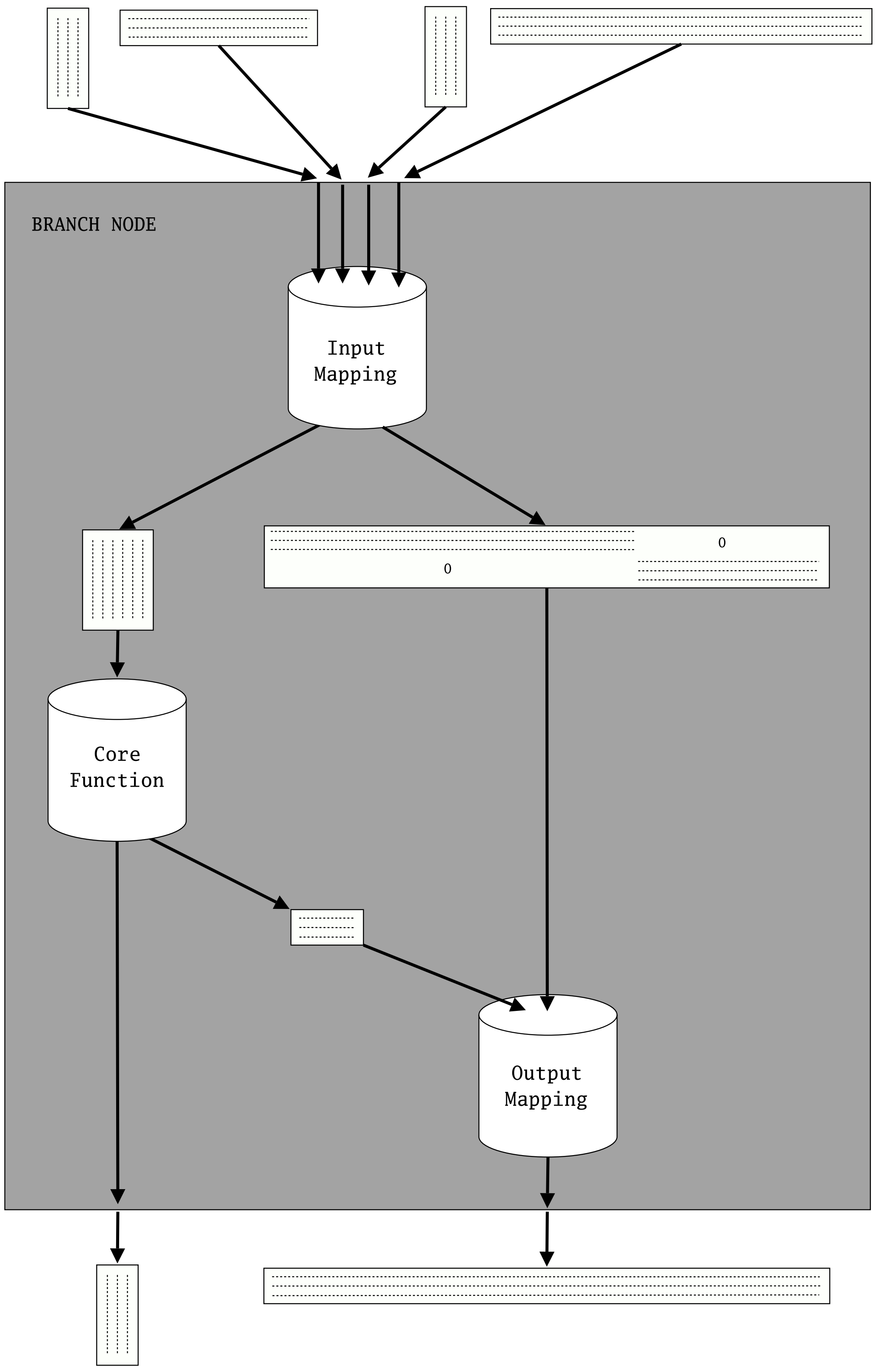


Figure 5.9: An outline of the flow of data within a branch node.

5.3.4 Martlet Code

Because of the loose coupling provided by the variation on the Block-Lanczos method, we are now able to introduce Martlet code implementing an SVD of a large, dynamically partitioned and distributed dataset. The body of this function first uses a `map` statement to construct the initial R and V values before using a `tree` statement to reduce these local answers to a single answer representing the leading p vectors and values for the entire distributed matrix A. To achieve this, it uses two helper functions shown in Figures 5.10 and 5.11. The first of these encapsulates a single iteration of the statements in the core function, and second uses this to construct a single method that represents the complete core function. These mean that the main method only has to manage the marshalling of the inputs and outputs for the application of this core method to the leaf and node cases.

```
// Declare URI abbreviations to improve the script readability
define
{
   uri1 = BaseFunction:system:http://cpdn.net:8080/Martlet;
   localCalculation =
            4298:Function:daniel:http://cpdn.net:8080/Martlet;
}

proc(A, U, S, V, P, err)
{
  splitU = new DisMatrix(A);
  splitS = new DisMatrix(A);
  splitV = new DisMatrix(A);

  map
  {
    two_P = new Integer(A);
    numberSum:$uri1$(P, P, two_P);

    readOnlyTranspose:$uri1$(A, splitV);
    matrixHorizontalSplitLeft:$uri1$(splitV, splitV, P);

    $localCalculation$(A, splitV, splitS, P, two_P, err);

    matrixMultiply:$uri1$(A, splitV, splitU);
  }

  tree((U1, U2)\splitU->U, (S1, S2)\splitS->S, (V1, V2)\splitV->V)
  {
    Width = new Integer(V1);
    T = new Matrix(V1);

    matrixCardinality:$uri1$(V1, Width);
    zeroMatrix:$uri1$(T,Width,P);
    matrixConcatenation:$uri1$(V1, T, V1);
    matrixCardinality:$uri1$(V2, Width);
    zeroMatrix:$uri1$(T,Width,P);
    matrixConcatenation:$uri1$(T, V2, V2);

    VT = new Matrix(V1);
    matrixVerticalConcatenation:$uri1$(V1, V2, VT);

    A = new Matrix(V);
    matrixConcatenation:$uri1$(U1, U2, A);

    two_P = new Integer(A);
    numberSum:$uri1$(P, P, two_P);

    Q = new Matrix(V);
    readOnlyTranspose:$uri1$(A, Q);
    matrixSplitTop:$uri1$(Q, Q, P);

    $localCalculation$(A, Q, S, P, two_P, err);

    matrixMultiply:$uri$(A, Q, U);

    matrixMultiply:$uri$(VT, Q, V);
  }

  readOnlyTranspose:$uri1$(V,V);
  matrixSplitLeft:$uri1$(V,V,P);

  matrixSplitLeft:$uri1$(S,S,P);
  matrixSplitTop:$uri1$(S,S,P);

  matrixSplitLeft:$uri1$(U,U,P);
  ST = new Matrix(S);
  inverseMatrix:$uri1$(S, ST);
  matrixMultiply:$uri1$(U, ST, U);
}
```

5.3.5 Alternative Data Partitionings

The algorithm as described here is only able to construct the SVD of data that is partitioned via a column-wise split, however through the use of recursion and transposition it is able to handle a much large set of data partitionings.

The first step is to handle the case where the data is partitioned row-wise instead of column-wise. Simply running the algorithm as normal over this data, and then transposing the result achieves this. Having handled the case where the data is partitioned row-wise it is now possible to hand the case where the data is split into blocks as shown in Figure 5.12. This is achieved by first calculating the leading part of the SVD for each block. Then by grouping together all blocks that appear in the same column, and using the algorithm for calculating SVDs over row-wise partitioned data on each of these groups it is possible to construct the leading part

```
// Declare URI abbreviations to improve the script readability
define
{
   uri1 = baseFunction:system:http://cpdn.net:8080/Martlet;
}

proc(A, Q, S, P, two_P)
{
  N = new Matrix(A); // A sink for unrequired results
  // Temporary values for the calculations
  T = new Matrix(A);
  U = new Matrix(A);
  V = new Matrix(A);
  X = new Matrix(A);

  matrixMultipy:uri1(A, Q, T);
  SVD:$uri1$(T, U, S, V);

  MatrixSplitLeft:$uri1$(V, V, P);
  matrixMultipy:$uri1$(Q, V, X);

  MatrixSplitLeft:$uri1$(U, U, P);
  readOnlyTranspose:$uri$(A, T);
  matrixMultiply:$uri1$(T, U, T);
  thinQR:$uri1$(T, Q, N);

  MatrixSplitLeft:$uri1$(Q, Q, P);
  matrixConcatenation:$uri1$(X, Q, Q);
  thinQR:$uri1$(Q, Q, N);

  MatrixSplitLeft:$uri1$(Q, Q, two_P);
}
```

Figure 5.10: The Martlet code for a single local iteration of the SVD algorithm. The code is split such that each block represents one line in the Matlab code in the Appendix. This code could be adjusted such that `T` and `N` are the same variable to save space; however, to aid readability, this is not done here.

of the SVD for each column. From this point it is then possible to use the original algorithm configuration to complete the result.

5.3.6 Analysis

The variation on the Block-Lanczos method implemented in this section has appealing properties, that allow it to compute just the leading part of the SVD in a rough-grained fashion, and without altering A. As the dataset can be broken down into a large number of small, independent pieces, the SVDs can be constructed from arbitrarily large datasets, and it is not necessary for the dataset to be complete at the time the calculation is started, as new data can be added to the tree in the form of a new leaf node at any time. For example, in the case of CPDN,

```
// Declare URI abbreviations to improve the script readability
define
{
   uri1 = BaseFunction:system:http://cpdn.net:8080/Martlet;
   SVD_iteration = 3345:Function:daniel:http://cpdn.net:8080/Martlet;
}

proc(A, Q, S, P, two_P, err)
{
  oldS = new Matrix(A);
  twoNorm = new Double(A);

  matrixCopy:$uri1$(S, oldS);

  $SVD_iteration$(A, Q, S, P, two_P);

  matrixSubtract:$uri1$(S, oldS, oldS);
  twoNorm:$uri1$(oldS, twoNorm);

  while( twoNorm > err )
  {
    matrixCopy:$uri1$(S, oldS);

    $SVD_iteration$(A, Q, S, P, two_P);

    matrixSubtract:$uri1$(S, oldS, oldS);
    twoNorm:$uri1$(oldS, twoNorm);
  }
}
```

Figure 5.11: The Martlet code for the local calculation of the SVD. It is split up such that each block represents a line of Matlab code in the Appendix.

this means that the SVD could be calculated for an initial set of columns. Then, when some additional results have been returned, instead of having to perform the calculation for the entire dataset again, it is possible to do the calculation just on the new results, and then use a branch node to combine this result with the existing result.

This is possible because this algorithm is only building results based on the largest p dimensional subspace from each of its children. It is therefore possible for it to not return the correct vectors and values as information is lost at each level of the tree. There are two main ways this can be overcome: First we can restart the calculation, using the output from the last iteration, instead of the QR of the first p vectors of A as a seed, thereby feeding the most significant information from the last iteration back into the calculation. Second, if the singular vectors are distinct enough, we can increase p to a slightly larger value than required. This results in the calculation of additional vectors, which carry sufficient additional information too ensure that the required vectors remain accurate. The extra vectors can then be discarded once the calcu-

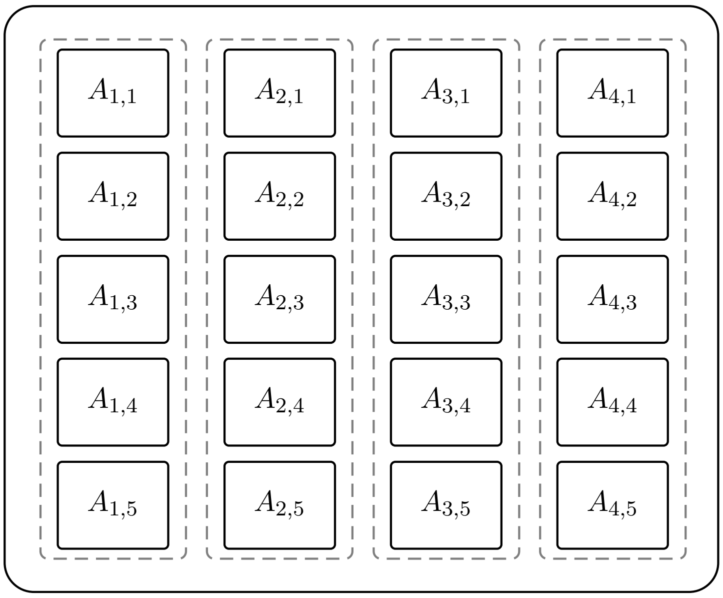


Figure 5.12: An example of a matrix A split both column-wise and row-wise. The SVD algorithm in Section 5.3.3, can be used on this matrix partitioning by, first running it on each of the individual elements, then grouping the results from these to form four sets of row vectors, represented by the grey dashed boxes. These groups can then have the method applied to each of them, producing four sets of column vectors. A final application of the method to these produces a single result for A.

lation is completed. Further analysis is required to work out the exact relationship between the type of input data, and the number of excess vectors required in order to maintain accuracy. Once such information is available, the possibility of an adaptive algorithm presents itself.

Simulation

As the SVD returned by this algorithm is only an approximation, the algorithm was implemented in Matlab [81] to test the accuracy of the results. This Matlab code can be seen in Appendix Section D.2. To ensure that an accurate comparison with the output results was possible, the starting matrix A was constructed by first constructing the matrices U, Σ and V that represent the matrices produced by an SVD. Then by multiplying these matrices together we produce an A that satisfies $U\Sigma V = SVD(A)$. This gives us a matrix A and the correct result of an SVD of A.

Three tests were chosen to measure the error in the returned results. They take the actual vector v_a and the returned vector v_r, and can be expressed as:

1. the angle between v_a and v_r

$$\left|\left(\arccos \frac{v_a \cdot v_r}{||v_a||_2 ||v_r||_2}\right) \bmod \pi\right|$$

2. the relative lengths of v_a and v_r under A

$$\left|\frac{||Av_a||_2}{||Av_r||_2} - 1\right|$$

3. the angle between v_a and v_r under A

$$\left|\left(\arccos \frac{Av_a \cdot Av_r}{||Av_a||_2 ||Av_r||_2}\right) \bmod \pi\right|$$

Using the simulation and these tests, the algorithm was tested against the following properties:

Matrix Size This showed the interesting property that as the size of the matrix *increased*, the accuracy of the approximations *improved*.

Tree Depth This showed that for a single node the answers are accurate to machine precision. The change to a tree introduced small errors, which increase very slowly as the tree depth increases and are insignificant relative to the error increase between a given vector and the next most significant vector.

Value of p Here the accuracy of vectors on a large tree was measured as the number of returned vectors was increased. This showed that the only vector with significant errors was the least significant vector. As discussed, rerunning the algorithm and seeding each of the leaf nodes with the output from the first iteration, so avoiding the loss of data, could remove this error.

All results are included in Appendix Section D.1. These results show a high level of accuracy is possible for very large, highly-distributed matrices with relatively little communication and processing required. Collectively these results represent a dataset in the order of the size presented by CPDN.

Algorithm Complexities

The only tight complexities that can be calculated for this algorithm are space and communication complexities, which in both cases are $O(np+mp)$. The time complexity is data dependent, as while upper bounds can be calculated [68] for the number of iterations of the core function before convergence, the speed of the convergence is dependant on data in question. As such, timing data gathered from running this algorithm over the CPDN dataset would tell us very little, as it would not be possible to separate out the effect of the different datasets. The time complexity for each iteration of the core function is less than $O(nmp)$ where $p << n, m$. Graphs showing the number of iterations before convergence for different size sample matrices with different p values are included in Appendix Section D.1.3. These show that, with the exception of very small matrices, the number of iterations of the core function required at each node is small and constant for a given style of data.

Comparing the SVD to the other case studies, there is no evidence that the distributions of the times varies from that observed with the mean, aside from the larger time complexities of the implemented algorithm and the quantifiably larger data transport costs. This means that as the size of the dataset increases, so does the time for the initial mapping to return, though the rest of the algorithm remains unchanged. By increasing the number of servers on a large dataset, the execution time for this algorithm for a given value of p goes down. This occurs because the most expensive step is the map; however, this is at the cost of accuracy, due to the increasing number levels in the tree. This therefore requires an increase in the value of p to get the same number of leading vectors to the same accuracy. Because of the ratio of processor power to network bandwidth, the optimal partitioning is highly dependent on the operating environment.

5.4 Summary

In this chapter, we have introduced four varied, real-world case studies of differing complexity. We then have gone on to demonstrate how solutions to these can be implemented with Martlet and its associated programming model. These solutions vary from trivial variations of existing algorithms, to the development of new algorithms. These new algorithms, as well as fitting this new programming model for dynamic distributed utility computing, are still applicable to other languages used in less dynamic environments, and their discovery raises the interesting possibility of a whole class of new algorithms waiting to be discovered once people start thinking in terms of this new model.

In addition to validating the programming model's descriptive power, we have also shown that an implementing middleware is able to hold desirable properties such as low transport, communication, and marshalling overheads in a way previously unmanageable in such a dynamic environment.

Chapter 6

Analysis

In this chapter we will examine how Martlet compares with some of the more comparable technologies introduced in Chapter 2. We then go on to examine how Martlet and its supporting middleware might develop into a fuller language that is better integrated into existing frameworks.

6.1 Comparison With Other Technologies

We have found no distributed projects that are able to adjust at runtime to reflect the partitioning of the data they are analysing. As such, a direct comparison with other projects has not been possible. Therefore, in this section, we will examine how Martlet compares with other functional and functionally inspired coordination languages, namely Functional Skeletons [39, 40], Data Parallel Haskell [29], and MapReduce [41]. These pieces of work use data structures and functional constructs to abstract the complexity of distributed and parallel systems.

6.1.1 Functional Skeletons

As discussed in Chapter 2, Functional Skeletons are used for programming clusters and parallel machines, where, as the name suggests, they provide a framework on which programs can then be constructed. The idea is that the skeleton provides all of the boilerplate code for process interactions with a specific architecture and using a particular function pattern. All the programmer has to do is flesh out the skeleton with the code describing what the program should do between these interactions by using techniques such as the provision of call back functions. Then, when a program is ported to a different architecture, the corresponding skeletons to support the program will have already been constructed, and all that will be required is for the code to be recompiled before it can be executed. The critical difference between this work and Martlet is that the pre-constructed skeletons are changed not to match the data, which changes from execution to execution, but to match the architecture, which is static for extended periods of time. As a result, skeletons tend to be very static, requiring significant input from the person who constructs them. This means skeletons are not currently applicable to the target problem domain.

However, if Martlet was scaled back to only include the expandable constructs and an existing language were used to describe the functions, calling, and being called by these. Martlet could then in many ways be thought of as a set of dynamically-constructed functional skeletons, which users use to coordinate programs as all the inter-process interactions would be hidden in the expandable constructs.

6.1.2 Data Parallel Haskell

In the 90s the NESL language [14] implemented Data-Parallel Computing [13], allowing the production of nested data structures that allowed more efficient implementations of parallel languages. This work is now being used by the Data Parallel Haskell project as a starting point for embedding a programming model that uses nested data parallelism into the Glasgow Haskell Compiler. This will greatly reduce the complexity faced by the user when trying to construct parallel programs, by making large parts of the parallelisation implicit. To achieve this, they have constructed two classes of data structure: the standard Haskell data structures, and *parallel arrays*. These can be considered in many ways analogues to the local and distributed data structures in Martlet. From these two types of data structure, it is then possible to automatically flatten structures, and then apply more conventional techniques to work out when it is safe to farm out function invocations to different threads to execute on different resources. An example of parallel arrays can be seen in Figure 6.1.

While Data Parallel Haskell is using a similar approach to the construction of data structures, and has produced good results so far, it is still very much a work in progress. It is currently aimed at parallel architectures and closely coupled machines, where the data can be thought of as a single piece. As such, it does not cover the same ground as Martlet. However, it would be fair to say that this research represents the application of the same ideas, where abstract data structures and constructs hide the complexity of parallelisation from the user, to a different

In data parallel Haskell, parallel arrays are denoted by the syntax `[:,:]`. So a dense vector of floating points could be constructed by the type declaration;

```
type Vector = [: Float :]
```

and a sparse vector could be represented as;

```
type SparseVector = [:(Int, Float):]
```

where the integer is the index of the non-zero elements in the vector, and the float is their value. This could then be used to construct a sparse matrix through the declaration:

```
type SparseMatrix = [:SparseVector:]
```

Because the parallel nature of the data structures is part of the type of the language, it is now possible for the user defined functions and the provided libraries to contain definitions taking this into account. The complier can then use these definitions to automatically separate parallel threads on different resources as required. An example of this is the dot product of a SparseVector and a Vector using the library function `sumP :: [:Float:] -> Float`, and *array comprehension*:

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP [: x * (v!:i) | (i,x) <- sv :]
```

First this produces another Vector by drawing the elements from `sv` and multiplying them with their counterparts in `v`. Here `!` means indexed by, so `(v!:i)` produces the element in `v` indexed by `i`. The sum of this new Vector is then computed in parallel to complete the dot product.

Figure 6.1: An example to the construction and use of parallel arrays in Data Parallel Haskell.

problem domain.

6.1.3 MapReduce

Independently developed at the same time as Martlet, MapReduce is one of the programming models created for programmers writing parallel applications at Google. This, like Martlet, uses functional constructs to abstract the parallelisation. Users specify three functions, one to be mapped over the raw data, one to partition this output into pieces for the next step, and one to be mapped over the output of the first two functions, reducing it to a set of results. A diagram outlining a complete MapReduce can be seen in Figure 2.1 on page 37. Google's implementation of this model is an API that works with the Google File System [57], allowing efficient scheduling of parallel calculations on data, while abstracting the complexity of the data storage and processing.

This highly successful model is aimed at the internal work of Google programmers, and is not appropriate for the environment targeted by this project for a range of reasons. Its rigid structure expects the user to write code in a fine-grained programming language such as C++, this requires users to be trusted, which is not a realistic restriction for many Grid applications including CPDN. The model also requires that the user supply information about the architecture that they wish to execute over; however, it should be possible to remove this requirement. Finally, it has issues with the amount of network traffic generated between the completion of the map stage and the start of the reduce stage, even in an environment such as Google. This is because all of the potentially huge output from the map phase must be sorted and redistributed before the reduction phase can begin.

However, it is interesting to compare the algorithms that can be run using MapReduce and the algorithms that can be run using Martlet, especially in the light of the discussion on extending Martlet and on using Martlet to improve the utilisation of clusters and Condor Pools that appears later in this chapter. In Dean and Ghemawat's paper on MapReduce [41], three algorithms are discussed: a distributed Grep, a distributed word count, and a distributed sort. Of these, Martlet is able to describe both Grep and the word count example. However, due to the high network capacity it requires, there is currently no construct for partitioning and sorting data from one distributed data structure into another, as occurs in the phase between the map and the reduce step in MapReduce. As such, it is not possible to implement sorting of large datasets in Martlet. This is due to the fact that in a sort, the answer size is equal to the input size, and, unlike MapReduce, where the reduction operation is just another mapping, Martlet employs a parallel prefix [74] and an accumulator style of reduction. Thess styles of reduction limit the final output to the size that can fit on a single machine, and the original premise for this work was that it was not possible to put all the data on one machine, so it had to be distributed.

There are algorithms that MapReduce cannot describe. Because MapReduuce only performs reductions through the use of a second Map, and because the structure of a MapReduce is very rigid, it is not possible in MapReduce to perform any calculation that requires a parallel prefix style reduction. While this does not affect Google and the style of calculations often performed in their operation, it does prevent MapReduce from describing a large class of problems.

An example of this is one of the analysis operations described in Chapter 5, a singular value decomposition returning the leading p vectors. This cannot be done efficiently using the MapReduce model, as the fixed number of steps — one map and one reduce — in any individual MapReduce prevents the steady merging of results from a range of separate locations. As such, while it is possible to chain together many MapReduces to make very complex analysis functions, it is not possible to describe a parallel prefix style reduction, so preventing such algorithms being described.

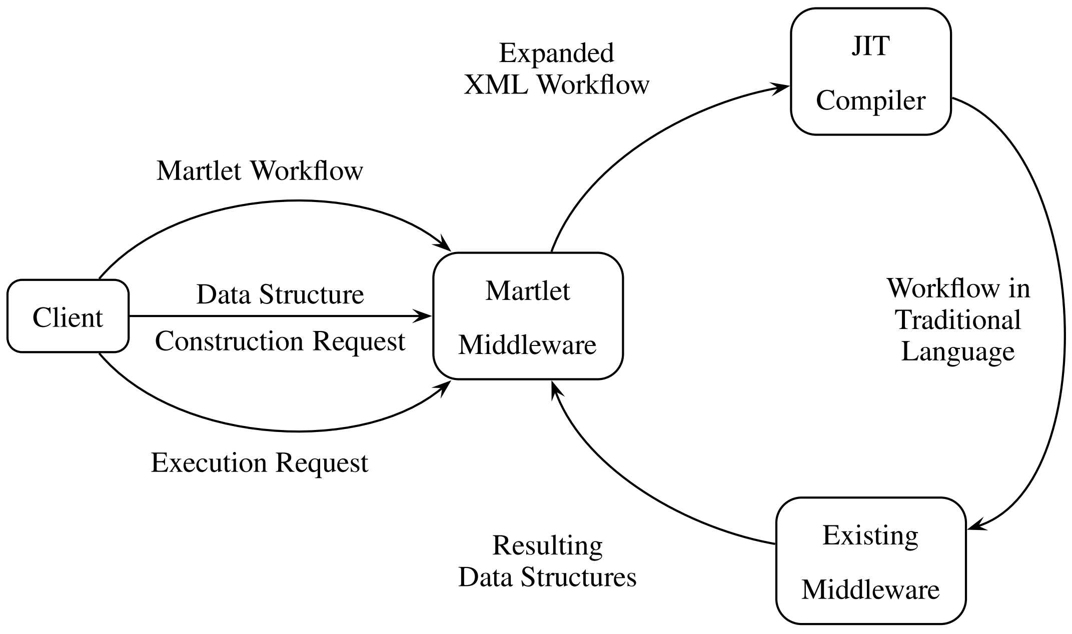
6.2 Further Development

In this section we examine ways Martlet can further develop in the future. These extensions fall into two specific categories: How Martlet might be extended so that it can operate in environments other than the prototype middleware that has supported it so far; and how the language itself may be developed to both improve its usability and extend the class of algorithms to which it is applicable.

6.2.1 Building on Other Workflow Engines

At runtime, when concrete values have been provided, the abstract Martlet functions are converted into concrete functions for execution. These concrete functions then contain only operations that are supported by a range of other workflow languages. As such, one logical development would be to provide a means by which these could be executed on existing middleware. This could be achieved by cutting back the Martlet middleware to just the data stores and the process coordinator. These could then sit on top of existing workflow engines as a layer supporting the construction of distributed data structures and the submission of abstract functions. This would then allow Martlet functions to be run on a wide range of middleware, in conjunction with a wide range of existing projects.

One of the neater ways to provide an interface between these layers is through the construction of a set of Just In Time (JIT) compilers for different workflow languages. If JIT compilers were used, the process coordinator would, just produce an XML document describing the concrete task instead of scheduling tasks across many machines. The JIT compliers would then perform an XML transformation to produce the language of choice, which could then be submitted to the appropriate underlying workflow engine. This would allow compilers for a range of languages to easily be produced, allowing this layer to be placed on top of a wide range of existing resources with minimal effort. This would extend their use without affecting their existing functionality. Such an extension would not only allow Martlet to provide extended functionality to a wide range of distributed computing applications, but also allow Martlet users to draw on all the work that has gone into these existing workflow projects. An outline of a request in this model is included below. Note that while they appear as single entities here, both sets of middleware could span many machines and locations.



With this extension in mind, a function constructor and a function object have been produced to take a function, a set of arguments to apply this function to, and a string. These, when executed, produce an XML document that represents the structure the function would take if executed over the provided arguments.

When these are invoked the function object copies and expands the function in the same way it would if it was going to execute it as normal. However, instead of next calling the scheduler and its visitors to distribute and execute the function, it uses a different visitor to construct a DOM tree [67] representing the expanded function. A graphical representation of the schema for this DOM tree can be seen in Figure 6.2; the source for the schema is in Appendix Section B.1. Once constructed, the DOM tree is then parsed into a string, which is stored in the string argument. It then can be retrieved through the publication of the string argument.

Using this system, an XML definition of the expanded syntax tree of the average function used in the example in Section 3.4 has been produced and can be seen in Appendix Section B.2. For each element, all the arguments needed to satisfy its subelements are listed. While this makes the description verbose, it was decided to do this as it is unknown which languages this XML will be transformed into, and it is easier to discard redundant information, than it is to regenerate missing information.

6.2.2 Working with Clusters and Condor Pools

The ability to move Martlet functions from the existing middleware onto other platforms provides the interesting possibility of using Martlet to improve the utilisation of clusters and Condor Pool style environments. While Martlet was constructed to allow functions to be run on arbitrarily partitioned data, it achieves this by partitioning the function into smaller functions at runtime. Therefore, if the data is in a form that can be automatically partitioned through the application of a *cutting function* at runtime, and the function is written in Martlet, then a job can be split into lots of smaller jobs like in MapReduce. This provides three benefits.

First, it provides another means for big jobs to be made parallel in environments such as clusters and Condor Pools. This can reduce the complexity that the user has to handle. While this may never be as good as handcrafted solutions, it has been shown, by the uptake of user friendly software that allows people to easily use many computers to tackle “embarrassingly parallel” problems, that there is a wide range of problems that do not require the highest level of refinement, and can benefit from more accessible parallel programming models.

Second, there is potential to improve the performance of such environments if they handle

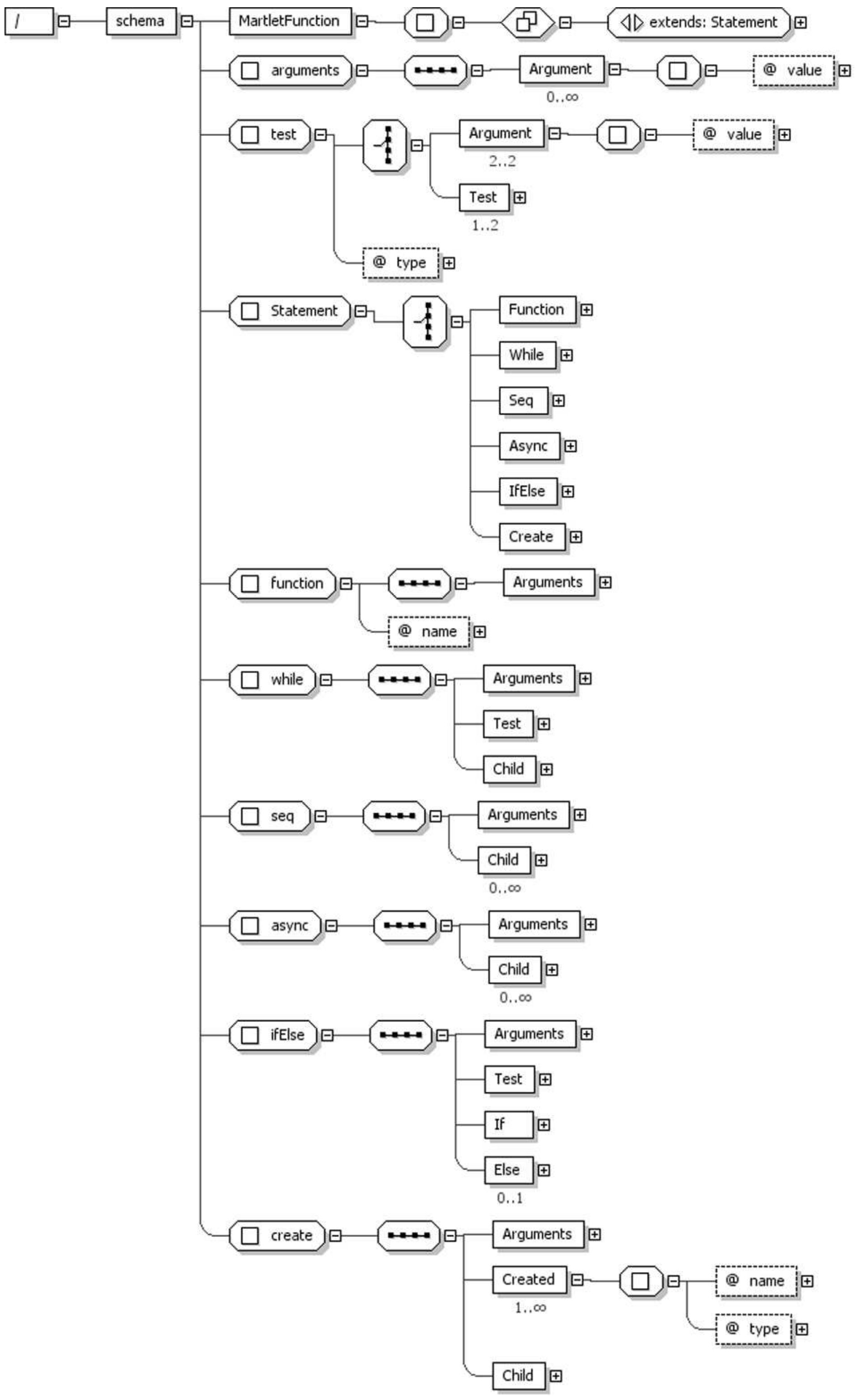


Figure 6.2: A graphical representation of the XML Schema describing the structure of XML documents produced to represent expanded Martlet functions. Most element names make their type obvious; all “Child” elements are of type “Statement”. As the functions are expanded it does not include any expandable constructs.

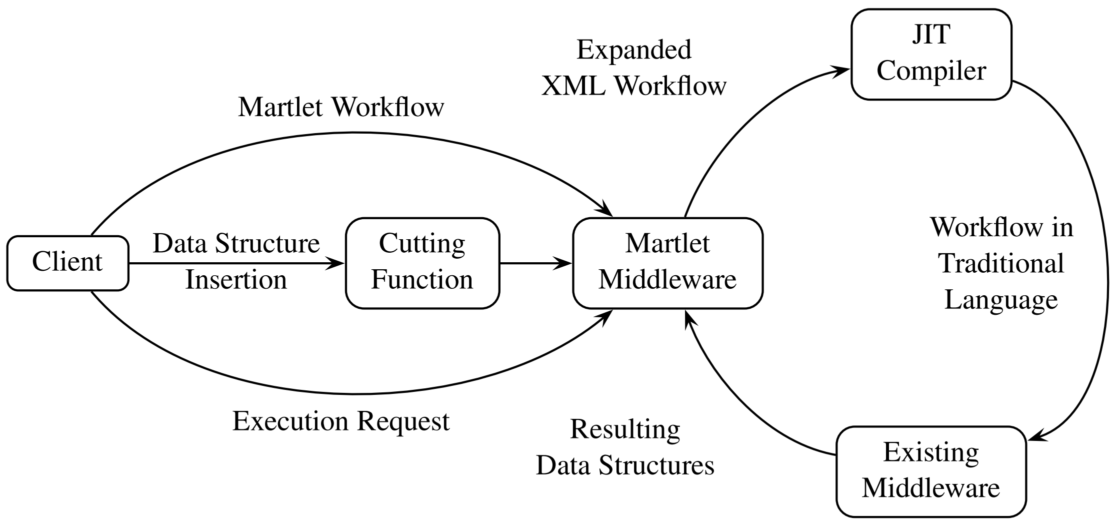


Figure 6.3: The outline execution cycle for a JIT compiler building on a Condor Pool or cluster.

large, but infrequent jobs. This is possible because Martlet makes it easier to make jobs arbitrarily parallel. This means where Condor traditionally uses lots of jobs to use lots of resources, and if there are not enough separate jobs, some resources went unused. Martlet allows one big job to be cut into many smaller jobs, so ensuring all resources are used.

Third, with e-Science projects trying to make clusters available as shared online resources, efficient scheduling of jobs is a problem, as the resource that the job will be sent to is not known, therefore it is not possible to design a job by hand such that it uses all available resources. This means that it is reasonable to assume that at any time, half the number of processors required for an average job will not be used on a resource. Martlet and the use of "cutting functions" creates a means by which the job can make the number of processors required variable. This allows the middleware to automatically change the number of processors needed to match the number that are free. This in turn makes it easier to schedule jobs on these resources such that all processors are in use, so improving performance.

An outline of the execution cycle built by extending the JIT execution cycle is shown in Figure 6.3. Note that this only adds one additional component to those already used in the JIT model, and does not interfere with the existing functionality of such resources, making its deployment very appealing.

6.2.3 Embedded Languages

The Martlet language can be spilt into two logical parts: The expandable constructs, which can be thought of as a coordination language for handling the abstraction of partitioned data; and the normal constructs, which are a coordination language for chaining together rough-grained functions into a single function that can then be run atomically.

This second language is similar to many other workflow languages, and while it is very powerful, it lacks the ability to interact with many of the resources that have caused the spawning of so many other workflow languages. To address this we could try to add all of the necessary functionality to interact with these resources, and try and keep track of the new features that are required, but ultimately, no matter how hard we tried, we would always be slightly behind the source languages.

Alternatively, we could provide the means to embed these existing languages into Martlet functions, allowing the user to take advantage of the other language's domain-specific content. While applicable to the stand-alone middleware, this would really come into its own if Martlet were running via a JIT compiler on top of a middleware that already supports the embedded language. This raises the possibility of different variants of Martlet that support different embedded languages, allowing the user to choose a variant that is both supported by the JIT covered middleware, and provides the appropriate domain-specific properties.

The idea of a range of different variants of Martlet with different domain-specific properties and levels of descriptiveness has another advantage. With CPDN, it was relatively easy to decide which functions we would support, as we could choose to just wrap the functionality of the Climate Data Analysis Toolkit [24], and we did not want too provide to much power to the user, as we want to make this resource open to the public. For many projects, it is not the case that they can so easily produce a set of suitable functions or that their users are not trusted. With the different variants of Martlet with different embedded languages, it would then be possible to make Martlet more descriptive for trusted users, while still having safer versions, such as the original, for less-trusted users. This would make it easier for resource administrators to decide in advance which domain specific jobs will be supported, and to grade the levels of access they provide to their resources.

6.2.4 Additional Expandable Constructs

As discussed in Section 6.1.3, Martlet is not able to perform a sort of a large dataset as it can only perform a parallel-prefix reduction, and does not have the means to redistribute local data structures over a distributed data structure. This restriction could be overcome by the addition of extra constructs enabling such functionality to be described. This then raises two questions:

1. What other constructs are required to describe other algorithms that are currently not possible?
2. If it is a good idea to include such constructs in a language such as Martlet?

The motivation for the second question stems from the level of network and compute resources required when executing the intermediate construct in a MapReduce, and the fact that users will be inclined to use it because of its power without thinking about the cost. Even in environments such as a Google data centre, they have problems with the availability of network bandwidth when executing the middle phase of a large MapReduce. As such adding additional constructs is not simply a case of harvesting them from elsewhere; detailed thought must be given as to their suitability for this domain.

6.3 Conclusion

In Section 6.1 we compared Martlet with other similar technologies, and showed that while there are other technologies starting to appear that are covering similar a similar problem domain, we are currently unaware of anything that addresses the same ground. In addition, we should draw comfort that we are heading in the right direction with this work, since a number of ideas which appear in Martlet that have now appeared in adjacent projects. This also adds to the speculation that these are all just parts of a bigger, clearer, and more intuitive model for parallel programming that will develop over the coming years in response to the more dynamic parallel environments, spanning from the hugely distributed Grids, to the multi-core processors for home computers.

In the previous chapter we showed that Martlet is a very powerful language that is capable of describing a wide range of algorithms. Though it is currently limited to our prototype middleware, as has been described in Section 6.2 Martlet can spread not only to build on existing languages extending their functionality with a minimal effort through JIT compilers, but can also spread out of its original target environment in into environments such as clusters and Condor Pools. These two expansions collectively represent a huge area of application for this work.

Chapter 7

Conclusion

In conclusion of this thesis, we will review both the contribution of this work, and an overview of how this work fits in with other work in the field of distributed utility computing, before making our final comments.

7.1 Contributions

The main contributions of this thesis are: a novel-programming model for loosely coupled, and dynamic computing systems; a high-level specification of a middleware to support this model; a prototype middleware; and a novel algorithm for calculating an SVD of a large dataset in a loosely coupled way.

7.1.1 Novel Programming Model

In Chapter 3 we introduced a novel programming model that addresses a very real issue in the construction of programs in distributed utility computing environments. This model allows users to write generic programs that can operate in a parallel, distributed computing environment without the user first having to know how their data is going to be partitioned across such an environment, and without having to handle the complexity that is raised when such partitionings change. This makes it possible for both inexperienced users and highly experienced users to achieve more in the irregular computing environment that is "Grid Computing". Many of the ideas used to achieve this are currently starting to appear in many other projects in different, but comparable fields. As such, this model is able to complement both existing models and appearing technologies with applications to both distributed utility computing, and smaller scale clusters and Condor pools.

The classes of algorithm that can be described by this programming model are best looked at as the union of two sets. It is able to use traditional data structures and constructs to describe all algorithms that can be described by traditional workflow languages. In addition to these algorithms, it is able to describe another class of algorithms based on the idea of a higher-level coordination language joining together pieces of more traditional workflow in a functional manner. In this class of algorithms, where all parts working on partitioned data can be described in terms of base and inductive operations, it is worth noting the similarity of this style to the techniques used in mathematics for constructing proofs in similar conditions. Collectively, these two sets describe a very broad and useful class of algorithm.

7.1.2 Middleware Specification

We have provided a high-level view of a minimal middleware required to implement this style of programming model in a distributed environment. From this, it is then possible to both evaluate the applicability of this model of programming to different environments, and to consider what extensions would be required to existing middleware to allow them to support this model of programming. Specifically, this has allowed us to show that through the use of JIT compilers, it would be possible to extend many existing middlewares with an additional layer so that they could support this new model of programming. This shows that this powerful extension to existing functionality can easily be integrated with existing work in such areas.

7.1.3 Prototype Middleware

The development of a prototype middleware has allowed us to demonstrate both the effectiveness of this programming model, and how it naturally fits a service based environment. It has also then gone on to provided a platform from which further research into this programming model can be based. The ease with which the middleware was extended to produce XML describing concrete instances of generic functions applied to specific partitioned data, supports our argument that existing middleware can be extended to support our model through the layering of a small amount of additional middleware on top of existing interfaces, so providing a huge range of applicability for the model implemented by Martlet.

7.1.4 Demonstration of Applications

Through the construction of the case studies, we have demonstrated both how many common problems can be described in this model by trivially adjusting existing arguments. We have then gone on to demonstrate how using this model we have produced new algorithms to solve problems that previously could only be solved on single processor or closely coupled systems, and how their construction can be made simple through the inductive nature of their construction in Martlet. This shows not only that this model is highly applicable to a wide range of problems, but also raises the potential of many more new and interesting algorithms waiting to be discovered when people start thinking about problems in this new way.

7.1.5 New Singular Value Decomposition Algorithm

The discovery of a new technique building on the Block-Lanczos method for generating the leading parts of an SVD in a loosely coupled way with a small memory footprint is a significant advance. This opens up a huge range of analysis possibilities for large datasets, ranging from drug discovery and genome work, to facial recognition and image processing, to climate analysis and weather forecasting, with many more applications likely to become apparent in the future.

7.2 Overview

The programming model and supporting infrastructure presented in this thesis falls between what Fox describes as "embarrassingly parallel" problems [129] addressed by projects such as BOINC [2] and Condor [77], and the closely coupled parallel systems based on message passing technologies such as MPI [113]. As has been shown by the uptake of projects that address the embarrassingly parallel problems, there is a huge demand for computing power from users who do not necessarily have either the time or the knowledge to develop the more intricate solutions afforded by closely coupled parallel systems. However, despite the uptake of these projects and the continuous improvements in their utility, the model has never moved far beyond being able to handle its initial class of algorithms. This class is not able to solve many of the problems faced by modern users, this has lead companies such as Google to develop alternative solutions to reduce the complexity of constructing more advanced algorithms for parallel environments. These solutions are fairly domain specific, addressing only the set of problems relevant to the company. Martlet represents one of the first steps towards an alternative model of programming to match an alternative style of computing presented through Grid Computing, and that allows users to write programs that extend beyond the embarrassingly parallel, without the complexity of having to handle message passing, changes to the data partitioning or the number of CPUs.

As this new model is constructed using abstract constructs that are transformed into traditional structures matching the specific conditions at runtime, the overhead of the algorithms constructed is negligibly larger than that of more traditional algorithms. As with all systems, some algorithms will take to this model better than others. Specifically with this model, algorithms that require access to specific pieces of information within a data structure, for example, the n^{th} column of a matrix are hard to describe with this model, and further research is required to overcome this. However, most algorithms that we have come across that require analysis on datasets of this size do not require such fine-grained access, and follow a more appropriate symmetrical pattern. While it would be possible to make further comments about the data size to processor requirements for our test cases, this would, in reality, be make more of a comment on the computing environment in question, not the programming model we are presenting here.

7.3 Final Comment

The work presented in this thesis solves a common but often unrecognised problem in many eResearch projects. However, this problem must be solved before eResearch can reach its full potential. While many middleware adapt jobs to the environment they are working in, none adapt the workflow. Instead they just use network capacity to overcome the problems raised by the dynamic distribution of data and resources in such environments. Ultimately, this is because, while people have been quick to uptake the new computing environments offered by Grid computing, they have constructed their programs using the same programming models as were used previously in more closely-coupled and stable environments. They have then made much progress by adjusting variables like the granularity of the statements in the programs, the means by which data is transmitted, and the way system information is used to optimise function calls. However, they remain bound by the constraints of models designed for different environments. Martlet is a first step into a new style of programming that we expect to see become more and more common in the near future, both as the project develops through the future work described in Section 6.2, and as the Grid environments and problems faced within them mature. This realisation can already be seen in the growing number of workshops, conferences, and drives appearing in an attempt to describe programming models for this new computing platform.

Bibliography

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, Christine Flood, Yossi Lev, and Cheryl McCosh. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc, September 2006.

[2] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID*, pages 4–10, 2004.

[3] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[4] Tony Andrews, Francisco Curbera, Hitesh Doholakia, Yaron Goland, Johannes Kiein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smitth, Satish Thatte, Ivana Trickovic, and Sanjiva Weerwarana. BPEL4WS. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, 2003.

[5] Apache Software Foundation. *Apache Axis*, 2005. URL: http://ws.apache.org/axis/.

[6] Apache Software Foundation. *The Apache Jakarta Project*, 2005. URL: http://jakarta.apache.org/tomcat/.

[7] Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach, John Manferdelli, Hiroshi Maruyama, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, John Shewchuk, and Dan Simon. Web Services Security (WS-Security). Technical report, W3C, 2002. URL: http://www-106.ibm.com/developerworks/webservices/library/ws-secure/.

[8] Malcolm Atkinson, David DeRoure, Alistair Dunlop, Geoffery Fox, Peter Henderson, Tony Hey, Norman Paton, Steven Newhouse, Savas Parastatidis, Anne Trefethen, and Paul Watson. Web service grids: An evolutionary approch. Technical report, Open Middleware Infrastructure Institute, 2004.

[9] Paul Avery and Ian Foster. The GriPhyN Project: Towards Petascale Virtual-Data Grids. Jan 2003.

[10] Zhou B. B. and Brent R. P. Parallel computation of the singular value decomposition on tree architectures. Technical report, Computer Science Laboratory, Australian National University, 1993.

[11] K. Belhajjame, S.M. Embury, H. Fan, C. Goble, H. Hermjakob, S.J. Hubbard, D. Jones, P. Jones, N. Martin, S. Oliver, C. Orengo, N.W. Paton, A. Poulovassilis, J. Siepen, R.D. Stevens, C. Taylor, N. Vinod, L. Zamboulis, and W. Zhu. Proteome data intergration: Charecteristics and challenges. In Simon J Cox and David W Walker, editors, *Proceedings of the UK e-Scienece All Hands Meeting 2005*, pages 418–425. EPSRC, EPSRC, September 2005.

[12] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, 1998.

[13] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[14] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[15] Michael Blow, Yaron Goland, Matthias Kloppmann, Frank Leymann, Gerhard Pfau, Dieter Roller, and Michael Rowley. BPELJ: BPEL for Java. Technical report, BEA and IBM, 2004.

[16] Jon Blower, Keith Haines, and Ed Llewellin. Data streaming, workflow and firewall-friendly Grid Services. In Simon J Cox and David W Walker, editors, *Proceedings of the UK e-Scienece All Hands Meeting 2005*, pages 858–865. EPSRC, EPSRC, September 2005.

[17] Jon Blower, Keith Haines, Adrityarajsingh Santokhee, and Roger Peppe. Composing workflows in the environmental sciences using Inferno. In *Proceedings of the UK e-Science All Hands Meeting*, pages 300–307. EPSRC, 2004.

[18] Jim Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task scheduling strategies for workflow-based applications in grids. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*. IEEE Press, 2005.

[19] Bruce M. Boghosian, Lucas I. Finn, and Peter V. Coveney. Moving the data to the computation: multi-site distributed parallel computation. Technical report, RealityGrid, January 2006.

[20] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugne Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. Web services addressing (ws-addressing). Technical report, W3C, 2004.

[21] Richard P. Brent and Franklin T. Luk. The solution of singular value problems using systolic arrays. *Real Time Signal Processing VII*, 495(495):7–12, 1984.

[22] Richard P. Brent and Franklin T. Luk. The solution of singular value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM Journal of Scientific Statistical Computing*, 6(1):69–84, 1985.

[23] Richard P. Brent, Franklin T. Luk, and Charles Van Loan. Computation of the singular value decomposition using mesh-connected processors. *Journal of VLSI and Computer Systems*, 1(3):242–270, 1985.

[24] British Atmospheric Data Centre. *Climate Data Analysis Tools*. URL: http://badc.nerc.ac.uk/help/software/cdat/.

[25] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[26] Doug Bunting, Martin Chapman, Oisin Hurley, Mark Little, Jeff Mischkinsky, Eric Newcomer, Jim Webber, and Keith Swenson. Web Services context service specification. Technical report, Arjuna Technologies Ltd, 2003.

[27] Stephen Buswell, Olga Caprotti, and Mike Dewar. Mathematical service description language. Technical report, The Monet Consortium, 2003. URL: http://monet.nag.co.uk/cocoon/monet/.

[28] Mark Carman, Luciano Serafini, and Paolo Traverso. Web service composition as planning. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services, June, Trento, Italy*, 2003.

[29] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. Technical report, University of New South Wales and Microsoft Research Ltd, Cambridge, November 2006.

[30] Steve J. Chapin. Disributed Scheduling Support in the Presence of Autonomy. Technical report, Department of Mathmatics and Computer Science, Kent State University, 1994.

[31] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11-12):1723–1743, 2003.

[32] Yannis Chicha, Manfred Riem, and David Roberts. The MONET Broker. Technical report, The MONET Consortium, 2004. URL: http://monet.nag.co.uk/cocoon/monet/.

[33] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC*, pages 246–254, 2002.

[34] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C, 1.1 edition, March 2001. URL: http://www.w3c.org/TR/wsdl.

[35] Jhon R. Corban. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls*. Springer-Verlag, 1991.

[36] Microsoft Corporations. *Microsoft .NET*, 2001. URL: http://www.microsoft.com/net.

[37] Peter Coveney, Jamie Vicary, Jonathan Chin, and Matt Harvey. Introducing WEDS: a WSRF-based enviroment for distributed simulation. Technical report, Centre for Computer Science, Department of Chemistry, University Collage London, 2004.

[38] Karl Czajkowski, Donald F Ferguson, Ian Foster, Jeffery Frey, Steve Graham, Igor Sedukhin, David Snelling, Steve Tuecke, and William Vambenepe. The WS resource framework. Technical report, Computer Associates International Inc and Fujitsu Limited, HP and IBM and The University of Chicago, 2004.

[39] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE '93: Parallel Architectures and Languages Europe*, pages 146–160, Berlin, DE, 1993. Springer-Verlag.

[40] J. Darlington, Y. K. Guo, H. W. To, and J. Yang. Functional skeletons for parallel coordination. In *EuroPar'95 — European Conference on Parallel Processing*, volume 966, pages 55–69, Stockholm, Sweden, August 29–31, 1995. Springer-Verlag.

[41] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Technical report, Google Inc, December 2004.

[42] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Pegasus: Planning for execution in grids. Technical report, Information Sciences Institute, 2002.

[43] EGEE. *EGEE: Enabling Grids for e-Science in Europe*. URL: http://egee-intranet.web.cern.ch/egee-intranet/gateway.html.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[45] Thomas Fahringer, Jun Qin, and Stefan Hainzer. Specification of grid workflow applications with agwl: an abstract grid workflow language. *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, 2:676–685 Vol. 2, 2005.

[46] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.

[47] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *IJSA*, 11(2):115–128, 1997.

[48] I. Foster and C. Kesselman. Globus: A toolkit-based grid architecture. In *The Grid: Blueprint for a Future Computing Infrastructure*, pages 259–278. MORGAN-KAUFMANN, 1998.

[49] I. Foster, R. Olson, and S. Tuecke. Productive Parallel Programming: The PCN Approach. *Journal of Scientific Programming*, 1(1):51–66, 1992.

[50] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceeding of the 14th International Conference on Scientific and Statistical Database Management*, 2002.

[51] Ian Foster and Carl Kesselman. The globus project: A status report. In *IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998. Describes the status of the Globus system as of early 1998.

[52] Ian Foster, Carl Kesselman, Jeffery M. Nick, and Steve Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. *Fourth Global Forum*, 2002.

[53] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable Virtual Organisations. *Lecture Notes in Computer Science*, 2150, 2001.

[54] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[55] Fabrizio Gagliardi, Bob Jones, Mario Reale, and Stephen Burke. European datagrid project: Experiences of deploying a large scale testbed for e-science applications. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, pages 480–500. Springer-Verlag, 2002.

[56] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[58] Gene H. Golub and Charles F Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3 edition, 1996.

[59] Daniel Goodman. Martlet; a scientific work-flow language for abstracted parallisation. In Simon J Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2006*. National e-Science Centre, National e-Science Centre, September 2006.

[60] Daniel Goodman. Introduction and Evaluation of Martlet, a Scientific Workflow Language for Abstracted Parallelisation. In *Proceedings of the 16th International World Wide Web Conference*. International World Wide Web Conference Committee, ACM, May 2007.

[61] Daniel Goodman and Andrew Martin. Scientific middleware for abstracted parallelisation. Technical Report RR-05-07, Oxford University Computing Laboratory, November 2005.

[62] Steve Graham, Karl Czajkowski, Donald F Ferguson, Ian Foster, Jeffrey Frey, Frank Leymann, Tom Maguire, Nataraj Nagaratnam, Martin Nally, Tony Storey, Steve Tuecke, William Vambenepe, and Sanjiva Weerawarana. Web service resource properties. Technical report, IBM and HP and The University of Chicago, 2003.

[63] Steve Graham, Tom Maguire, Jeffery Frey, Nataraj Nagaratnam, Igor Sedukin, David Snelling, Karl Czajkowski, Steve Tuecke, and William Vambenepe. Web Services service group - specification. Technical report, Computer Associates International Inc and Fujitsu Limited, HP and IBM and The University of Chicago, 2004.

[64] Steve Graham, Peter Niblett, Dave Chappell, Amy Lewis, Nataraj Nagaratnam, Jay Parikh, Sanjay Patil, Shivajee Samdarshi, Igor Sedukin, David Snelling, Steve Tuecke, William Vambenepe, and Bill Weihl. Publish-subscribe notification for web services. Technical report, IBM and Akamai Technologies and Computer Associates International and SAP AG and Fujitsu Laboratories of Europe and Globus and Hewlett-Packard and Sonic Software and TIBCO Software, 2004.

[65] GriPhyN. *Virtual Data Language*, 2006. http://www.griphyn.org/workspace/VDS/langref/.

[66] Roy Grønmo, David Skogan, Ida Solheim, and Jon Oldevik. Model-driven web services development. In *EEE*, pages 42–45, 2004.

[67] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*, pages 324–347. O'Reilly Media Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 3rd edition, September 2004.

[68] Raphael Hauser and Daniel Goodman. A Block-Lanczos Method for Computing the Leading Singular Values and Vectors of a Large Column-Distributed Matrix. Technical report, Oxford University Computing Laboratory, 2007.

[69] Pat Helland. Data on the Outside vs. Data on the Inside. Technical report, Microsoft Corporation, 2004.

[70] Peter Henderson and Jr. James H. Morris. A lazy evaluator. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 95–103, New York, NY, USA, 1976. ACM Press.

[71] C.A.R. Hoare, editor. *occam 2 Reference Manual*. Orentice Hall International Series in Computer Science. Prentice Hall International, 1988.

[72] Ryan Huebsch, Brent Chun, and Joseph M Hellerstein. PIER on PlanetLab: Initla experience and open problems. Technical report, Intel Research Berkeley, 2003.

[73] Geraint Jones and Michael Goldsmith. *Programming in occam 2*. Orentice Hall International Series in Computer Science. Prentice Hall International, 1988.

[74] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The power of parallel prefix. *IEEE Transactions on Computing*, C-34(10):965–968, October 1985.

[75] Bryan Lawrence, David Boyd, Kerstin Kleese van Dam, Roy Lowry, Dean Williams, Mike Fiorino, and Bob Drach. The NERC DataGrid. Technical report, CCLRC - Rutherford Appleton Laboratory, 2002.

[76] Bryan Lawrence, Ray Cramer, Marta Gutierrez, Kerstin Kleese van Dam, Siva Kondapalli, Susan Latham, Roy Lowry, Kevin O'Neill, and Andrew Woolf. The NERC Data-

Grid Prototype. Technical report, CCLRC e-Science Centre, British Atmospheric Data Centre and British Oceanographic Data Centre, 2003.

[77] Michael J. Litzkow. Remote UNIX, turning idle workstations into cycle servers. Technical report, University of Wisconsin Computer Sciences Department, 1988.

[78] Liiy Liu and Sam Meder. Web service base faults. Technical report, OASIS, 2006.

[79] Juliette Mainka, Yannis Chicha, and Marc Gaetano. Mathematical object manager release candidate. Technical report, University of Nice, 2004. URL: http://monet.nag.co.uk/cocoon/monet/.

[80] MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatable MPL) User Guide*, a3 edition, November 1992.

[81] Mathworks. *Matlab*. Mathworks, Inc., Natick, MA, 1999.

[82] Mono Project. *Mono*. URL: http://www.mono-project.com.

[83] Shyam Mudambi and Joachim Schimpf. Parallel CLP on Hetrogeneous Networks. Technical report, European Computer-Industry Research Centre, 1994.

[84] National Science Foundation. *US TeraGrid*. URL: http://www.teragrid.org.

[85] NGS. *National Grid Service*. URL: http://www.ngs.ac.uk/.

[86] OASIS. Uddi executive overview: Enabling service-oriented architecture. Technical report, OASIS, October 2004.

[87] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[88] Leonid Oliker, Rupak Biswas, Hongzhang Shan, and Warren Smith. Job scheduling in a heterogenous grid environment. Technical report, Lawrence Berkeley National Laboratory, 2004.

[89] OMG. *CORBA Componets*, June 2002.

[90] OMII. The OMII Product Roadmap. Technical report, OMII, 2004. URL: http://www.omii.ac.uk/roadmap.htm.

[91] Web Services Interoperabillity Organisation. *Web Services Interoperablity*, 2004. URL: http://www.ws-i.org.

[92] S. Parastatidis and J. Webber. The soap service description language. Technical Report 899, University of Newcastle upon Tyne, School of Computing Science, Apr 2005.

[93] S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield. An introduction to the soap service description language. Technical Report 898, University of Newcastle upon Tyne, School of Computing Science, Apr 2005.

[94] Savas Parastatidis, Paul Watson, and Jim Webber. Grid resource specification. Technical report, North East Regional e-Science Centre, 2003.

[95] Savas Parastatidis, Jim Webber, and Paul Watson. Using web services to build grid applications. the no risk wsgaf profile. Technical report, University of Newcastle, 2004.

[96] Savas Parastatidis, Jim Webber, Paul Watson, and Thomas Rischbeck. A grid application framework based on web services specifications and practices. Technical report, North East Regional e-Science Centre, 2003.

[97] Rob Pike, Dave Presotto, Sean Dorward, Dennis M. Ritchie, Howard Trickey, and Phil Winterbottom. The inferno operating system. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[98] Rob Pike and Dennis M. Ritchie. The styx architecture for distributed systems. *Bell Labs Technical Journal*, 4(2):146–152, April-June 1999.

[99] PlanetLab. *PlanetLab*. URL: http://www.planet-lab.org/.

[100] Radu Prodan and Thomas Fahringer. From Web Services to OGSA: Experiences in implementing an OGSA-based grid application. Technical report, University of Vienna and University of Innsbruck, 2003.

[101] M. Radenkovic and B. Wietrzyk. Life science grid middleware in a more dynamic environment. In *Proceedings of Grid Computing and its Application to Data Analysis*, pages 264–273. LNCS, 2005.

[102] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[103] M. A. Rappa. The utility business model and the future of computing services. *IBM Syst. J.*, 43(1):32–42, 2004.

[104] Daniel A Reed, Celso L Mendes, Chang da Lu, Ian Foster, and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure - Application Tuning and Adaptation.* Morgan Kaufman, San Francisco, CA, second edition, 2003. pp.513-532.

[105] Russ Rew, Glenn Davis, Steve Emmerson, and Harvey Davies. *NetCDF User's Guide*. UCAR/Unidata Program Center P.O. Box 3000 Boulder, Colorado, USA 80307, 2.4 edition, Feb 1996.

[106] Jeffery Ricker, Mayilraj Krishnan, and Keith Swenson. Asynchronous service access protocol. Technical report, OASIS, 2004.

[107] Arnold Robbins. *Unix in a Nutshell.* O'reilly and Associates, Inc, 101 Morris Street, Sebastopol, CA 95472, 3rd edition, August 1999.

[108] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, 1997.

[109] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In *ER*, pages 353–368, 2005.

[110] Scientific Computing Associates Inc, One Century Tower, 265 Church Street, New Haven, CT, USA. *Paradise User's Guild and Reference Manual*, 6.2 edition, April 2000.

[111] Scientific Computing Associates Inc, One Century Tower, 265 Church Street, New Haven, CT, USA. *Linda, User Guild*, September 2005.

[112] Scientific Computing Associates Inc, One Century Tower, 265 Church Street, New Haven, CT, USA. *NetWorkSpaces for Python, User Guide*, 1.0 edition, January 2006.

[113] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[114] Latha Srinivasan and Tim Banks. Web service resource lifetime. Technical report, OASIS, 2006.

[115] B. Srivastava and J. Koehler. Web service composition - current solutions and open problems, 2003.

[116] David Stainforth, Jamie Kettleborough, Andrew Martin, Andrew Simpson, Richard Gillis, Ali Akkas, Richard Gault, Mat Collins, David Gavaghan, and Myles Allen. Climate*prediction*.net: Design principles for public-resource modeling research. In *14th IASTED International Conference Parallel and Distributed Computing and Systems*, Nov 2002.

[117] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[118] Vijay Subramani, Rajkumar Kettimuthu, Srividya Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 359. IEEE Computer Society, 2002.

[119] Sun Microsystems, Inc. *Java Remote Method Invocation*. URL: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

[120] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. Visual grid workflow in triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[121] The MONET Consortium. Monet architecture overview. Technical report, The MONET Consortium, 2003. URL: http://monet.nag.co.uk/cocoon/monet/.

[122] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE*

conference on Supercomputing (CDROM), page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[123] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[124] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[125] W3C. *Simple Object Access Protocol (SOAP) 1.2*, 2003. URL: http://www.w3c.org/TR/SOAP.

[126] David C. H. Wallom and Anne E. Trefethen. Oxgrid, a campus grid for the university of oxford. In Simon J Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2006*. National e-Science Centre, National e-Science Centre, September 2006.

[127] Paul Watson and Chris Fowler. Dynasoar: An architecture for the dynamic deployment of web services on a grid or the internet. In Simon J Cox and David W Walker, editors, *Proceedings of the UK e-Scienece All Hands Meeting 2005*, pages 843–849. EPSRC, EPSRC, September 2005.

[128] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[129] G. Wilson. *Parallel Programming for Scientists and Engineers*. MIT Press, Cambridge, MA, 1995.

[130] Asim YarKhan, Keith Seymour, Kiran Sagi, Zhiao Shi, and Jack Dongarra. Recent Developments in Gridsolve. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(1):131–141, 2006.

[131] B. B. Zhou and R. P. Brent. On parallel implementation of the one-sided Jacobi algorithm for singular value decompositions. In *Proceedings Euromicro Workshop on Parallel and Distributed Processing*, pages 25–27, 1995.

[132] B. B. Zhou, R. P. Brent, and M. Kahn. A one-sided Jacobi algorithm for the symmetric eigenvalue problem. In *Proceedings Third Parallel Computing Workshop*, 1994.

[133] B. B. Zhou and Richard P. Brent. A parallel ring ordering algorithm for efficient one-sided Jacobi SVD computations. *Parallel and Distributed Computing*, 1997.

Appendix A

Martlet Syntax

This is a complete description of the grammar of Martlet expressed in Extended BNF.

```
Procedure → [Define] Proc

Define → “define” “{” (Identifier “=” Identifier “;”)+ “}”

Proc → “proc” Arguments ExpandableStatements

Arguments → “(” [ Identifier ( “,” Identifier )* ] “)”

TreeArgs → “(” TreeArg ( “,” TreeArg )* “)”

TreeArg →   “(” Identifier “,” Identifier “)\” Identifier “->” Identifier

ExpandanbleStatements →   ExpandableStatement
                         | “{“ ( ExpandableStatement )+ “}”

ExpandableStatement →   “if” “(” Test “)” ExpandableStatements
                        [ “else” ExpandableStatements ]
                       | “while” “(“ Test “)” ExpandableStatements
                       | “async” ExpandableStatements
                       | “seq” ExpandableStatements
                       | “foldl” Statements
                       | “foldr” Statements
                       | “map” Statements
                       | “tree” TreeArgs Statements
                       | Identifier “=” “new” Type “(” Identifier “)” “;”
                       | Identifier “=” “const” Type “(” Description “)” “;”
                       | (Identifier | URI) Arguments “;”
```

```
Statements →   Statement
               | "{" ( Statement )+ "}"

Statement →   "if" "(" Test ")" Statements
               [ "else" Statements ]
              | "while" "(" Test ")" Statements
              | "async" Statements
              | "seq" Statements
              | Identifier "=" "new" Type "(" Identifier ")" ";"
              | Identifier "=" "const" Type "(" Description ")" ";"
              | (Identifier | URI) Arguments ";"

Test → AndTest ( "||" AndTest )*

AndTest → NotTest ( "&&" NotTest )*

NotTest → ( "!" )* RelationalTest

RelationalTest →   Identifier ( "<" | ">" | "<=" | ">=" | "==" | "!=" ) Identifier
                  | '(" Test ")"

Identifier → (a-z|A-Z)+ (a-z|A-Z|0-9)*

URI → (a-z|A-Z|0-9|":"|"-"|"$"|..........)+

Description → (a-z|A-Z|0-9|":"|"-"|","|..........)+

Type → "Matrix" | "DisMatrix" | "Int" | "Double" | ...
```

Appendix B

Martlet XML

B.1 Expanded Martlet Schema

In this section, we present the schema for the XML documents produced at runtime to represent an expanded Martlet function. A graphical representation of this schema can be seen in Figure 6.2 on page 108.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="MartletSchema"
           targetNamespace="http://cpdn.net/MartletSchema.xsd"
           elementFormDefault="qualified"
           xmlns="http://cpdn.net/MartletSchema.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <!-- The root of the function -->
    <xs:element name="MartletFunction">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="Statement">
                    <xs:sequence/>
                    <xs:attribute name="uri"
                                  type="xs:string"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>

    <!-- Arguments passed to statements -->
    <xs:complexType name="arguments">
        <xs:sequence>
            <xs:element name="Argument"
                        minOccurs="0"
                        maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="value"
```

```
                                type="xs:string"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>

    <!-- Tests for while and ifElse statements -->
    <xs:complexType name="test">
        <xs:choice>
            <xs:element name="Argument"
                        minOccurs="2"
                        maxOccurs="2">
                <xs:complexType>
                    <xs:attribute name="value"
                                  type="xs:string"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="Test"
                        minOccurs="1"
                        maxOccurs="2"
                        type="test"/>
        </xs:choice>
        <xs:attribute name="type" type="xs:string"/>
    </xs:complexType>

    <!-- The type used to mimic polymorphism -->
    <xs:complexType name="Statement">
        <xs:choice>
            <xs:element name="Create" type="create"/>
            <xs:element name="Seq" type="seq"/>
            <xs:element name="Async" type="async"/>
            <xs:element name="While" type="while"/>
            <xs:element name="IfElse" type="ifElse"/>
            <xs:element name="Function" type="function"/>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="create">
        <xs:sequence>
            <xs:element name="Arguments" type="arguments"/>
            <xs:element name="Created"
                        minOccurs="1"
                        maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="name"
                                  type="xs:string"/>
                    <xs:attribute name="type"
                                  type="xs:string"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="Child" type="Statement"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="seq">
        <xs:sequence>
            <xs:element name="Arguments" type="arguments"/>
            <xs:element name="Child"
                        minOccurs="0"
                        maxOccurs="unbounded"
                        type="Statement"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="async">
        <xs:sequence>
            <xs:element name="Arguments" type="arguments"/>
            <xs:element name="Child"
                        minOccurs="0"
                        maxOccurs="unbounded"
                        type="Statement"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="while">
        <xs:sequence>
            <xs:element name="Arguments" type="arguments"/>
            <xs:element name="Test" type="test"/>
            <xs:element name="Child" type="Statement"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ifElse">
        <xs:sequence>
            <xs:element name="Arguments" type="arguments"/>
            <xs:element name="Test" type="test"/>
            <xs:element name="If" type="Statement"/>
            <xs:element name="Else"
                        minOccurs="0"
                        maxOccurs="1"
                        type="Statement"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="function">
        <xs:sequence>
            <xs:element name="Arguments" type="arguments"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>

</xs:schema>
```

B.2 Example XML Function

The expanded tree from Figure 3.11 on page 56 is converted to the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<MartletFunction xmlns="http://cpdn.net/MartletSchema.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://cpdn.net/MartletSchema.xsd
                                            MartletSchema.xsd"
  uri="DistributedMatrixAverage">
  <Create>
    <Arguments>
      <Argument value="MatrixSum"/>
      <Argument value="A1"/>
      <Argument value="MatrixCardinality"/>
      <Argument value="A2"/>
      <Argument value="A3"/>
      <Argument value="MatrixSumToVector"/>
      <Argument value="IntegerSum"/>
      <Argument value="B"/>
      <Argument value="MatrixDivide"/>
    </Arguments>
    <Created name ="C1" type="Matrix"/>
    <Created name ="C2" type="Matrix"/>
    <Created name ="C3" type="Matrix"/>
    <Created name ="D1" type="Integer"/>
    <Created name ="D2" type="Integer"/>
    <Created name ="D3" type="Integer"/>
    <Created name ="E" type="Integer"/>
    <Child>
      <Seq>
        <Arguments>
          <Argument value="MatrixSum"/>
          <Argument value="A1"/>
          <Argument value="C1"/>
          <Argument value="MatrixCardinality"/>
          <Argument value="D1"/>
          <Argument value="A2"/>
          <Argument value="C2"/>
          <Argument value="D2"/>
          <Argument value="A3"/>
          <Argument value="C3"/>
          <Argument value="D3"/>
          <Argument value="MatrixSumToVector"/>
          <Argument value="IntegerSum"/>
          <Argument value="B"/>
          <Argument value="E"/>
          <Argument value="MatrixDivide"/>
        </Arguments>
        <Child>
          <Async>
            <Arguments>
              <Argument value="MatrixSum"/>
              <Argument value="A1"/>
              <Argument value="C1"/>
              <Argument value="MatrixCardinality"/>
              <Argument value="A1"/>
              <Argument value="D1"/>
              <Argument value="A2"/>
              <Argument value="C2"/>
              <Argument value="A2"/>
              <Argument value="D2"/>
              <Argument value="A3"/>
              <Argument value="C3"/>
              <Argument value="A3"/>
              <Argument value="D3"/>
            </Arguments>
            <Child>
              <Seq>
                <Arguments>
                  <Argument value="MatrixSum"/>
                  <Argument value="A1"/>
                  <Argument value="C1"/>
                  <Argument value="MatrixCardinality"/>
                  <Argument value="A1"/>
                  <Argument value="D1"/>
                </Arguments>
                <Child>
                  <Function name ="MatrixSum">
                    <Arguments>
                      <Argument value="A1"/>
                      <Argument value="C1"/>
                    </Arguments>
                  </Function>
                </Child>
                <Child>
                  <Function name ="MatrixCardinality">
                    <Arguments>
                      <Argument value="A1"/>
                      <Argument value="D1"/>
                    </Arguments>
                  </Function>
                </Child>
              </Seq>
            </Child>
            <Child>
              <Seq>
                <Arguments>
                  <Argument value="MatrixSum"/>
                  <Argument value="A2"/>
                  <Argument value="C2"/>
                  <Argument value="MatrixCardinality"/>
                  <Argument value="A2"/>
                  <Argument value="D2"/>
                </Arguments>
                <Child>
                  <Function name ="MatrixSum">
                    <Arguments>
```

```
              <Argument value="A2"/>
              <Argument value="C2"/>
            </Arguments>
          </Function>
        </Child>
        <Child>
          <Function name ="MatrixCardinality">
            <Arguments>
              <Argument value="A2"/>
              <Argument value="D2"/>
            </Arguments>
          </Function>
        </Child>
      </Seq>
    </Child>
    <Child>
      <Seq>
        <Arguments>
          <Argument value="MatrixSum"/>
          <Argument value="A3"/>
          <Argument value="C3"/>
          <Argument value="MatrixCardinality"/>
          <Argument value="A3"/>
          <Argument value="D3"/>
        </Arguments>
        <Child>
          <Function name ="MatrixSum">
            <Arguments>
              <Argument value="A3"/>
              <Argument value="C3"/>
            </Arguments>
          </Function>
        </Child>
        <Child>
          <Function name ="MatrixCardinality">
            <Arguments>
              <Argument value="A3"/>
              <Argument value="D3"/>
            </Arguments>
          </Function>
        </Child>
      </Seq>
    </Child>
  </Async>
</Child>
<Child>
  <Create>
    <Arguments>
      <Argument value="MatrixSumToVector"/>
      <Argument value="C3"/>
      <Argument value="C2"/>
      <Argument value="IntegerSum"/>
      <Argument value="D3"/>
      <Argument value="D2"/>
      <Argument value="C1"/>
      <Argument value="B"/>
      <Argument value="D1"/>
      <Argument value="E"/>
    </Arguments>
    <Created name="E1" type="Integer"/>
    <Created name="B1" type="Matrix"/>
    <Child>
      <Seq>
        <Arguments>
          <Argument value="MatrixSumToVector"/>
          <Argument value="C3"/>
          <Argument value="C2"/>
          <Argument value="B1"/>
          <Argument value="IntegerSum"/>
          <Argument value="D3"/>
          <Argument value="D2"/>
          <Argument value="E1"/>
          <Argument value="C1"/>
          <Argument value="B"/>
          <Argument value="D1"/>
          <Argument value="E"/>
        </Arguments>
        <Child>
          <Seq>
            <Arguments>
              <Argument value="MatrixSumToVector"/>
              <Argument value="C3"/>
              <Argument value="C2"/>
              <Argument value="B1"/>
              <Argument value="IntegerSum"/>
              <Argument value="D3"/>
              <Argument value="D2"/>
              <Argument value="E1"/>
            </Arguments>
            <Child>
              <Function name ="MatrixSumToVector">
                <Arguments>
                  <Argument value="C3"/>
                  <Argument value="C2"/>
                  <Argument value="B1"/>
                </Arguments>
              </Function>
            </Child>
            <Child>
              <Function name ="IntegerSum">
                <Arguments>
                  <Argument value="D3"/>
                  <Argument value="D2"/>
                  <Argument value="E1"/>
                </Arguments>
              </Function>
            </Child>
          </Seq>
```

```
                </Child>
                <Child>
                  <Seq>
                    <Arguments>
                      <Argument value="MatrixSumToVector"/>
                      <Argument value="C1"/>
                      <Argument value="B1"/>
                      <Argument value="B"/>
                      <Argument value="IntegerSum"/>
                      <Argument value="D1"/>
                      <Argument value="E1"/>
                      <Argument value="E"/>
                    </Arguments>
                    <Child>
                      <Function name ="MatrixSumToVector">
                        <Arguments>
                          <Argument value="C1"/>
                          <Argument value="B1"/>
                          <Argument value="B"/>
                        </Arguments>
                      </Function>
                    </Child>
                    <Child>
                      <Function name ="IntegerSum">
                        <Arguments>
                          <Argument value="D1"/>
                          <Argument value="E1"/>
                          <Argument value="E"/>
                        </Arguments>
                      </Function>
                    </Child>
                  </Seq>
                </Child>
              </Seq>
            </Child>
          </Create>
        </Child>
        <Child>
          <Function name ="MatrixDivide">
            <Arguments>
              <Argument value="B"/>
              <Argument value="E"/>
              <Argument value="B"/>
            </Arguments>
          </Function>
        </Child>
      </Seq>
    </Child>
  </Create>
</MartletFunction>
```

Appendix C

Test Grid Results

This appendix lists the raw data gathered from a test grid of 10 machines used to profile Martlet and its middleware in Chapter 5. Servers 1 through 10 are data stores and data processors, and server 11 is the process coordinator. All values are listed in seconds.

C.1 Varying Dataset Size

In this section we list the experimental results for the ten machines performing the average function on 45,000 model runs including differing numbers of variables.

Server 1	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.81	3.8	3.84	4.62	8.06	60.23	787.46
Communication	0.09	0.09	0.1	0.09	0.07	0.12	0.39
Transport	0.01	0.03	0.02	0.02	0.01	0.05	0.32
Marshalling	0.01	0	0	0.02	0.01	0	0
Total	3.92	3.92	3.96	4.75	8.15	60.4	788.17

Server 2	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.84	3.74	3.88	4.59	8.56	60.27	783.37
Communication	0.04	0.05	0.05	0.06	0.06	0.12	0.31
Transport	0	0.01	0.02	0.01	0.03	0.06	0.26
Marshalling	0	0.01	0.02	0.02	0	0	0
Total	3.88	3.81	3.97	4.68	8.65	60.45	783.94

Server 3	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.83	3.77	3.92	4.58	8.64	59.12	786.15
Communication	0.06	0.09	0.08	0.06	0.1	0.12	0.58
Transport	0.01	0.02	0.02	0.01	0.03	0.05	0.55
Marshalling	0.01	0	0	0.01	0.01	0	0.01
Total	3.91	3.88	4.02	4.66	8.78	59.29	787.29

Server 4	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.85	3.78	3.91	4.59	8.84	58.9	785.34
Communication	0.32	0.32	0.31	0.28	0.33	0.37	0.57
Transport	0.02	0.01	0.02	0.01	0.03	0.05	0.27
Marshalling	0.06	0.02	0.02	0.01	0.05	0.03	0.02
Total	4.25	4.13	4.26	4.89	9.25	59.35	786.2

Server 5	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.8	3.81	3.92	4.43	8.18	58.43	789.2
Communication	0.44	0.39	0.4	0.37	0.44	0.51	0.93
Transport	0	0.01	0.03	0.02	0.02	0.08	0.54
Marshalling	0.02	0.02	0.02	0.03	0.03	0.02	0.01
Total	4.26	4.23	4.37	4.85	8.67	59.04	790.68

Server 6	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.76	3.77	3.92	4.72	8.61	58.71	787.82
Communication	0.06	0.02	0.04	0.07	0.06	0.11	0.49
Transport	0	0.01	0.01	0.03	0.01	0.07	0.42
Marshalling	0	0	0	0	0.01	0	0
Total	3.82	3.8	3.97	4.82	8.69	58.89	788.73

Server 7	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.87	3.85	3.93	4.57	8.2	60.17	786.75
Communication	0.07	0.04	0.07	0.07	0.08	0.15	0.58
Transport	0.01	0	0.01	0.02	0.03	0.09	0.53
Marshalling	0	0.01	0.01	0	0	0.02	0.03
Total	3.95	3.9	4.02	4.66	8.31	60.43	787.89

Server 8	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.77	3.83	3.95	4.56	8.42	59.61	782.96
Communication	0.07	0.08	0.08	0.06	0.09	0.15	0.6
Transport	0.02	0.03	0.04	0.02	0.03	0.07	0.54
Marshalling	0	0	0	0	0.01	0	0
Total	3.86	3.94	4.07	4.64	8.55	59.83	784.1

Server 9	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.85	3.81	3.85	4.52	8.57	59.85	785.69
Communication	0.03	0.06	0.06	0.08	0.08	0.08	0.47
Transport	0.01	0.02	0.01	0.03	0.02	0.05	0.42
Marshalling	0	0	0	0	0	0	0
Total	3.89	3.89	3.92	4.63	8.67	59.98	786.58

Server 10	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	3.82	3.79	3.91	4.59	8.34	60.5	783.59
Communication	0.15	0.18	0.15	0.2	0.2	0.27	0.78
Transport	0.04	0.02	0.03	0.03	0.05	0.11	0.67
Marshalling	0	0	0.01	0	0	0	0
Total	4.01	3.99	4.1	4.82	8.59	60.88	785.04

Server 11	Number of Entries in each Model						
	1	10	100	1000	10000	100000	1000000
Execution	0	0	0	0	0	0	0
Communication	0.97	0.97	1.02	0.98	0.98	0.98	0.95
Transport	0	0	0	0	0	0	0
Marshalling	0.06	0.06	0.04	0.05	0.05	0.04	0.05
Total	1.03	1.03	1.06	1.03	1.03	1.02	1

C.2 Varying Server Numbers

In this section we list the experimental results from varying the number of servers used to analysis 9000 model runs each containing 1 million entries.

Server 1	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	788.37	502.98	333.64	265.98	220.86	189.11	162.79	141.38	126.74
Communication	0.21	0.23	0.15	0.15	0.19	0.22	0.14	0.12	0.13
Transport	0.12	0.16	0.12	0.0 9	0.12	0.12	0.09	0.07	0.06
Marshalling	0.02	0.01	0	0	0.01	0.01	0.02	0	0.01
Total	788.6	503.22	333.79	266.13	221.06	189.34	162.95	141.5	126.88

Server 2	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	784.31	503.76	334.27	265.91	220.41	188.21	161.36	141.34	126.6
Communication	0.38	0.28	0.23	0.17	0.12	0.14	0.15	0.16	0.13
Transport	0.3	0.23	0.18	0.14	0.1	0.1	0.11	0.12	0.1
Marshalling	0.01	0.01	0	0.01	0	0.01	0	0	0.01
Total	784.26	503.66	334.16	265.77	220.24	188.07	161.23	141.23	126.49

Server 3	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	534.03	333.95	264.52	223.38	188.24	162.27	141.98	126.04
Communication	0	0.44	0.22	0.28	0.19	0.2	0.28	0.35	0.17
Transport	0	0.29	0.17	0.22	0.15	0.14	0.14	0.15	0.11
Marshalling	0	0.01	0.01	0.02	0.01	0	0.01	0.01	0
Total	0	534.48	334.18	264.82	223.58	188.44	162.56	142.34	126.21

Server 4	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	333.58	265.22	221.89	187.87	160.43	141.27	125.43
Communication	0	0	0.48	0.21	0.14	0.14	0.16	0.38	0.37
Transport	0	0	0.28	0.18	0.14	0.13	0.12	0.1	0.06
Marshalling	0	0	0.02	0	0	0	0.01	0.03	0.03
Total	0	0	334.08	265.43	222.03	188.01	160.6	141.68	125.83

Server 5	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	0	284.85	222.4	188.69	160.57	140.9	127.77
Communication	0	0	0	0.56	0.67	0.54	0.51	0.53	0.48
Transport	0	0	0	0.25	0.32	0.21	0.16	0.14	0.13
Marshalling	0	0	0	0.01	0.01	0.02	0.06	0.02	0.04
Total	0	0	0	285.42	223.08	189.25	161.14	141.45	128.29

Server 6	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	0	0	222.27	188.31	160.65	140.29	127.14
Communication	0	0	0	0	0.36	0.3	0.32	0.14	0.13
Transport	0	0	0	0	0.24	0.13	0.11	0.09	0.1
Marshalling	0	0	0	0	0.01	0	0.05	0	0
Total	0	0	0	0	222.64	188.61	161.02	140.43	127.27

Server 7	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	0	0	0	205.8	161.21	140.88	127.7
Communication	0	0	0	0	0	0.25	0.11	0.13	0.15
Transport	0	0	0	0	0	0.19	0.07	0.1	0.12
Marshalling	0	0	0	0	0	0.02	0	0	0.02
Total	0	0	0	0	0	206.07	161.32	141.01	127.87

Server 8	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	0	0	0	0	160.91	140.66	128.16
Communication	0	0	0	0	0	0	0.24	0.35	0.23
Transport	0	0	0	0	0	0	0.18	0.24	0.14
Marshalling	0	0	0	0	0	0	0.02	0.01	0.01
Total	0	0	0	0	0	0	161.17	141.02	128.4

Server 9	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	0	0	0	0	0	162.34	128.54
Communication	0	0	0	0	0	0	0	0.14	0.13
Transport	0	0	0	0	0	0	0	0.11	0.1
Marshalling	0	0	0	0	0	0	0	0.01	0.01
Total	0	0	0	0	0	0	0	162.49	128.68

Server 10	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	0	0	0	0	0	0	127.52
Communication	0	0	0	0	0	0	0	0	0.25
Transport	0	0	0	0	0	0	0	0	0.13
Marshalling	0	0	0	0	0	0	0	0	0.02
Total	0	0	0	0	0	0	0	0	127.79

Server 11	Number of Servers								
	2	3	4	5	6	7	8	9	10
Execution	0	0	0	0	0	0	0	0	0
Communication	0.3	0.38	0.73	0.56	0.69	1.07	1.16	1.34	0.97
Transport	0	0	0	0	0	0	0	0	0
Marshalling	0.05	0.05	0.02	0.02	0.05	0.03	0.08	0.04	0.04
Total	0.35	0.43	0.75	0.58	0.74	1.1	1.24	1.38	1.01

Appendix D

SVD Simulation

D.1 Simulation Results

This section contains the results of tests performed on a selection of SVDs generated to evaluate the accuracy of the algorithm presented in Section 5.3.3. We first generated the three constituent matrices of an SVD, then multiplied these together to produce a single matrix, so providing both input for our algorithm, and a correct result to compare the results against. The algorithm was implemented in Matlab to run on a single computer, and its coding can be seen in the next section.

Having run the algorithm on our test data, we then used three tests to evaluate the accuracy of the results. These take the actual vector v_a from our original matrix V, and the vector returned from our algorithm v_r, using these, the tests can be expressed as;

1. the angle between v_a and v_r

$$\left| \left(\arccos \frac{v_a \cdot v_r}{||v_a||_2 ||v_r||_2} \right) \bmod \pi \right|$$

2. the relative lengths of v_a and v_r under A

$$\left| \frac{||Av_a||_2}{||Av_r||_2} - 1 \right|$$

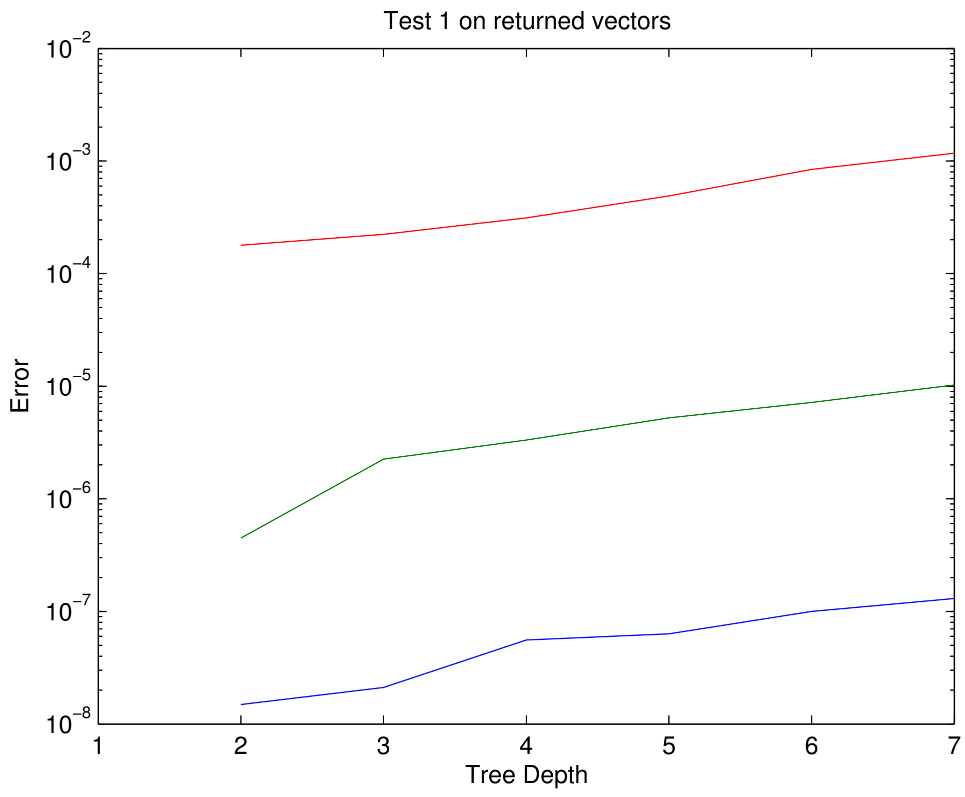
3. the angle between v_a and v_r under A

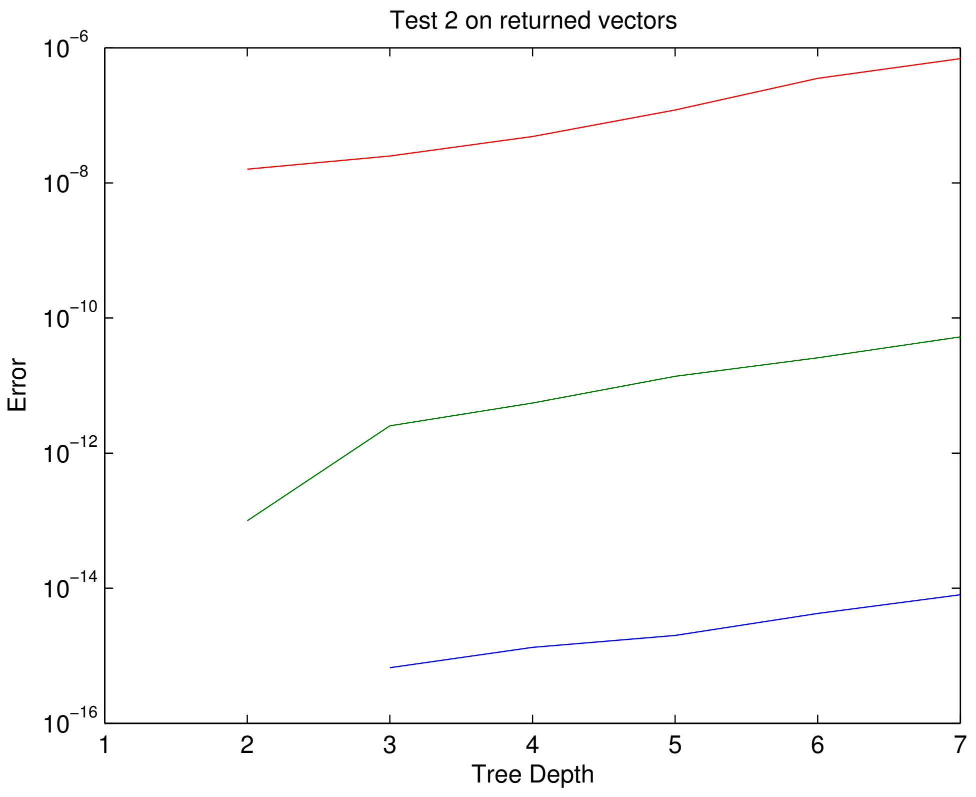
$$\left| \left(\arccos \frac{Av_a \cdot Av_r}{||Av_a||_2 ||Av_r||_2} \right) \bmod \pi \right|$$

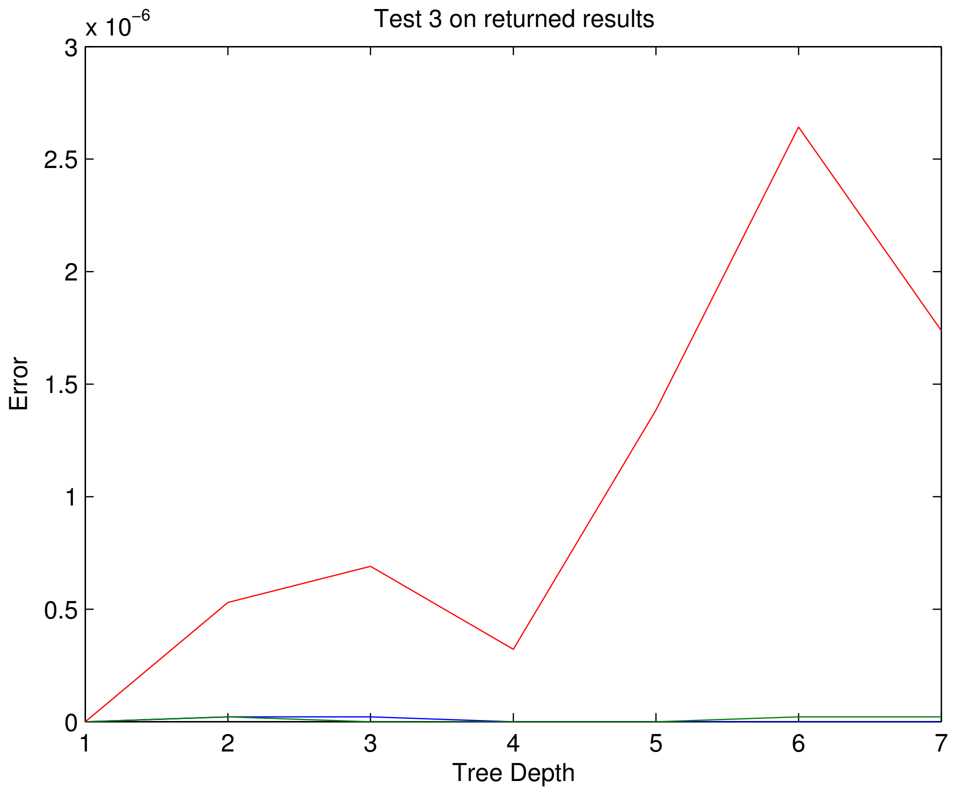
Using these tests we then examined various properties of our algorithm.

D.1.1 Effect of Tree Depth on Accuracy

These graphs show the effect on the accuracy of the results caused by increasing the tree depth on a moderately sized matrix. The red is the least significant vector, moving through to the blue representing the most significant vector. Note the relative insignificance of the error due to the depth of the tree compared to the increase of error due to the position of the vector being returned.

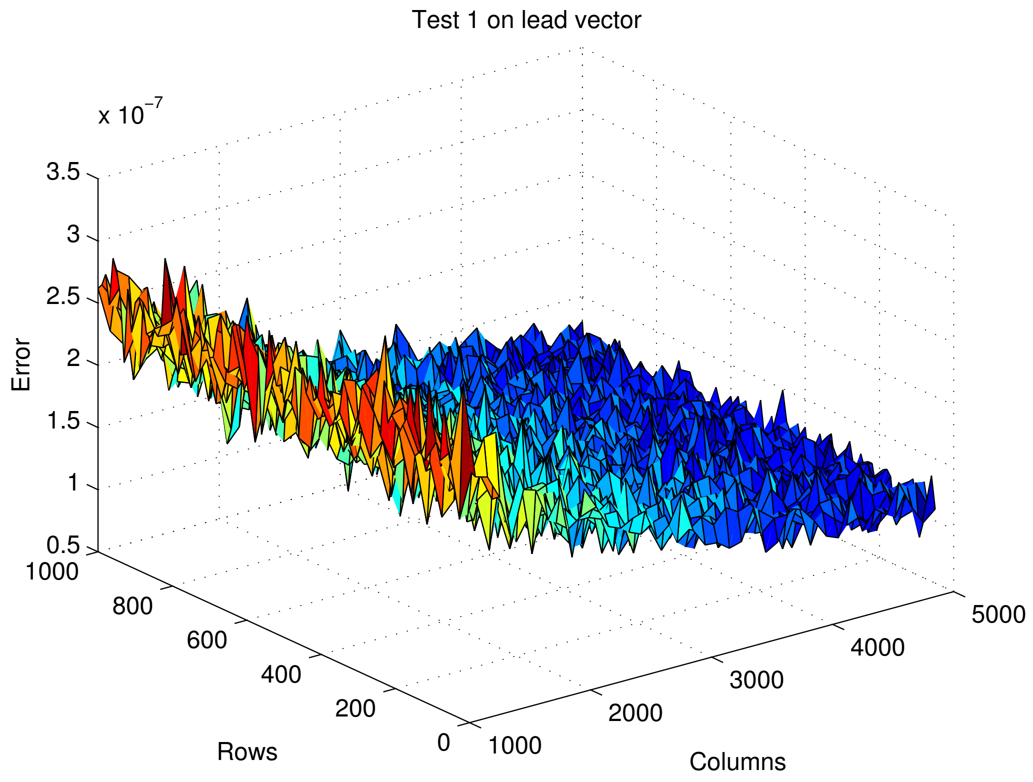


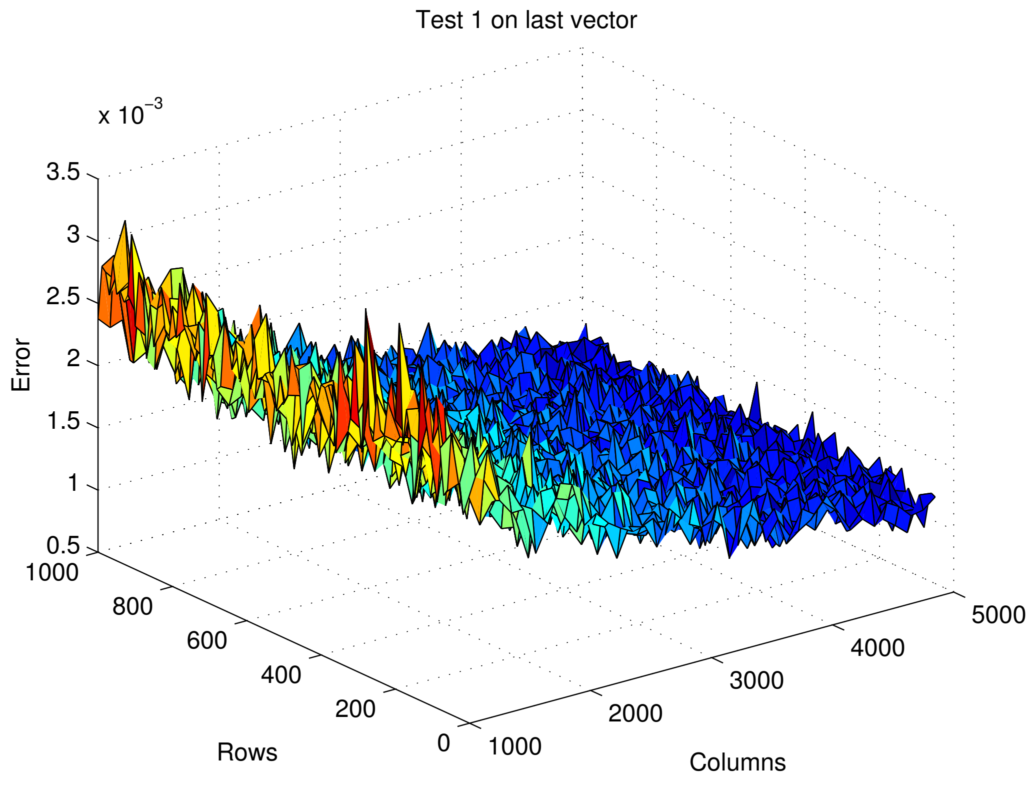

Test 2 on returned vectors
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
Error
1
2
3
4
5
6
7
Tree Depth

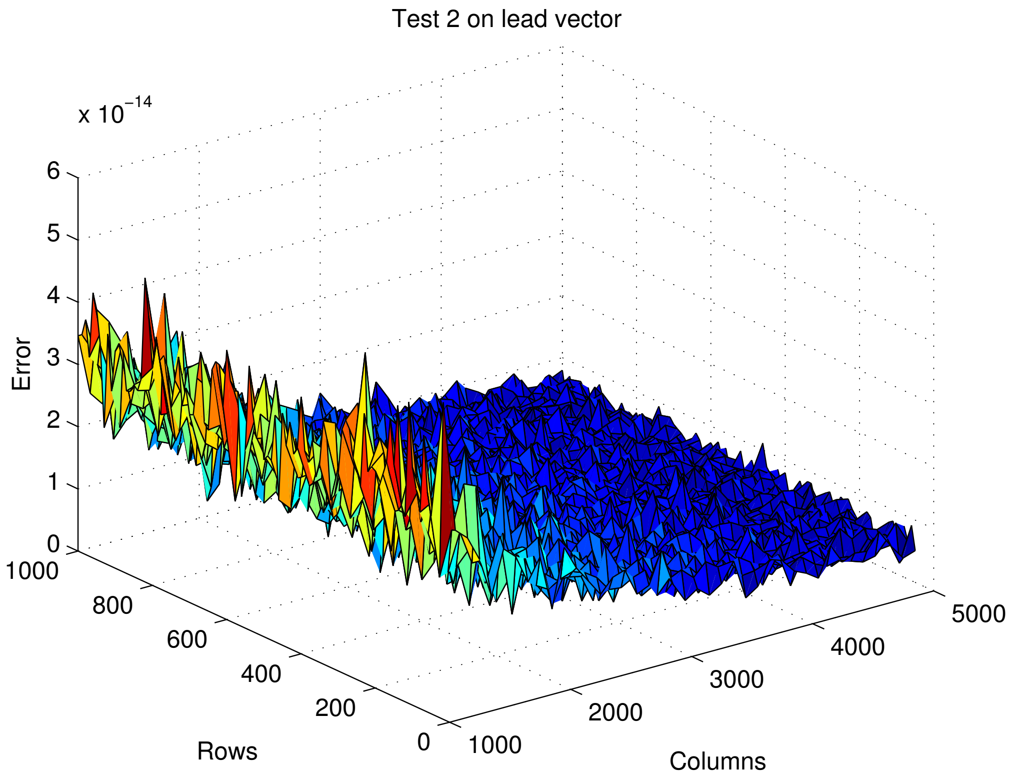

Test 3 on returned results
x 10^{-6}
3
2.5
2
1.5
1
0.5
0
Error
1
2
3
4
5
6
7
Tree Depth

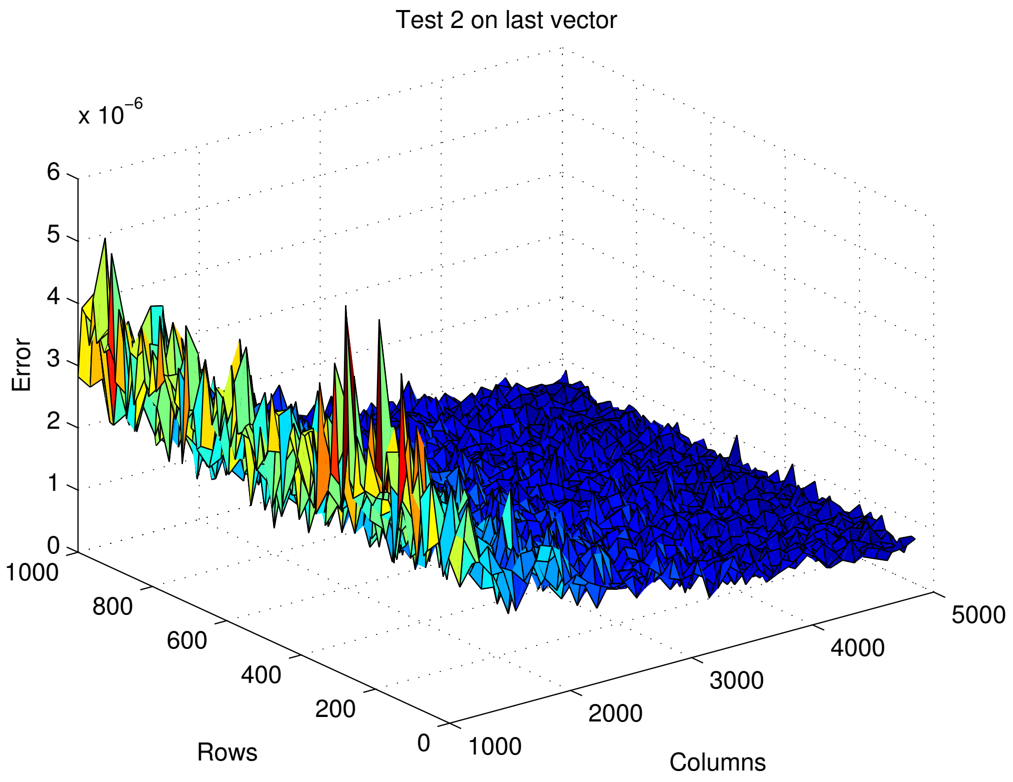
D.1.2 Effect of Matrix Size on Accuracy

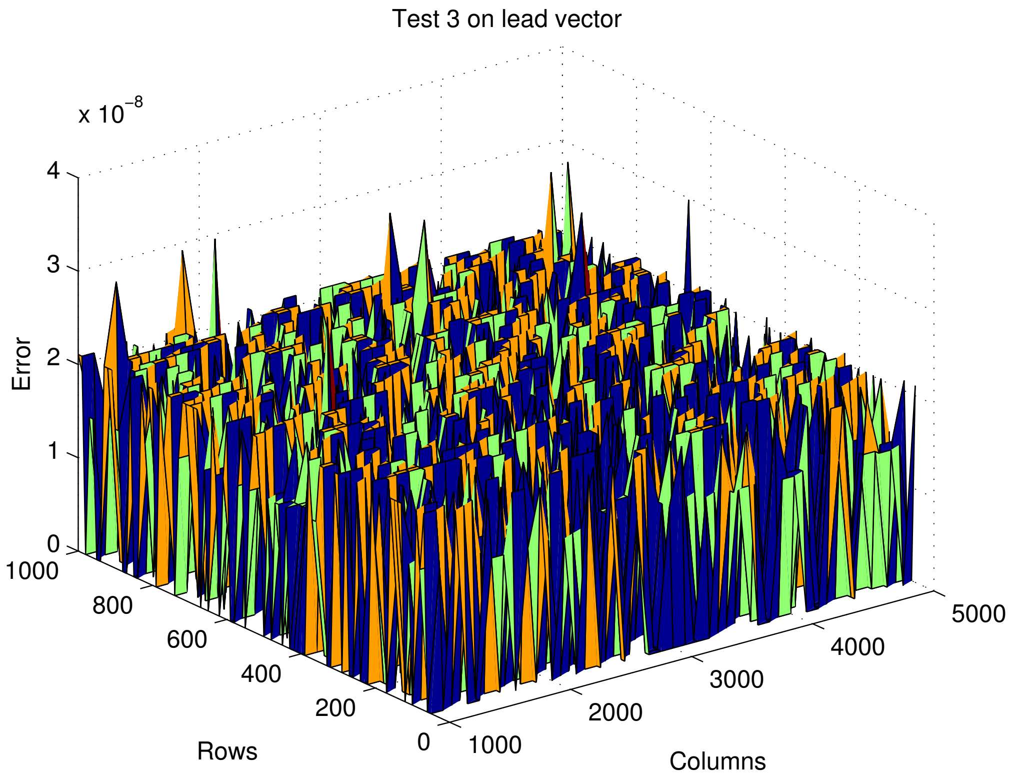
These graphs show the accuracy for matrices of different sizes. It is expected that the surface would be smooth if each point was the average error for a large number of tests on that size matrix. Note the increase in accuracy as the matrix size increases.

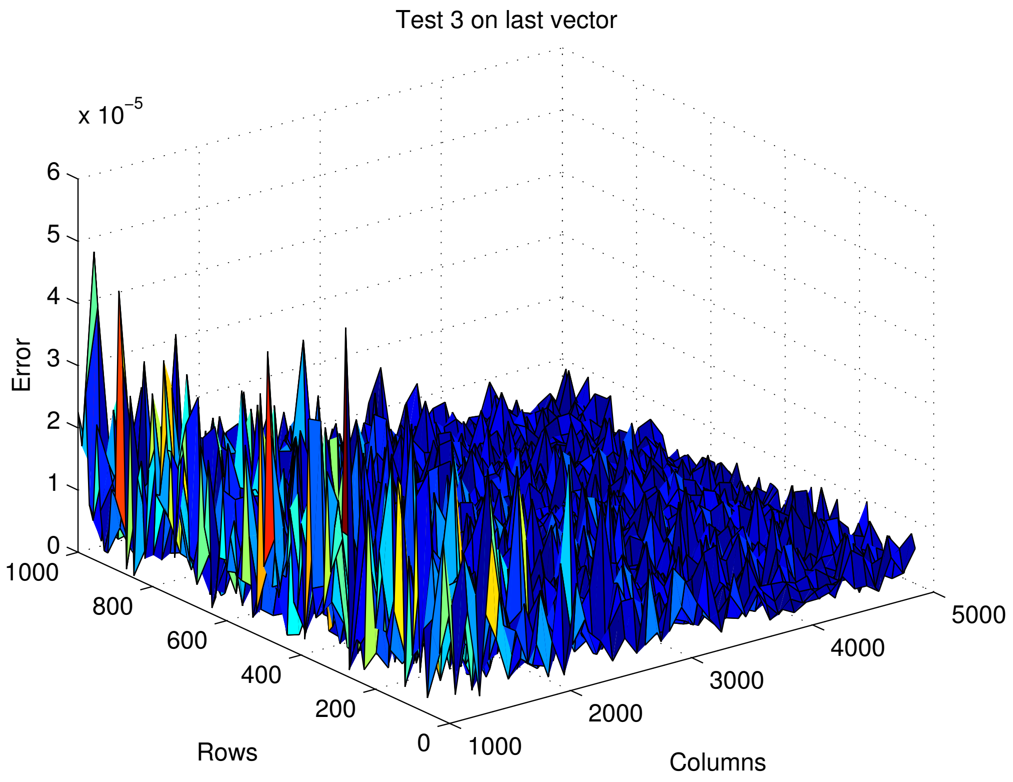



Test 1 on last vector
x 10^-3
3.5
3
2.5
2
1.5
1
0.5
Error
1000
800
600
400
200
0
Rows
1000
2000
3000
4000
5000
Columns

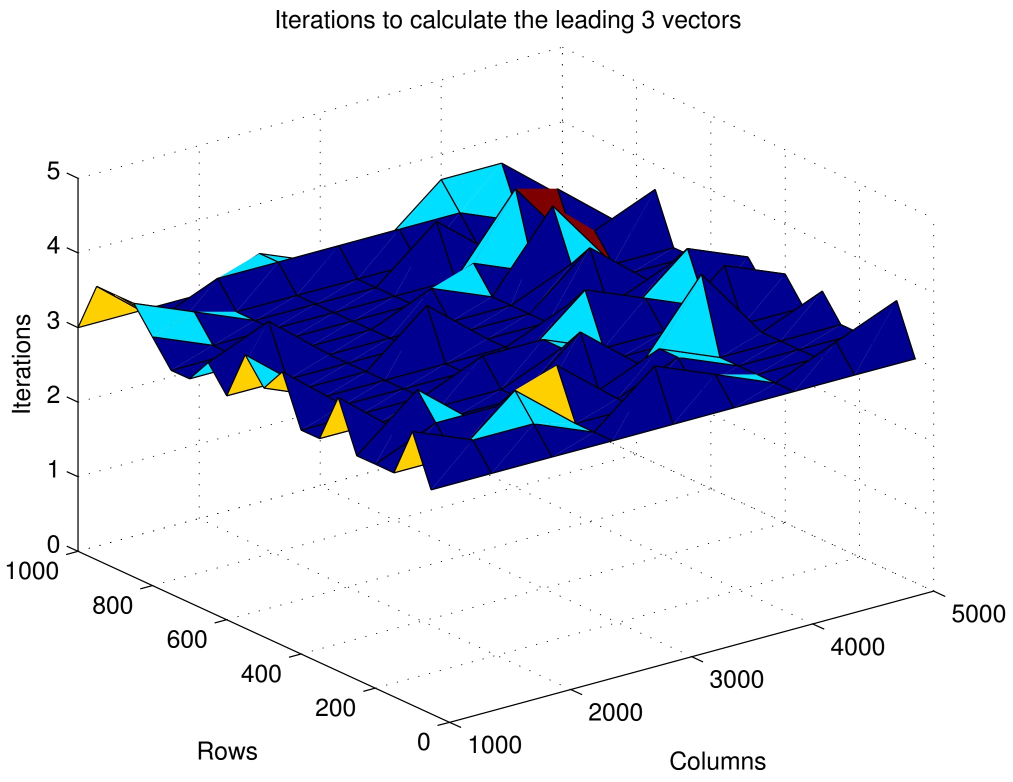

Test 2 on lead vector
x 10^-14
6
5
4
3
2
1
0
Error
1000
800
600
400
200
0
Rows
1000
2000
3000
4000
5000
Columns

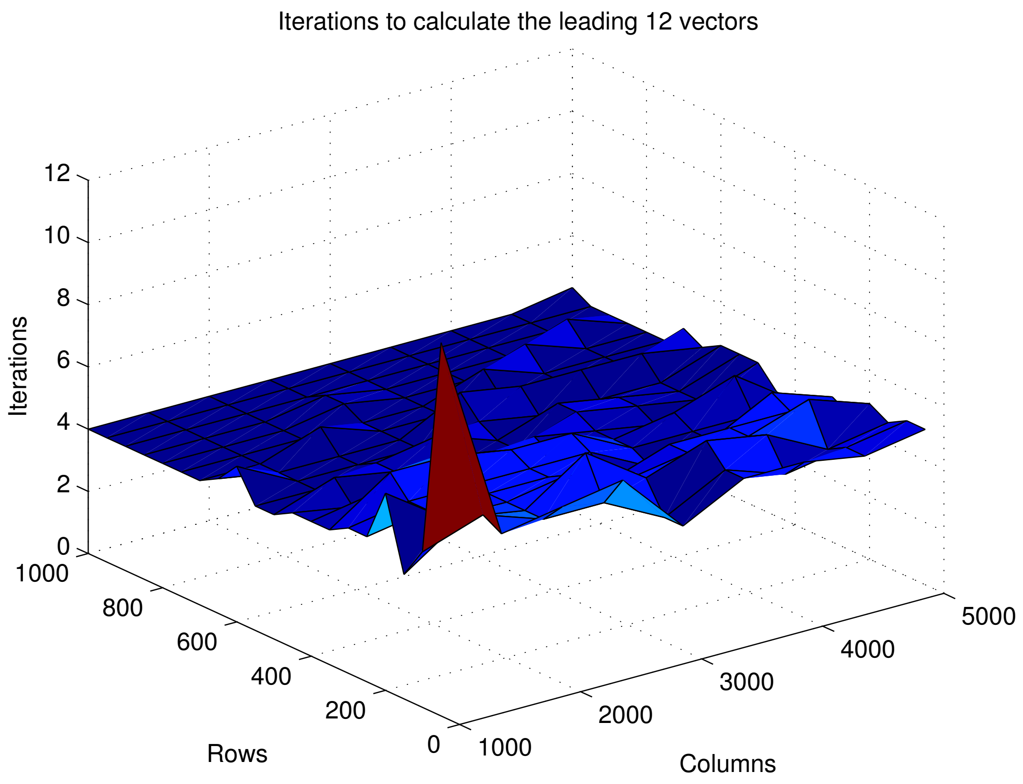

Test 2 on last vector
x 10^−6
6
5
4
3
2
1
0
Error
1000
800
600
400
200
0
Rows
1000
2000
3000
4000
5000
Columns


Test 3 on lead vector
x 10^−8
4
3
2
1
0
Error
1000
800
600
400
200
0
Rows
1000
2000
3000
4000
5000
Columns


Test 3 on last vector
x 10^-5
6
5
4
3
2
1
0
Error
1000
800
600
400
200
0
Rows
1000
2000
3000
4000
5000
Columns

D.1.3 Effect of Matrix Size on Iterations to Convergence





These graphs show the number of iterations of the core function required before convergence is achieved for different size matrices. Note that with the exception of small matrices, the number of iterations is relatively small and constant.

D.1.4 Effect of Results set Size on Accuracy

These tables list the error on the nine leading vectors for varying values of p. Note the rapid decrease in error as you move away from the least significant vector

Test 1

		Number of Vectors								
		1	2	3	4	5	6	7	8	9
Vector	1	6.52e-4	7.63e-6	8.43e-8	1.49e-8	0	3.65e-8	1.49e-8	0	2.11e-8
	2		8.59e-4	6.95e-6	7.89e-8	2.58e-8	3.65e-8	2.11e-8	2.11e-8	0
	3			7.38e-4	7.69e-6	9.66e-8	1.49e-8	2.98e-8	2.11e-8	2.11e-8
	4				8.55e-4	7.82e-6	6.99e-8	2.11e-8	2.58e-8	2.98e-8
	5					6.75e-4	7.75e-6	9.19e-8	2.11e-8	0
	6						6.46e-4	6.70e-6	7.3e-8	1.49e-8
	7							7.25e-4	9.58e-6	8.03e-8
	8								7.94e-4	7.66e-6
	9									8.85e-4

Test 2

		Number of Vectors								
		1	2	3	4	5	6	7	8	9
Vector	1	2.13e-7	2.91e-11	2.87e-15	1.11e-16	2.22e-16	1.11e-16	3.33e-16	5.55e-16	4.44e-16
	2		3.69e-7	2.41e-11	2.89e-15	1.11e-16	2.22e-16	0	4.44e-16	0
	3			2.72e-7	2.96e-11	4.44e-15	0	2.22e-16	0	2.22e-16
	4				3.65e-7	3.05e-11	2.22e-15	2.22e-16	2.22e-16	1.79e-15
	5					2.28e-7	3.01e-11	9.55e-15	3.33e-16	4.44e-16
	6						2.09e-7	2.25e-11	1.34e-14	1.67e-14
	7							2.63e-7	4.57e-11	5.61e-13
	8								3.15e-7	1.11e-11
	9									3.91e-7

Test 3

		Number of Vectors								
		1	2	3	4	5	6	7	8	9
Vector	1	3.51e-7	0	1.49e-8	1.49e-8	0	0	0	0	0
	2		2.35e-6	2.11e-8	0	2.11e-8	0	0	2.11e-8	0
	3			2.02e-6	1.49e-8	1.49e-8	0	1.49e-8	0	1.49e-8
	4				3.94e-7	0	0	0	0	1.49e-8
	5					2.26e-6	2.11e-8	0	0	2.11e-8
	6						3.83e-6	0	2.11e-8	2.11e-8
	7							1.36e-6	2.11e-8	0
	8								2.53e-6	0
	9									1.89e-6

D.2 Matlab Code

This section contains a listing of the Matlab code used to implement and test the local and distributed versions of our variation on the Block-Lanczos method. The main method for the local simulation is `control.m` and for the distributed simulation is `treeControl.m`.

D.2.1 Local Algorithm

control.m

```
m = 10; %Set Global values
n = 20;
p = 3;
err = 0.0000000001;

[A,Q] = init(m,n,p); %initialise values

[S, Q] = localCalculation(A, Q, p, err); %Perform calculation

V=Q(:,1:p)';
S=S(1:p,1:p);
U=A*V'*inv(S);
```

init.m

```
function [A,Q] = init(m,n,p) %A function to initialise values.

if n < m
    [U,R] = qr(rand(m,m));
    U = U(:,1:n);
    [V,R] = qr(rand(n,n));
    S = eye(n);

    i=0;
    j=1000000000000;
    while i < n
        i = i + 1;
        S(i,i) = j;
        j = j/10;
    end
else
    [U,R] = qr(rand(m,m));
    [V,R] = qr(rand(n,n));
    V = V(1:m,:);
    S = eye(m);

    i=0;
    j=1000000000000;
    while i < m
        i = i + 1;
        S(i,i) = j;
        j = j/10;
    end
end

A = U*S*V;
Q = A(1:2*p, :)';
```

localCalculation.m

```
function [S, Q] = localCalculation(A, Q, p, err)

Qp = Q;
[S, Q] = iteration_SVD(A, Q, p);

while (norm(Q(:,1:p)-Qp(:,1:p))) > err
    Qp = Q;
    [S, Q] = iteration_SVD(A, Q, p);
end
Q = Q(:,1:p);
```

iteration_SVD.m

```
function [S, Q] = iteration_SVD(A, Q, p)
    [U,S,V] = svd(A*Q);
    X = Q * V(:,1:p);
    [Q, R] = qr(A'*U(:,1:p)); %Calculate the new value of Q
    [Q, R] = qr([X, Q(:,1:p)]);
    Q=Q(:,1:2*p);
```

D.2.2 Distributed Extention

This is a listing of the functions created to extend the local algorithm into a distributed algorithm using a tree structure.

treeControl.m

```
m = 50; %Set values
n = 400;
p = 5;
err = 0.0000000001;

[A,Q] = init(m,n,p); %initialise matrix

[R, S, V] = tree(A, Q, p, err, 5); %construct tree and execute

V=V(:,1:p)';
S=S(1:p,1:p);
U=R(:,1:p)*inv(S);
```

tree.m

```
function [R, S, V] = tree(A, Q, p, err, level)
if level == 1
    [S, V] = localCalculation(A, Q, p, err);
    R = A*V;
else %not fault tolerant, A must be able to split 2\^level times
    [m, n] = size(A);
    d = floor(n/2);
    A1 = A(:,1:d);
    Q1 = Q(1:d,:);
    A2 = A(:,d+1:n);
    Q2 = Q(d+1:n,:);

    [R1, S1, V1] = tree(A1, Q1, p, err, level-1);
    [R2, S2, V2] = tree(A2, Q2, p, err, level-1);

    V1 = [V1', zeros(p,n-d)]';
    V2 = [zeros(p,d), V2']';
    VT = [V1,V2];
    A = [R1,R2];
    Q = A(1:2*p, :)';

    [S, V] = localCalculation(A, Q, p, err);
    R = A*V;
    V = VT*V;
end
```