

Introduction and Evaluation of Martlet, a Scientific Workflow Language for Abstracted Parallelisation

Daniel Goodman
Oxford University Computing Laboratory
Parks Road, Oxford,
OX1 3QD, UK
Daniel.Goodman@comlab.ox.ac.uk

ABSTRACT

The workflow language Martlet described in this paper implements a new programming model that allows users to write parallel programs and analyse distributed data without having to be aware of the details of the parallelisation. Martlet abstracts the parallelisation of the computation and the splitting of the data through the inclusion of constructs inspired by functional programming. These allow programs to be written as an abstract description that can be adjusted automatically at runtime to match the data set and available resources. Using this model it is possible to write programs to perform complex calculations across a distributed data set such as Singular Value Decomposition or Least Squares problems, as well as creating an intuitive way of working with distributed systems.

Having described and evaluated Martlet against other functional languages for parallel computation, this paper goes on to look at how Martlet might develop. In doing so it covers both possible additions to the language itself, and the use of JIT compilers to increase the range of platforms it is capable of running on.

Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classification—*Concurrent, distributed, and parallel languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Abstract data types, Concurrent programming structures, Control structures, Procedures, functions and subroutines*

General Terms

Algorithms, Design, Languages

Keywords

Martlet, workflow, e-Science, abstraction, parallel computing, distributed computing, scientific computing

1. INTRODUCTION

The workflow language Martlet [11] described in this paper implements a new programming model that allows users

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

to write parallel programs and analyse distributed data without having to be aware of the details of the parallelisation. It abstracts the parallelisation of the computation and the splitting of the data through the inclusion of constructs inspired by functional programming. These allow programs to be written as an abstract description that can be adjusted to match the data set and available resources automatically at runtime. While this programming model adds some restrictions to the way programs can be written, it is possible to perform complex calculations across a distributed data set such as Singular Value Decomposition or Least Squares problems, and it creates an intuitive way of working with distributed systems. This allows inexperienced users to take advantage of the power of distributed computing resources, and reduces the workload on experienced distributed programmers.

Existing workflow languages such as BPEL[2], Pegasus [9] and Taverna/Scufl [13] allow the chaining together of computational functions to provide additional functions. They have a variety of supporting tools and are compatible with a wide range of different middlewares, databases and scientific equipment. However, they all implement the same imperative programming model where a known number of data inputs are mapped to computational resources and executed with the standard imperative constructs, taking advantage of the potential for parallelisation where possible. As they only take a known number of inputs, none of them are able to describe a generic workflow in which the number of inputs is unknown, which the middleware can then adapt to perform the described function at runtime once the final number of inputs is known.

While applicable to a wide range of projects, including the Oxford e-Research Centre's Campus Grid [18], Martlet was originally created in response to some of the problems faced in the distributed analysis of data generated by the ClimatePrediction.net¹[6, 15] project. ClimatePrediction.net is a distributed computing project inspired by the success of the SETI@home²[1] project. Users download a model of the Earth's climate and run it for approximately fifty model years with a range of perturbed control parameters before returning results read from their model to one of the many upload servers.

The output of these models creates a data set that is distributed across many servers in a well-defined fashion. This data set is too big to transport to a single location for anal-

¹<http://www.climateprediction.net>

²<http://setiathome.ssl.berkeley.edu>

ysis, so it must be worked on in a distributed manner if a user wants to analyse more than a small subset of the data.

In order to derive results, it is intended that users will submit analysis functions to the servers holding the data set. As this data set provides a resource for many people, it would be unwise to allow users to submit arbitrary source code to be executed. In addition, users are unable to ascertain how many servers are spanned by a given subset of this data that they wish to analyse, and nor should they care. Their interest is in the information they can derive from the data, not how it is stored. These requirements mean a trusted workflow language is required as an intermediate step, allowing the construction of analysis functions from existing components, and abstracting the distribution of the data from the user.

Section 2 describes in more detail the style of problem Martlet is designed to address, and why existing languages are not sufficient, before going on to describe Martlet in Section 3 and, Section 4. In Section 5 we look briefly at the middleware constructed as a proof of concept for this model. We then compare Martlet to other technologies in Section 6, and look at how this work might progress in Section 7, before concluding in Section 8.

2. EXAMPLE PROBLEM

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

$$y_0 = \sum_{i=0}^{n_1-1} x_i$$

$$z_0 = n_1$$

$$y_1 = \sum_{i=n_1}^{n_2-1} x_i$$

$$z_1 = n_2 - n_1$$

$$\vdots$$

$$\vdots$$

$$y_{a-1} = \sum_{i=n_{a-1}}^{n_a-1} x_i$$

$$z_{a-1} = n_a - n_{a-1}$$

$$\bar{x} = \frac{\sum_{i=0}^{a-1} y_i}{\sum_{i=0}^{a-1} z_i}$$

where $x_0 \dots x_{n-1}$ are the individual runs partitioned across the a servers. y_0 through y_{a-1} then store the sum of the runs on each server, and z_0 through z_{a-1} stores how many runs each sum represents. So each subset of the data set has a computation performed on it, with the results then being used by a final computation to produce the overall average. Each of these computations could occur on a different computing resource.

To write this in an existing workflow language in such a way that it is properly executed in parallel, the user must first find out how many servers their required subset of data spans. 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 that the user himself must write the code to deal with 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 greatly increasing the 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, so it is not possible to just produce a library for most languages. This problem is removed with Martlet, by making such abstractions a fundamental part of the language.

3. INTRODUCING MARTLET

Our workflow language *Martlet* supports most of the common constructs of the existing workflow languages. In addition to these, it also has constructs inspired by inductive constructs of functional programming languages [5]. These are used to implement a new 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 concrete 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 extending an existing one 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.

The inspiration for this programming model came from functional programming languages where it is possible to write extremely concise powerful functions based on recursion. The reverse of a list of elements for instance can be defined in Haskell [5] as:

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

This simply states that if the list is empty, the function will return an empty list, otherwise it will take the first element from the list and turn it into a singleton list. Then it will recursively call reverse on the rest of the list and concatenate the two lists back together. The importance of this example is the explicit separation between the base case and the inductive case. Using these ideas it has been possible to construct a programming model and language that abstracts the level of parallelisation of the data away from the user, leaving the user to define algorithms in terms of a base case and an inductive case.

Along with the use of functional programming constructs, two classes of data structure, *local* and *distributed*, were created. Local data structures are stored in a single piece; distributed data structures are stored in an unknown number of pieces spanning an unknown number of machines. Distributed data structures can be considered as a list of references to local data structures. These two classes of data structure allow the functional constructs to take a set of distributed and local data structures, and functions that perform operations on local data structures. These are then used as a base case and an inductive case to construct a workflow where the base function gets applied to all the local data structures referenced in the distributed data structures, before the inductive function is used to reduce these partial results to a single result. So, for example, the distributed average problem looked at in Section 2, taking the distributed matrix **A** and returning the average in a column vector **B**, could be written in Martlet as the function in Figure 1.

```

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

proc(A,B)
{
    // Declare the required temporary 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 1: Function for computing the average of a matrix A split across an unknown number of servers. The syntax and semantics of this function are explained in Section 4.

Due to this language being developed for large scale distributed computing on huge data sets, 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.

4. SYNTAX AND SEMANTICS

To allow the global referencing of data and functions, both are assigned URIs. 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 are used for variables in the procedure, so allowing the URIs for data to only be entered 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 parts of the URI. 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.

The description of the process itself starts with the keyword “**proc**”, followed by a list of arguments that are passed to the procedure. There must be at least one argument due to the stateless nature of processes. While additional syntax describing the read-write nature of the arguments could improve readability, it is not included, as it would also prevent certain patterns of use. This may change in future variants of the language as discussed in Section 7. 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.

There are two types of statement: normal statements and expandable statements. The difference between the two 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 call makes it adjust its shape to suit the set of concrete data references it has been passed. Normal statements only propagate the *expand* call through to any children they have, whereas expandable statements adjust the structure of the tree to match the specific data set it is required to operate on.

4.1 Normal Statements

As the language currently stands, there are six different types of normal statement. These are sequential composition, asynchronous composition, if-else, while, temporary variable creation, and process calls. Their syntax is:

Sequential Composition is marked by the keyword **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 that this construct really is required is when one wants to create a function in which a set of sequential lists of statements were run concurrently by an asynchronous composition. An example of this is shown in Figure 2.

Asynchronous Composition is marked by the keyword **async** and encompasses a set of statements. When this is

```
async{
  seq{
    function1(A,B,C);
    function2(A,B);
    function3(B,C);
  }

  seq{
    function4(D,E);
    function1(D,E,F);
    function5(E,F);
  }
}
```

Figure 2: **seq** used to run two sequential sets of operations asynchronously.

executed each statement in the set is started concurrently. The asynchronous statement only terminates when all the sub-statements have returned.

In order to prevent race conditions it is necessary that no process uses a variable concurrently with a process that writes to the variable. This is enforced by the middleware at runtime.

if-else & while are represented and behave the same as they would in any other procedural language. There is a test and then a list of statements.

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 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 will 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 distributed data structure of the right type is going to appear at this stage in the procedure, as was the case in the average calculation example in Figure 1.

Process calls fall into one of two categories. They can either be statically named in the function with a URI, or are passed in as a reference at runtime. Both appear as an identifier and a list of arguments.

4.2 Expandable Statements

There are four expandable statements, **map**, **foldr**, **foldl** and **tree**. Each of these has a functional programming equivalent. Expandable statements do not propagate the call to expand to their children and must have been expanded before the function can be computed. This means

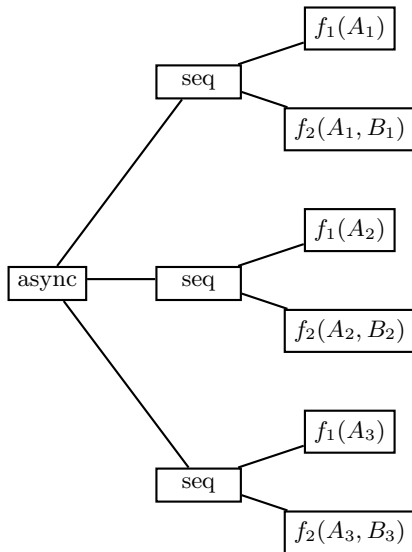


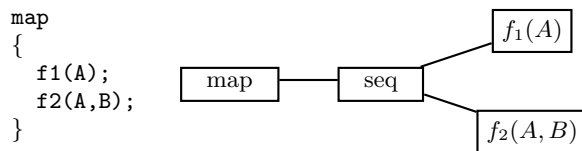
Figure 3: The abstract syntax tree for the example **map** statement after **expand** has been called setting $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$.

that on any given path between the root and a leaf, there must be at most one expandable statement.

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

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

Map in Martlet encompasses a list of statements as shown in the example below. Here function calls **f1** and **f2** are implicitly joined in a sequential composition to create the function represented by **f** in the Haskell definition. The list is created by distributed values **A** and **B**. While in its unexpanded abstract form, this example maps onto the abstract syntax tree also shown below.



When this is expanded, it looks at the distributed data structures it has been passed and creates a copy of these statements to run independently on each piece of the distributed data structure as shown in Figure 3.

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** node can be written to.

foldr is a way of applying a function and an accumulator value to each element of a list. This is defined in Haskell as:

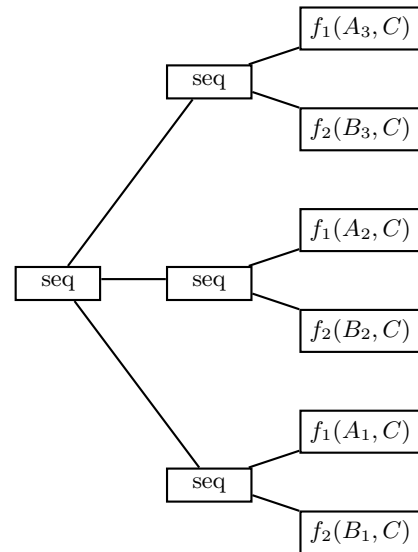
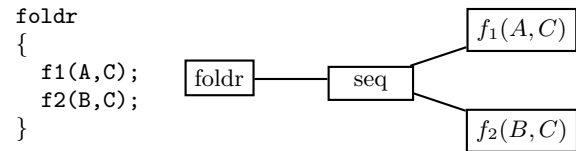


Figure 4: The abstract syntax tree for the example **foldr** statement after **expand** has been called setting $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$.

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

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.



When this function 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 4.

foldl is the mirror image of **foldr** so the Haskell example would now evaluate to $((((0+1)+2)+3)+4)+5$

The syntax tree in Martlet is expanded in almost exactly the same way as **foldr**. The only difference is the function calls from the sequential statement are in reverse order, see Figure 5. The only time that there is any reason to choose between **foldl** and **foldr** is when **f** is not commutative.

tree is a more complex statement type. It constructs a binary tree with a part of the distributed data structure at each leaf, and the function **f** at each node. When executed, unlike the folds, this is able to take advantage of the potential for parallel computation. A Haskell equivalent is:

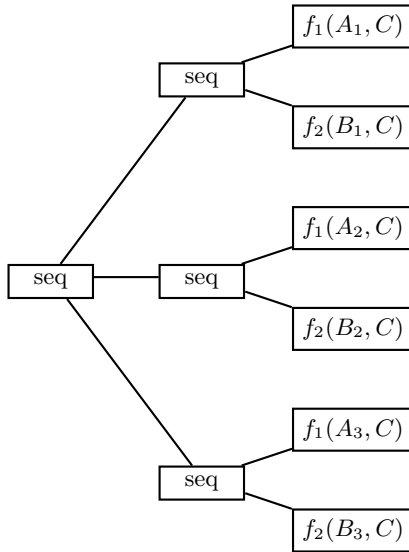


Figure 5: The abstract syntax tree after a foldl with the same structure as the foldr example has been expanded with concrete arguments $A = [A_1, A_2, A_3]$ and $B = [B_1, B_2, B_3]$. Note the reversal of the ordering of the distributed arguments compared with foldr

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

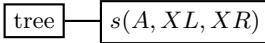
`split` is not defined here since the shape of the tree is not part of the specification. It will however always split the list so that neither is empty.

Unlike the other expandable statements, each node in a tree takes $2n$ inputs from n distributed data structures, and produces n outputs. As there is insufficient information in the structure to construct the mappings of values between nodes within the tree, the syntax requires the arguments that the statements use to be declared in brackets above the function so as to provide this additional information.

Non-distributed constants and processes used in f are simply denoted as a variable name. The relationship between distributed inputs and the outputs of f are encoded as $(XLeft, XRight) \backslash X \rightarrow A$, 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 and can be used as an input from X at the next level in the tree.

Consider a function that uses a method `sum` passed into the statement as s , a distributed argument X as input and outputs the result to the non-distributed argument A . This could 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 6. Because of the use of asynchronous statements any value that is written to must be passed in as either an input or an output.

4.3 Example

If the Martlet function to calculate averages from the example in Figure 1 where submitted it would produce the abstract syntax tree shown in Figure 7. This can then be expanded using the techniques show here to produce concrete functions for different concrete datasets, so allowing the user to generate averages from many differently partitioned datasets with just this one workflow.

5. MIDDLEWARE

To allow the testing, evaluation and *use* of this language and programming model, a supporting middleware has been constructed [12] using web services supported by Apache Axis [3] and Jakarta Tomcat [4]. This supporting platform was chosen in order to leave open the option of migrating to the Open Middleware Infrastructure Institutes [14] platform if desirable in the future.

The middleware consists of three logical elements, *Data Stores*, *Data Processors* and *Process Coordinators*. These elements can be grouped together at will, creating a structure in many ways similar to that used by the MONET [16] project. An example of a possible grouping is shown in Figure 8.

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 a data store.

Process Coordinators are the only component that users interact with. They handle access to the rest of the middleware. This is where the generic trees that represent submitted functions are expanded to fit the arguments on which the function has been called before they are broken up and scheduled across the data processors. Process Coordinators are the only component to have any knowledge of other nodes in the system.

These three elements use SOAP [17] to send control signals and small data structures, while an out of band system, currently SFTP, is used to transfer large data structures. In addition each server publishes information about its configuration and available operations for other parts of the middleware to read.

6. EVALUATION

As we have found no projects with a similar approach aimed at a similar style of environment, a direct comparison with other projects has not been possible. As such, in this section we will look at how Martlet compares with other functional coordination languages, namely Functional Skeletons [7] and Map-Reduce [8]. These are two pieces of work using functional constructs to abstract the complexity of distributed systems.

6.1 Functional Skeletons

Functional skeletons are used for programming clusters, and parallel machines, where as the name suggests they

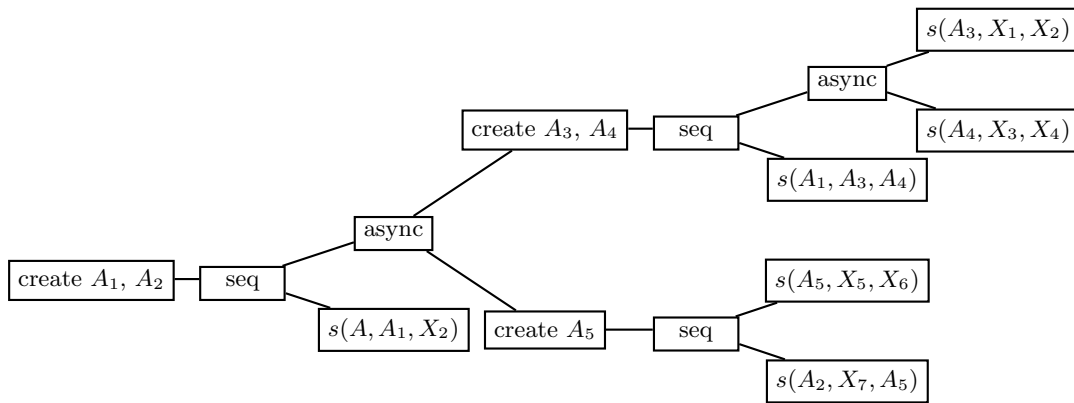


Figure 6: When the tree function from the end of Section 4.2 is expanded with $X = [X_1, X_2, X_3, X_4, X_5, X_6, X_7]$, this is one of the possible trees that could be generated.

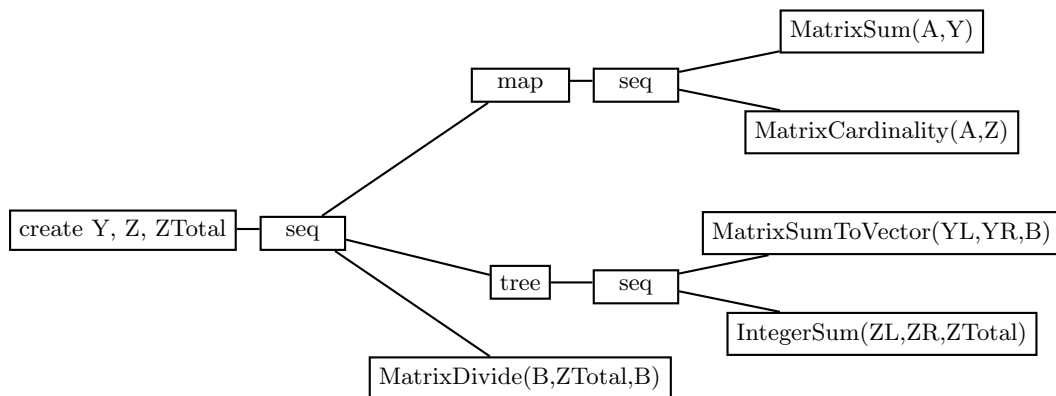


Figure 7: The abstract syntax tree representing the generic work-flow to compute the average introduced in Figure 1.

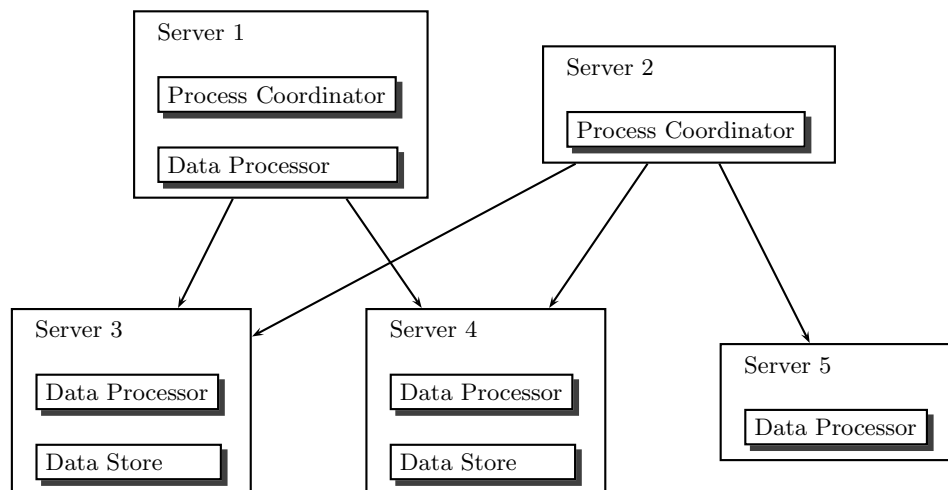


Figure 8: An example of how five servers could be configured. Note that more than one Process Coordinator can use each Data Store and Data Processor and the Process Coordinator does not need to know about all available servers.

provide a framework on which programs can then be constructed. The idea is that when a program is ported to a different architecture, the corresponding skeletons 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 here 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 user input to construct. This makes skeletons, while interesting and a potentially valuable source of inspiration, not directly applicable to our target problem domain.

6.2 Map-Reduce

Independently developed at the same time as Martlet, Map-Reduce 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. Google's implementation of this model is an api that works with the Google File System [10] allowing parallel calculations on data, while abstracting the complexity of the data storage and processing.

While highly successful, this 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++, and again the model requires that the user supply information about the architecture that they wish to execute over. It also 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.

However it is interesting to compare the algorithms that can be run using Map-Reduce and the algorithms that can be run using Martlet. In Google's paper on Map-Reduce [8], three algorithms are discussed, a distributed grep, a distributed word count, and a distributed sort. Of these Martlet is able to describe both the grep, and the word count. However, 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 Map-Reduce, as such it is not possible to implement sorting in Martlet. This is due to the fact that in a sort the answer size is equal to the input size, and the parallel prefix style reduction performed by Martlet's tree construct limits the final output to the size that can fit on a single machine.

One of the analysis operations which is desired by ClimatePrediction.net and possible with Martlet is a Singular Value Decomposition returning the leading p vectors. This cannot be done efficiently using the Map-Reduce model as there are a fixed number of steps, one map and one reduce in any individual Map-Reduce. As such, while it is possible to chain together many map-reduces, it is not possible to perform a parallel prefix style reduction, so preventing such algorithms from being described.

7. FUTURE WORK

In this section we look to the future, and examine both ways in which Martlet could be applicable to existing middleware, and how Martlet itself might develop to better fit potential use cases.

7.1 Building on other workflow engines

At runtime, when concrete values have been provided, the abstract functions are converted into concrete functions for execution. These concrete functions then contain only operations that are supported by a range of other languages. As such, one possible development would be to provide a means by which these could be executed on existing middleware. This could be achieved by cutting back the 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 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, instead of scheduling tasks across many machines, just produce an XML document describing the concrete task. The JIT compilers would then perform an XML transformation to produce the language of choice, which could then be submitted. This would allow compilers for a range of languages to be easily produced, allowing this layer to be placed on top of a wide range of existing resources with minimal effort, extending 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.

This also provides the possibility to use Martlet in a new scenario. 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, and the function is written in Martlet, then like in Map-Reduce, a job can be split into lots of smaller jobs. This provides two benefits. First it provides another means for big jobs to be made parallel in environments such as clusters and Condor Pools. Second, and probably more importantly, with eScience projects trying to make clusters and Condor Pools online resources, it creates a means by which the function can automatically have the number of processors it needs adjusted. This then makes scheduling of jobs on these resources such that all processors are in use easier, so improving performance.

7.2 Possible additions and variations

The difficulty Martlet has performing sorts on very large data sets raises the question of how Martlet might develop to overcome these and other restrictions. The three main points for possible development are:

- The addition new of constructs to allow classes of algorithm which are currently unavailable. However many constructs, such as the step between the Map and the

Reduce in Map-Reduce may, while required for sorting, be inappropriate for the highly distributed environments this project is aimed at, as they encourage people to write network intensive code.

- The addition of extra constraints on function construction in order to make them easier to read and less likely to contain errors, for example type information on function headers. While the addition of a type system would make functions easier to read and help prevent functions being partially evaluated before failing, it would also need to be able to describe generics and type hierarchies if it were not to limit the general nature of the language. Other possible additions such as flags to show if a parameter is an input, a result, or both, would, while improving the readability, restrict the classes of algorithm available. More thought and experience about how Martlet is going to be used is needed before decisions on such additions can be made.
- The ability to embed other languages into Martlet scripts in addition to making function calls through predefined functions would extend the ability to express domain specific information. This would remove the need to decide in advance which domain specific jobs will be supported, and will ease the integration of Martlet constructs into other languages. 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. It does bring with it issues of relating to the amount of power given to the user, and therefore raises the possibility of different variants of Martlet.

8. CONCLUSIONS

In this paper we have introduced a language, Martlet and programming model that uses functional constructs and, two classes of data structure (local and distributed). Using these constructs and structures, Martlet is able to abstract from users the complexity of creating parallel processes over distributed data and computing resources. This allows users simply to think about the functions they want to perform and does not require them to worry about the implementation details.

While we have not been able to perform a direct comparison with other projects, this work is currently being tested with data from the ClimatePrediction.net project with favourable results and will hopefully be deployed on all our servers over the course of the next year allowing testing on a huge data set. In addition we are also looking towards using JIT compilers to deploy on OeRC's Campus Grid.

Using Martlet, it has been possible to describe a wide range of algorithms, including algorithms for performing Singular Value Decomposition, North Atlantic Oscillation and Least Squares. While this model restricts some algorithms, there are new algorithms such as the one used to perform an SVD, that has been developed to work with this new programming model. This raises the interesting possibility of a whole set of new algorithms just waiting to be discovered once people start to think about programming in this new way.

While this work is perfectly viable both through its stand alone middleware, and through the use of JIT compilers, hopefully the ideas in Martlet will then be absorbed into the next generation of workflow languages. This will allow both existing and future languages to deal with a type of problem that thus far has not been addressed, but will become ever more common as we generate larger and larger data sets.

9. ACKNOWLEDGMENTS

This work is funded by the Natural Environmental Research Council. The author would like to thank Dr Andrew Martin and ClimatePrediction.net for all their help, as well as the reviewers for their feed back.

10. REFERENCES

- [1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.
- [2] T. Andrews, F. Curbera, H. Doholakia, Y. Golland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smitth, S. Thatte, I. Trickovic, and S. Weerwarana. BP4WS. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, 2003.
- [3] Apache Software Foundation. *Apache Axis*, 2005. URL: <http://ws.apache.org/axis/>.
- [4] Apache Software Foundation. *The Apache Jakarta Project*, 2005. URL: <http://jakarta.apache.org/tomcat/>.
- [5] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, 1998.
- [6] C. Christensen, T. Aina, and D. Stainforth. The challenge of volunteer computing with lengthy climate modelling simulations. In *Proceedings of the 1st IEEE Conference on e-Science and Grid Computing*, Melbourne, Australia, December 2005.
- [7] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. White. 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.
- [8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Technical report, Google Inc, December 2004.
- [9] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Pegasus: Planning for execution in grids. Technical report, Information Sciences Institute, 2002.
- [10] S. Ghemawat, H. Gobioff, and S.-T. 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.
- [11] D. Goodman. Martlet; a scientific work-flow language for abstracted parallelisation. In S. J. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2006*. National e-Science Centre, National e-Science Centre, September 2006.
- [12] D. Goodman and A. Martin. Scientific middleware for abstracted parallelisation. Technical Report RR-05-07, Oxford University Computing Lab, November 2005.

- [13] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
- [14] OMII. The omii product roadmap. Technical report, OMII, 2004. URL: <http://www.omii.ac.uk/roadmap.htm>.
- [15] D. Stainforth, J. Kettleborough, A. Martin, A. Simpson, R. Gillis, A. Akkas, R. Gault, M. Collins, D. Gavaghan, and M. Allen. Climateprediction.net: Design principles for public-resource modeling research. In *14th IASTED International Conference Parallel and Distributed Computing and Systems*, Nov 2002.
- [16] The MONET Consortium. Monet architecture overview. Technical report, The MONET Consortium, 2003. URL: <http://monet.nag.co.uk/cocoon/monet/>.
- [17] W3C. *Simple Object Access Protocol (SOAP) 1.2*, 2003. URL: <http://www.w3c.org/TR/SOAP>.
- [18] D. C. H. Wallom and A. E. Trefethen. Oxgrid, a campus grid for the university of oxford. In S. J. Cox, editor, *Proceedings of the UK e-Science All Hands Meeting 2006*. National e-Science Centre, National e-Science Centre, September 2006.