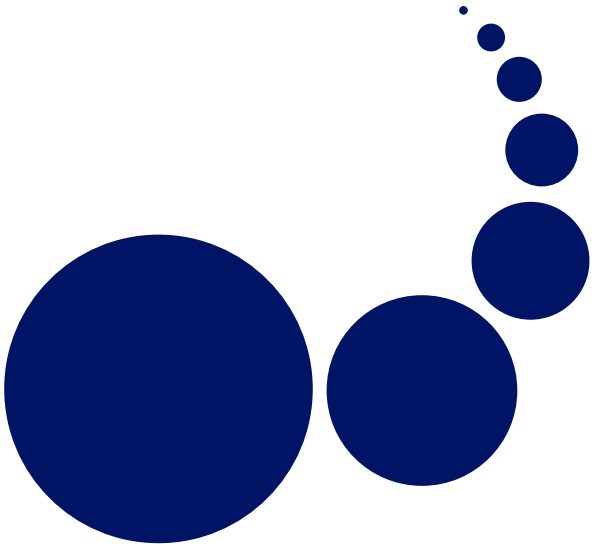
SCALABLE COMPUTING
Practice and Experience

Special Issue: Practical Aspects of High-Level Parallel Programming

Editor: Frédéric Loulergue



Selected papers from the ISPDC'05 Conference

Editors: Richard Olejnik, Marek Tudruj

Volume 7, Number 3, September 2006

ISSN 1895-1767

SWPS

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 7, Number 3, September 2006

TABLE OF CONTENTS

Scalable Computing: Practice and Experience
Volume 7, Number 3, p. i. http://www.scpe.org
ISSN 1895-1767

EDITORIAL: QUO VADIS GRID COMPUTING

An October 2003 report by the 451 Group was entitled "Grids 2004: From Rocket Science to Business Service". In the Key Findings Section we can find several optimistic predictions. For example: "Grid computing developments will result in commercially viable, mainframe-like performance and manageability across distributed systems within the next 12 months". The report listed financial services, life sciences and manufacturing vertical markets as the early adopters of grid computing. At the same time the report warned that grid computing can stumble if vendors do not provide sufficient security and authentication.

In 2006 we can add other observations. The most important of them is grid computing has become a key technology for products and services of several significant vendors including IBM, HP and Sun. In addition grid computing has captured imagination of most IT professionals from academic and government institutions. Every year from 2000 we have witnessed many respectable conferences discussing, advocating and pushing grid computing. Despite this formidable effort and related expense we cant say in 2006 that grid computing has succeeded and has become a common utility computing technology. A simple explanation is related to human factors and the existing computing culture. After several decades of computing based on hardware ownership and total control of resources and services grid computing is a major paradigm discontinuity. Those who need very large amounts of computing or need to collaborate with geographically dispersed partners have used grid computing successfully. Others still wait. More complex explanations must take into account other factors such as: business data and software security, benchmarking difficulties, IT staff retraining and the expenses of the paradigm shift implementation.

In conclusion what we need now is not more vendor business strategy declarations and scientific conferences pushing grid computing but a careful and open analysis of the stumbling blocks on the way to Enterprise Grid Computing. This analysis could be done by several leading vendors together with potential commercial users who are not interested in early adoptions but safe, economically justified real life applications of grid computing. The commercial user companies participating in the analysis should not be only large rich businesses that are afraid of missing the boat and can afford some experimentation. The analysis should include small and medium companies that need large amounts of computing for their real-life business functions.

Janusz S. Kowalik


SCALABLE COMPUTING: PRACTICE AND EXPERIENCE ISSN 1895-1767
Volume 7, Number 3, p. iii. http://www.scpe.org

GUEST EDITOR'S INTRODUCTION: PRACTICAL ASPECTS OF HIGH-LEVEL PARALLEL PROGRAMMING

Computational Science applications are getting more and more complex to develop and require more and more computing power. Parallel and grid computing provide solutions to this increasing need for computing power. High level languages offer a high degree of abstraction which eases the development of complex systems. By basing them on formal semantics, it even becomes possible to certify the correctness of critical parts of the application. Algorithmic skeletons, parallel extensions of functional languages such as Haskell and ML, as well as parallel logic and constraint programming or parallel execution of declarative programs such as SQL queries, etc. have all produced methods and tools to improve the price/performance ratio of parallel software, and broaden the range of target applications.

This special issue of *Scalable Computing: Practice and Experience* presents recent work of researchers in these fields. These articles are a selection of extended and revised versions of papers presented at the second international workshop on Practical Aspects of High-Level Parallel Programming (PAPP), affiliated to the International Conference on Computational Science (ICCS 2005). The PAPP workshops focus on practical aspects of high-level parallel programming: design, implementation and optimization of high-level programming languages and tools (performance predictors working on high-level parallel/grid source code, visualisation of abstract behaviour, automatic hotspot detectors, high-level GRID resource managers, compilers, automatic generators, etc.), applications in all fields of computational science, benchmarks and experiments. The PAPP workshops are aimed both at researchers involved in the development of high level approaches to parallel and grid computing and at computational science researchers who are potential users of these languages and tools.

One concern in the development of parallel programs is the prediction of their performance from the source code. This is valuable to enable for their optimization, or to fit the resources needed by the program into the resources offered by the architecture. In their paper, *Empirical Parallel Performance Prediction from Semantics-Based Profiling*, Norman Scaife, Greg Michaelson and Susumu Horiguchi propose a hybrid approach by combining static analytic cost models for algorithmic skeletons with dynamic information gathered from the sequential instrumentation of higher-order functions.

If high-performance computing is mainly concerned with processing ressources, solving large problems also raises memory ressources issues. *Dynamic Memory Management in the Loci Framework* by Yang Zhang and Edward A. Luke provides a solution for the *Loci* declarative high-performance data-parallel programming system.

Grid systems offer a tremendous computing power. Nevertheless, this power is far from being effectively exploited. In addition to technical problems related to portability and access, grid computing needs suited programming paradigms. A. D. Al Zain et al. present in *Managing Heterogeneity in a Grid Parallel Haskell*, GridGUM an initial port of the distributed virtual shared-memory implementation of Glasgow Parallel Haskell for computational grids.

I would like to thank all the people who made the PAPP workshop possible: the organizers of the ICCS conference, the other members of the programme committee: Marco Aldinucci (CNR/Univ. of Pisa, Italy), Rob Bisseling (Univ. of Utrecht, The Netherlands), Frank Dehne (Griffith Univ., Australia), Alexandros Gerbessiotis (NJIT, USA), Stephen Gilmore (Univ. of Edinburgh, UK), Clemens Grelck (Univ. of Luebeck, Germany), Sergei Gorlatch (Univ. of Muenster, Germany), Isabelle Guérin-Lassous (INRIA, France), Zhenjiang Hu (Univ. of Tokyo, Japan), Fethi A. Rabhi (Univ. of New South Wales, Australia), Casiano Rodríguez León (Univ. La Laguna, Spain). I also thank the other referees for their efficient help. Finally I thank all authors who submitted papers for their interest in the workshop, the quality and variety of the research topics they proposed.

Frédéric Loulergue
Laboratoire d'Informatique
Fondamentale d'Orléans,
University of Orléans,
rue Léonard de Vinci,
B. P. 6759 F-45067
Orleans Cedex 2,
FRANCE

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 1–8. http://www.scpe.org
ISSN 1895-1767

EMPIRICAL PARALLEL PERFORMANCE PREDICTION FROM SEMANTICS-BASED PROFILING

NORMAN SCAIFE*, GREG MICHAELSON†, AND SUSUMU HORIGUCHI‡

Abstract. The PMLS parallelizing compiler for Standard ML is based upon the automatic instantiation of algorithmic skeletons at sites of higher order function (HOF) use. Rather than mechanically replacing HOFs with skeletons, which in general leads to poor parallel performance, PMLS seeks to predict run-time parallel behaviour to optimise skeleton use.

Static extraction of analytic cost models from programs is undecidable, and practical heuristic approaches are intractable. In contrast, PMLS utilises a hybrid approach by combining static analytic cost models for skeletons with dynamic information gathered from the sequential instrumentation of HOF argument functions. Such instrumentation is provided by an implementation independent SML interpreter, based on the language's Structural Operational Semantics (SOS), in the form of SOS rule counts. PMLS then tries to relate the rule counts to program execution times through numerical techniques.

This paper considers the design and implementation of the PMLS approach to parallel performance prediction. The formulation of a general rule count cost model as a set of over-determined linear equations is discussed, and their solution by single value decomposition, and by a genetic algorithm, are presented.

Key words. Parallel computation, profiling, performance prediction, program transformation.

1. Introduction. The optimal use of parallel computing resources depends on placing processes on processors to maximise the ratio of processing to communication, and to balance loads, to ensure that all processors are maximally and gainfully occupied. These two requirements are strongly related: moving processing from one processor to another in search of load balance changes inter-processor communication patterns. However, in the absence of standard methodologies or generic support tools, process/processor placement remains something of a black art, guided primarily by empirical experimentation on the target architecture. This can be a long and painstaking activity, which ties up scarce, costly parallel resources at the expense of other users. It would be most desirable to develop analytic techniques to guide process placement which do not depend on direct use of the target system.

Process placement is greatly simplified given accurate measures of individual process communication and processing behaviour. However, such measures, like most interesting properties of programs, are in general undecidable. The only alternative is to use to some mix of approximative techniques for static and dynamic analysis.

One approach is to focus on general patterns of processing with known behavioural characteristics. For parallel programming, Cole's *algorithmic skeletons* [9] form a popular class of patterns, including the data parallel task farm and the binary tree structured divide and conquer. Here, simple analytic cost models have been constructed which give good predictions of parallel behaviour when instantiated appropriately [18].

Alas, this just pushes the problem down a level as it is still necessary to characterise the task specific behaviours that such patterns are to be populated with. For non-conditional and non-repetitive program fragments, precise measures may be found through static cost modeling. However, in the more general cases, either approximative static analyses or empirical measures must still be used.

Static methods such as computational complexity analysis [12] or microanalysis [8] break down in the presence of conditional and repetitive constructs. Computational complexity analysis implementations are often limited to libraries of known instances in these cases. Microanalysis has similar limitations, requiring the solutions to sets of difference equations which, in turn, lack direct analytic solutions.

Dynamic measures may be based on *sampling* and *counting* methods. Sampling is where the program is interrupted at regular intervals and a picture of where processing is concentrated can be built up. This has been used for Standard ML [4]. Counting is where passage through specific points in the program are recorded. Examples of this are used in the PUFF compiler [7] and the SkelML compiler [6].

*LASMEA, Blaise Pascal University, Les Cezeaux, F-63177 Aubiere cedex, France, (Norman.Scaife@lasmea.univ-bpclermont.fr).

†Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, (G.Michaelson@hw.ac.uk).

‡Department of Computer Science, Graduate School of Information Sciences, Tohoku University, Aobayama 6-3-09, SENDAI 980-8579, JAPAN, (susumu@ecei.tohoku.ac.jp).

Whatever technique is used to cost a program, the final measure must be related back to an actual performance on the target architecture. Typically, both static and empirical measures give counts of features, such as the presence of known operations, or behaviours, such as the number of times a construct is carried out. The equivalent costs on the target architecture may be established in terms of individual CPU instructions, either by direct instrumentation or from manufacturer's specifications. This approach gives accurate predictions from sequential profiling but is highly implementation dependent.

An alternative is to time representative programs on the target architecture and relate the times back to the modeled costs. This offers a high degree of implementation independence but requires well chosen exemplars and, like rule counting, is very dependent on the test data.

2. Background. The PMLS (Parallelising ML with Skeletons) compiler for Standard ML [14] translates instances of a small set of common higher-order functions (HOFs) into parallel implementations of algorithmic skeletons [9]. As part of the design of the compiler, we wish to implement performance-improving transformations guided by dynamic profiling. We contend that the rules that form the dynamic semantics of Standard ML provide an ideal set of counting points for dynamic profiling since they capture the *essence* of the computation at an appropriate level of detail. They also arise naturally during the evaluation of an SML program, eliminating difficult decisions about where to place counting points. Finally, the semantics provides an implementation independent basis for counting.

Our approach follows work by Bratvold [6] who used SOS rule counting, plus a number of other costs, to obtain sequential performance predictions for unnested HOFs. Bratvold's work was built on Busvine's sequential SML to Occam translator for linear recursion [7] and was able to relate abstract costs in the SML prototype to specific physical costs in the Occam implementation.

Contemporaneous with PMLS, the FAN framework [2] uses costs to optimise skeleton use through transformation. FAN has been implemented within META [1] and applied to Skel-BSP, using BSP cost models and parameterisations. However, costs of argument functions are not derived automatically.

Alt *et al.* [3] have explored the allocation of resources to Java skeletons in computational Grids. Their skeleton cost models are instantiated by counting instruction executions in argument function byte code and applying an instruction timing model for the target architecture. As in PMLS, they solve linear equations of instruction counts from sequential test program runs to establish the timing model. However, the approach does not seem to have been realised within a compiler.

Hammond *et al.* [11] have used Template Haskell to automatically select skeleton implementations using static cost models at compile time. This approach requires substantial programmer involvement, and argument function costs are not derived automatically.

Holger *et al.* [5] use dynamic measurements from sequential versions to optimize skeleton implementations but have only applied it to specific algorithms.

The main goal of our work is to provide predictions of sequential SML execution times to drive a transformation system for an automated parallelizing SML compiler. In principle, purely static methods may be used to derive accurate predictions, but for very restricted classes of program. From the start, we wished to parallelise arbitrary SML programs and necessarily accepted the limitations of dynamic instrumentation, in particular incomplete coverage and bias in test cases leading to instability and inaccuracy in predictions. However, we do not require predictions to be highly accurate so long they order transformation choices correctly.

In the following sections, we present our method for statistical prediction of SML based on the formal language definition, along with a set of test programs. We discuss the accuracy of our method and illustrate its potential use through a simple example program.

3. Semantic rules and performance prediction. SML [15] was one of the first languages to be fully formally specified. The definition is based on Plotkin's Structural Operational Semantics (SOS) [16], where the evaluation of a language construct is defined in terms of the evaluation of its constituent constructs. Such evaluation takes place in what is termed a *background*, which usually consists of an *environment* and a *state*. Environments bind identifiers and values, are modified by definitions and are inspected to return the values of identifiers in expressions. States bind addresses and values, are modified by assignments to references and inspected to return values from references.

A typical rule has the form:

$$\frac{B_1 \vdash e_1 \Rightarrow v_1 \quad \ldots B_i \vdash e_i \Rightarrow v_i \ldots \quad B_N \vdash e_N \Rightarrow v_N}{B \vdash e \Rightarrow v}$$

This defines the evaluation of language construct e in background B to give result v, in terms of the prior evaluation of the N constituent constructs e_i in backgrounds B_i to give constituent results v_i.

For example, the rule for a local definition (93):

$$\frac{E \vdash dec \Rightarrow E' \quad E + E' \vdash exp \Rightarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v}$$

says that if the evaluation of the declaration *dec* in environment E gives a new environment E', and the evaluation of expression *exp* in the environment E extended with the new environment E' gives a value v, then the evaluation of the local definition `let` *dec* `in` *exp* `end` in the environment E gives the value v.

Our methodology for dynamic profiling is to set up a dependency between rule counts and program execution times, and solve this system on a learning-set of programs designated as "typical".

Suppose there are N rules in an SOS and we have a set of M programs. Suppose that the time for the ith program on a target architecture is T_i, and that the count for the jth rule when the ith program is run on a sequential SOS-based interpreter is R_{ij}. Then we wish to find weights W_j to solve:

$$\begin{aligned} R_{11}W_1 + R_{12}W_2 + \ldots + R_{1N}W_N &= T_1 \\ R_{21}W_1 + R_{22}W_2 + \ldots + R_{2N}W_N &= T_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\ R_{M1}W_1 + R_{M2}W_2 + \ldots + R_{MN}W_N &= T_M \end{aligned}$$

such that given a set of rule counts for a new program P we can calculate a good prediction of the time on the target architecture T_P from:

$$R_{P1}W_1 + R_{P2}W_2 + \ldots + R_{PN}W_N = T_P$$

This linear algebraic system can be expressed in matrix form as:

$$RW \ = \ T \tag{3.1}$$

Then given a set of rule counts for a new program P we can calculate a good prediction of the time on the target architecture T_P from:

$$R_{P1}W_1 + R_{P2}W_2 \ldots + R_{PN}W_N = T_P \tag{3.2}$$

These are then substituted into the skeleton cost model for skeleton `S`. For the currently supported list HOFs `map` and `fold` of function `f` over list `L`, the models take the very simple form, parallel cost:

$$\text{Cost}_S \ = \ C_1 * \text{size}(\texttt{L}) + C_2 * \text{size}(TX) + C_3 * \text{size}(RX) + C_4 * T_{\texttt{f}} \tag{3.3}$$

where TX is the message required to transmit the arguments to function `f`, RX is the message returning the result of `f` and $T_{\texttt{f}}$ is the time to process `f`. The coefficients $C_1 \ldots C_4$ are determined by measurements on the target architecture, over a restricted range of a set of likely parameters[19]. We then deploy a similar fitting method to this data, relating values such as communications sizes and instance function execution times to measured run-times.

4. Solving and predicting. We have tried to generate a set of test programs which, when profiled, include all of the rules in the operational semantics which are fired when our application is executed. We have also tried to ensure that these rules are as closely balanced as possible so as not to bias the fit towards more frequently-used rules.

We have divided our programs into a *learning* and a *test* set. The learning set consists of 99 "known" programs which cover a fair proportion of the SML language. These include functions such as mergesort, maximum segment sum, regular expression processing, random number generation, program transformation, ellipse fitting and singular value decomposition.

The test set consists of 14 "unknown" programs which, in turn, represent a fair cross-section of the learning set in terms of the sets of rules fired and the range of execution times. These include polynomial root finding,

least-squares fitting, function minimisation and geometric computations. The test set was generated by classifying the entire set of programs according to type (e.g. integer-intensive computation, high-degree of recursion) and execution time. A test program was then selected randomly from each class.

To generate the design matrix R, we take the rule counts R_i^{td} and execution time T_i^{td} for top level declaration number td. The first timing T_i^0 in each repeat sequence is always ignored reducing the effect of cache-filling. The execution times $T_i^{t_i}$ are always in order of increasing number of repeats such that $T_i^x < T_i^y$ for $x < y$. Using this and knowing that outliers are always greater than normal data we remove non-monotonically increasing times within a single execution. Thus if $T_i^{td-1} < T_i^{td} < T_i^{td+1}$ then the row containing T_i^{td} is retained in the design matrix. Also, to complete the design matrix, rules in R_{all} which are not in R_i^{td} are added and set to zero.

Some rules can be trivially removed from the rule set such as those for type checking and nesting of expressions with atomic expressions. These comprise all the rules in the static semantics. However, non-significant rules are also removed by totaling up the rule counts across the entire matrix. Thus for rule r_x and a threshold θ, if:

$$\sum_{i=0}^{n}\sum_{j=0}^{t_i} R_i^j[r_x].c < \theta \sum_{i=0}^{n}\sum_{j=0}^{t_i} R_i^j[r_{max}].c \tag{4.1}$$

r_{max} is the most frequent rule and $R_i^j[r_k].c$ means the count for rule r_k in the list of rule counts R_i^j. Thus rules with total counts less than a threshold value times the most frequently fired rule's total count have their columns deleted from the rule matrix R. This threshold is currently determined by trial and error. The execution time vector T_n is generated from the matching execution times for the surviving rows in the rule matrix.

Fitting is then performed and the compiler's internal weights updated to include the new weights. Performance prediction is then a simple application of Equation 3.1, where R is the set of rules remaining after data-workup and W is the set of weights determined by fitting. For verification, the new weights are applied to the original rule counts giving reconstructed times T_{recon} and are compared with the original execution times T_n.

Once the design matrix is established using the learning set, and validated using the test set, we can then perform fitting and generate a set of weights. We have experimented with *singular value decomposition* (SVD) to solve the system as a linear least-squares problem [17]. We have also adapted one of the example programs for our compiler, a parallel *genetic algorithm* (GA) [13], to estimate the parameters for the system.

5. Accuracy of fitting. Our compilation scheme involves translating the Standard ML core language as provided by the ML Kit Version 1 into Objective Caml, which is then compiled (incorporating our runtime C code) to target the parallel architecture. We have modified the ML Kit, which is based closely on the SML SOS, to gather rule counts directly from the sequential execution of programs. The ML Kit itself has evolved into a sophisticated compiler with profiling tools but the effort which would be required to incorporate our existing system into the current implementation of the ML Kit would be prohibitive. For cleaner sources, however, we would consider Hamlet[1] which is intended as a reference implementation of the SML definition.

Using an IBM RS/6000 SP2, we ran the 99 program fragments from the learning set using a modest number of repeats (from 10 to about 80, depending upon the individual execution time). After data cleanup, the resulting design matrix covered 41 apply functions[2] and 36 rules from the definition, and contained 467 individual execution times.

Applying the derived weights to the original fit data gives the levels of accuracy over the 467 measured times shown in Figure 5.1. This table presents a comparison of the minimum, maximum, mean and standard deviation of the measured and reconstructed times for both fitting methods. The same summary is applied to the percentage error between the measured and reconstructed times.

First of all, the errors computed for both the learning and test sets look very large. However, an average error of 25.5% for SVD on the learning set is quite good considering we are estimating runtimes which span a scale factor of about 10^4. Furthermore, we are only looking for a *rough* approximation to the absolute values. When we apply these predictions in our compiler it is often the *relative* values which are more important and these are much more accurate although more difficult to quantify.

[1] `http://www.ps.uni-sb.de/hamlet/`

[2] Apply functions are external primitive functions called by the SML core language.

	Fit	χ^2	Time (s)	Min	Max	Mean	Std. Dev.
Learning	x		Measured	5.11×10^{-6}	0.00235	0.000242	0.000425
Set	SVD	4.1×10^{-7}	Reconstructed	-2.65×10^{-6}	0.00239	0.000242	0.000424
		% Error	0.00571%	267.0%	25.5%	41.3%	
	GA	4.9×10^{-5}	Reconstructed	5.98×10^{-8}	0.00163	0.000179	0.000247
		% Error	0.00977%	1580.0%	143.0%	249.0%	
Test Set	x		Measured	8.61×10^{-6}	0.0399	0.00221	0.0076
	SVD		Reconstructed	-8.06×10^{-5}	0.0344	0.00195	0.00656
		% Error	0.756%	836.0%	158.0%	208.0%	
	GA		Reconstructed	1.67×10^{-7}	0.01600	0.000965	0.000304
		% Error	1.56%	284.0%	67.9%	71.1%	

FIG. 5.1. *Summary of fit and prediction accuracy*

The SVD is a much more accurate fit than GA as indicated by the χ^2 value for the fit. However, the SVD fit is much less stable than the GA fit as evidenced by the presence of negative reconstructed times for SVD. This occurs at the very smallest estimates of runtime near the boundaries of the ranges for which our computed weights are accurate. The instability rapidly increases as the data moves out of this region.

These points are graphically illustrated in Figure 5.2 which shows how the errors are greater for smaller time measurements and shows the better quality of fit for SVD.

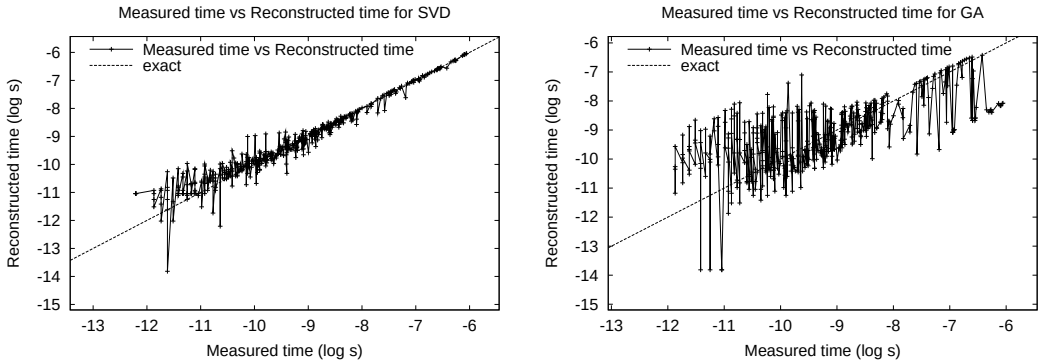


FIG. 5.2. *Quality of fit for SVD and GA*

In summary, SVD results in a fast, accurate fit for the given data but is prone to numerical instability[3] which limits the range over which the generated weights are valid. The GA, on the other hand, is very slow and does not result in accurate fits to the given data but is less prone to the problems of numerical instability. GAs also provide a very simple method of experimenting with constraints, such as forcing the weights to be strictly positive. Applying linear constraints to SVD is also possible but would require non-trivial modifications to the existing routine.

6. Performance Prediction Example. As part of the PMLS project we have used *proof-planning* to construct a synthesiser which extracts HOFs from arbitrary recursive functions [10]. For example, given the following program which squares the elements of a list of lists of integers:

```
fun squares [] = []
  | squares ((h:int)::t) = h*h::squares t

fun squs2d [] = []
  | squs2d (h::t) = squares h::squs2d t
```

the synthesizer generates the six programs shown in Figure 6.1. Note that there is no parallelism in this program suitable for our compiler and we would expect our predictions to validate this.

[3]By *numerical instability* we mean large and small coefficients of the fit canceling each other out which gives very accurate fits within a small region of coefficients but which prevent extrapolation to values of coefficients outside of the range of the test data. Note that this is not manifested in the spread of points in Figure 5.2 which represents accuracy of fit but not numerical instability.

```
1.  val squs2d = fn x => map (fn y => map (fn (z:int) => z*z) y) x
2.  val squs2d =
        fn x => foldr (fn y => fn z => (map (fn (u:int) => u*u) y::z)) [] x
3.  val squs2d =
        fn x => map (fn y => foldr (fn (z:int) => fn u => z*z::u) [] y) x
4.  val squs2d =
        fn x => foldr (fn y => fn z =>
                    foldr (fn (u:int) => fn v => u*u::v) [] y::z) [] x
5.  val squs2d = fn x => map (fn y => squares y) x
6.  val squs2d = fn x => foldr (fn y => fn z => squares y::z) [] x
```

FIG. 6.1. *Synthesizer output for* `squs2d`

V	Position	HOF	Rules	T_{SVD}	T_{GA}	$T_{measured}$
1	outer	map	21	2.63	5.56	8.61
	inner	map	8	0.79	1.40	3.36
2	outer	fold	21	4.97	6.01	9.17
	inner	map	8	0.79	1.40	3.14
3	outer	map	20	1.73	7.53	12.6
	inner	fold	15	12.5	3.66	3.71
4	outer	fold	20	4.06	7.98	11.1
	inner	fold	15	12.5	3.66	3.53
5	single	map	19	3.58	3.45	6.65
6	single	fold	19	5.91	3.90	7.97

FIG. 6.2. *Predicted and measured instance function times (μS)*

We require the execution times for the instance functions to the `map` and `foldr` HOFs. We have not yet automated the collection of this data or linked the output from the performance prediction into the compiler so we present a hand analysis of this code.

Figure 6.2 shows the predicted instance function execution times for the two fitting methods alongside the actual measured times. The input data is a 5×5 list of lists of integers. The predictions are in roughly the correct range but differ significantly from the measured times. Despite the greater accuracy of the SVD fit to the learning-set data, the GA-generated weights give more consistent results compared to actual measured values. This is due to the numerical instability of the SVD fit. However, these discrepancies are sufficient to invert the execution times for nested functions. For instance, for Version 3 the inner fold instance function takes longer than the outer one, even though the outer computation encompasses the inner.

Applying the skeleton performance models to the measured instance function times, plus data on communications sizes gathered from sequential executions, gives the predicted parallel run-times for 1, 2, 4 and 8 processors, shown in Figure 6.3.

The GA- and SVD-predicted instance function times give identical predictions for parallel run-times. This is because the parallel performance model is in a range where the run-time is dominated by communications rather than computation. However, the P_1 predictions are erroneous. These predictions represent an extrapolation of a parallel run onto a sequential one which has no overheads such as communication. This also applies to the P_2 predictions, where these overheads are not accurately apportioned. Furthermore, the absolute values of the predictions are unreliable. For the P_8 values, some are accurate but some are out by an order of magnitude. The most relevant data in this table is the ratio between the P_4 and P_8 values. This, in most cases, increases as the number of processors increases, indicating slowdown.

7. Conclusions. Overall, our experimentation gives us confidence that combining automatic profiling with cost modeling is a promising approach to performance prediction. We now intend to use the system as it stands in implementing a performance-improving transformation system for a subset of the SML language. As well as exploring the automation of load balancing, this gives us a further practical way to assess the broader utility of our approach.

We already have a complex system of compiler pragmas which allow some degree of programmer control over both the performance prediction and the transformation system. This was instituted to allow debugging

V	Position	HOF	P/M	P_1	P_2	P_4	P_8
1	outer	map	P	1.6000	3.230	6.480	12.990
			M	0.1423	6.806	5.279	4.910
	inner	map	P	3.2700	4.900	8.150	14.660
			M	0.2846	35.200	15.620	14.440
2	outer	fold	P	7.3700	10.940	18.070	32.340
			M	0.1617	4.204	3.101	3.634
	inner	map	P	3.2700	4.900	8.150	14.660
			M	0.3040	35.360	14.900	14.940
3	outer	map	P	1.6000	3.230	6.480	12.990
			M	0.2205	7.314	3.923	4.739
	inner	fold	P	14.2000	17.760	24.900	39.170
			M	0.3875	26.020	14.570	15.770
4	outer	fold	P	7.3700	10.940	18.070	32.340
			M	0.2344	5.058	2.907	4.047
	inner	fold	P	14.2000	17.760	24.900	39.170
			M	0.3907	23.080	13.200	16.110
5	single	map	P	1.6000	3.230	6.480	12.990
			M	0.1375	6.590	4.092	4.570
6	single	fold	P	7.3700	10.940	18.070	32.340
			M	0.1587	4.024	3.002	3.750

FIG. 6.3. *Predicted (P) and measured (M) parallel run-times (mS)*

and testing of the profiling mechanism but has wider implications for the development process where it could be useful for example to help the compiler when it is unable to get accurate predictions or gets stuck in the search space. Note that since our ultimate goal is fully automated parallelism we have not elaborated upon this point.

While we have demonstrated the feasibility of semantics-based profiling for an entire extant language, further research is needed to enable more accurate and consistent predictions of performance from profiles. Our work suggests a number of areas for further study:

- introducing non-linear costs into the system relating profile information and runtime measurements. The system would no longer be in matrix form and may require the use of generalised function minimisation instead of deterministic fitting;
- identifying which semantic rules counts are most significant for predicting run times, through standard statistical techniques for correlation and factor analyses. Focusing on significant rules would reduce profiling overheads and might enable greater stability in the linear equation solutions;
- investigating the effects on prediction accuracy of optimisations employed in the back end compiler. Such optimisations fundamentally affect the nature of the association between the language semantics and implementation;
- systematically exploring the relationship between profiles and run-times for one or more constrained classes of recursive constructs, in the presence of both regular and irregular computation patterns. Our studies to date have been of very simple functions and of unrelated substantial exemplars;
- modeling explicitly aspects of implementation which are subsumed in the semantics notation. In particular the creation and manipulation of name/value associations are hidden behind the semantic notion of *environment*.

Acknowledgement. This work was supported by the Japan JSPS Postdoctoral Fellowship P00778 and UK EPSRC grants GR/J07884 and GR/L42889.

REFERENCES

[1] M. ALDINUCCI, *Automatic Program Transformation: The META Tool for Skeleton-based Languages*, in Constructive Methods for Parallel Programming, S. Gorlatch and C. Lengauer, eds., vol. 10 of Advances in Computation: Theory and Practice, NOVA Science, 2002.

[2] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelegatti, *Towards Parallel Programming by Transformation: The FAN Skeleton Framework*, Parallel Algorithms and Applications, 16 (2001), pp. 87–122.
[3] M. Alt, H. Bischof, and S. Gorlatch, *Program Development for Computational Grids Using Skeletons and Performance Prediction*, Parallel Processing Letters, 12 (2002), pp. 157–174.
[4] A. W. Appel, B. F. Duba, and D. B. MacQueen, *Profiling in the Presence of Optimization and Garbage Collection*, Tech. Report CS-TR-197-88, Princeton University, Dept. Comp. Sci., Princeton, NJ, USA, November 1987.
[5] H. Bischof, S. Gorlatch, and E. Kitzelmann, *Cost optimality and predictability of parallel programming with skeletons*, in Europar 03, vol. 2790 of LNCS, Jan 2003, pp. 682 – 693.
[6] T. Bratvold, *Skeleton-based Parallelisation of Functional Programmes*, PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.
[7] D. Busvine, *Implementing Recursive Functions as Processor Farms*, Parallel Computing, 19 (1993), pp. 1141–1153.
[8] C. Cohen, *Computer-Assisted Microanalysis of Programs*, Communications of the ACM, 25 (1982), pp. 724–733.
[9] M. I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman/MIT, 1989.
[10] A. Cook, A. Ireland, G. Michaelson, and N. Scaife, *Discovering Applications of Higher-Order Functions through Proof Planning*, Formal Aspects of Computing, 17 (2005), pp. 38–57.
[11] K. Hammond, J. Berthold, and R. Loogen, *Automatic Skeletons in Template Haskell*, Parallel Processing Letters, 13 (2003), pp. 413–424.
[12] D. L. Métayer, *ACE: An Automatic Complexity Evaluator*, ACM TOPLAS, 10 (1988), pp. 248–266.
[13] G. Michaelson and N.Scaife, *Parallel functional island model genetic algorithms through nested skeletons*, in Proceedings of 12th International Workshop on the Implementation of Functional Languages, M. Mohnen and P. Koopman, eds., Aachen, September 2000, pp. 307–313.
[14] G. Michaelson and N. Scaife, *Skeleton Realisations from Functional Prototypes*, in Patterns and Skeletons for Parallel and Distributed Computing, F. Rabhi and S. Gorlatch, eds., Springer, 2003.
[15] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT, 1990.
[16] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Tech. Report DAIMI FN-19, Arrhus University, Denmark, Sep 1981.
[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, CUP, 2nd ed., 1992.
[18] R. Rangaswami, *A Cost Analysis for a Higher-Order Parallel Programming Model*, PhD thesis, University of Edinburgh, 1995.
[19] N. R. Scaife, *A Dual Source, Parallel Architecture for Computer Vision*, PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1996.

Edited by: Frédéric Loulergue
Received: November 14, 2005
Accepted: February 1st, 2006

Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 7, Number 3, pp. 9–25. http://www.scpe.org

MANAGING HETEROGENEITY IN A GRID PARALLEL HASKELL

A. D. AL ZAIN , P. W. TRINDER , G. J. MICHAELSON*, AND H-W. LOIDL†

Abstract. Computational Grids potentially offer cheap large-scale high-performance systems, but are a very challenging architecture, being heterogeneous, shared and hierarchical. Rather than requiring a programmer to explicitly manage this complex environment, we recommend using a high-level parallel functional language, like GpH, with largely automatic management of parallel coordination.

We present *GRID-GUM*, an initial port of the distributed virtual shared-memory implementation of GpH for computational Grids. We show that, *GRID-GUM* delivers acceptable speedups on relatively low latency homogeneous and heterogeneous computational Grids. Moreover, we find that for heterogeneous computational Grids, load management limits performance.

We present the initial design of *GRID-GUM*2, that incorporates new load management mechanisms that cheaply and effectively combine static and dynamic information to adapt to heterogeneous Grids. The mechanisms are evaluated by measuring four non-trivial programs with different parallel properties. The measurements show that the new mechanisms improve load distribution over the original implementation, reducing runtime by factors ranging from 17% to 57%, and the greatest improvement is obtained for the most dynamic program.

Key words. Parallel Computing, Programming Languages

1. Introduction.

Hardware price/performance ratios and improved middleware and network technologies make cluster computing and computational Grids increasingly attractive. These architectures are typically heterogeneous in the sense that they combine processing elements with different CPU speeds and memory characteristics. Parallel programming on such heterogeneous architectures is more challenging than on classical homogeneous high performance architectures.

Rather than requiring the programmer to explicitly manage low level issues such as heterogeneity we advocate a high-level parallel programming language, specifically Glasgow parallel Haskell (GpH), where the programmer controls only a few key parallel coordination aspects. The remaining coordination aspects, including heterogeneity, are dynamically managed by a sophisticated runtime environment, Gum. Gum has been engineered to deliver good performance on classical HPCs and clusters [1].

This paper presents *GRID-GUM*, a port of Gum to computational Grids using the de-facto standard Globus Toolkit, in Section 4. Measurements in Section 6 show that *GRID-GUM* gives good performance in some instances, e. g. on homogeneous low-latency multi-clusters. However for heterogeneous architectures load management emerges as the performance-limiting issue.

We present the initial design of *GRID-GUM*2 in Section 7, which incorporates new load distribution mechanisms for virtual shared-memory over a wide area network. The new mechanisms are decentralised, obtaining complete static information during start up, and then cheaply propagating partial dynamic information during execution. The effectiveness of the new mechanisms for multi-clusters Grid environment is investigated using four non-trivial programs from a range of application areas, and with varying degrees of irregular parallelism and using both data parallel and divide-and-conquer paradigms in Section 8. Related work is discussed in Section 9, and we conclude in Section 10.

2. Grids & the Globus Toolkit.

2.1. Overview.

Grid technology is an infrastructure which provides the ability to dynamically link distributed resources as an ensemble to support these execution of large scale, resource-intensive applications [19].

The idea behind the Grid is to serve as an enabling technology for a broad set of applications in science, business, entertainment, health and other areas. Using Berman's classification [19], we are working with computational Grids, which use the Grid to aggregate substantial computational resources in order to tackle problems that cannot be solved on a single system

2.2. Globus Toolkit.

The Globus Toolkit is open source software with an open architecture, comprising a collection of software components designed to support the development of applications for high performance distributed computing environments or "Grids" [20]. The three main components are:

* School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U. K. ({ceeatia,trinder,greg}@macs.hw.ac.uk)

† Ludwig-Maximilians-Universität München, Institut für Informatik, D 80538 München, Germany, hwloidl@informatik.uni-muenchen.de

- Resource Management: allocation and management of GRID resources;
- Information Services: providing information about GRID resources;
- Data Management: accessing and managing data in a GRID environment.

Globus Toolkit is similar to a distributed operating system with uniform access to system features. Globus Toolkit uses a standard application programming interface (API) for sending data and work to other machines which can be expressed in terms of extensible *resource specification language (RSL)*, which is used as a common notation for describing resource requirements. While, RSL is no more sophisticated than other systems for cluster computing e. g. a Beowulf cluster running standard Linux distribution, there are components which might be very useful for *GRID-GUM* in the long run: for example for monitoring system behaviour.

The GRID architecture in the Globus Toolkit [26] identifies the fundamental system components, specifies the purpose and function of these components, and indicates how these components interact with one another. GRID layers defines a slim API for resource and connectivity protocols, so that collective services have a simple interface to work with; on fabric layer, many and often specialised resources are covered (e.g. storage, sensors), not just the usual for parallel computing such as memory, CPU etc. The fabric layer provides the resources that are shared by the GRID: CPU time, storage, sensors. The connectivity layer defines the core communication and authentication protocols required for GRID-specific multi-clusters transactions. In the resource layer there are *information protocols* that tells us about the state of the resource and *management protocols* that negotiate access to a resource. The collective layer includes directory services, scheduling, data replication services, workload management, col-laboratory services and monitoring services.

3. GUM and GPH.

3.1. GPH. GPH is a parallel dialect of the functional language Haskell. Its only extension to Haskell is a primitive, `par`, which indicates a possible parallel execution for a program expression. All dynamic control of the parallelism is completely implicit. This programming model encourages the generation of massive amounts of fine-grained parallelism and puts even higher importance on the efficiency of its management in the runtime environment.

3.2. GUM. GUM (*Graph reduction for a Unified Machine model*) is a parallel runtime environment, implements a functional language and is based on parallel graph reduction [7]. In this model a program is represented as a graph structure and parallelism is exploited by reducing independent subgraphs in parallel. The most natural implementation of parallel graph reduction uses a shared heap for memory management. GUM implements a virtual shared heap on a distributed memory model, using PVM as generic communication library for transferring data. For efficient compilation we use a state-of-the-art, optimising compiler, namely the Glasgow Haskell Compiler [2]. Originally GUM was defined for homogeneous clusters and currently does not consider information on latencies or the load on other nodes. GUM uses blind load distributed mechanism, where requests are sent to random processing elements (PEs).

GUM has a simple run-time model. Essentially, in the course of execution, PEs generate *sparks* representing potential parallel activities which may subsequently be realised as *threads*. Idle PEs which lack local sparks may request work from other randomly chosen PEs by sending them *fish* messages. If a fished PE does not have spare sparks then it will pass the message onto another PE. Thus, GUM utilises a *pull* approach for work stealing to dynamically balance activity across PEs.

In a homogeneous HPC, the GUM model assumes that all PEs have the same processing and communication characteristics. It is also assumed that a parallel program has sole use of the HPC so its performance is not affected by unpredictable concurrent usage. Thus load balance in GUM may be maintained without reference to run-time loads, with communication overhead from excess fishing restricted through a very simple throttling mechanism.

3.3. Communication Libraries. GUM is independent of the library used to communicate between PEs. GUM was originally based on the PVM communication library but now has been adapted to use MPI, in particular MPICH and MPICH-G2. We summarise these libraries before considering their integration into GUM. PVM(*Parallel Virtual Machine*) emerged as one of the most popular cluster message-passing systems in 1992 [21].

The MPI (*Message Passing Interface*) Standard defines a library of routines that implement the message passing model [22], and it has a richer set of constructs than PVM. MPICH is a popular implementation of the MPI standard [23]. MPICH-G2 is a Grid-enabled implementation of the MPI standard [25]. It is a port of

MPICH, built on top of services provided from the Globus Toolkit to support efficient, transparent execution in the Grid heterogeneous environments.

4. *GRID-GUM.* To port GPH to computational GRIDs its GUM runtime environment must be ported to the GRID collective layer as depicted in Figure 4.1. GUM sits above the collective layer provided by Globus Toolkit, which in turn provides a unified distributed environment on the clusters comprising the underlying GRID. Such integration depends on the provision of appropriate communication libraries within the collective layer to link GUM transparently to the GRID: this is considered in more detail in the next section.

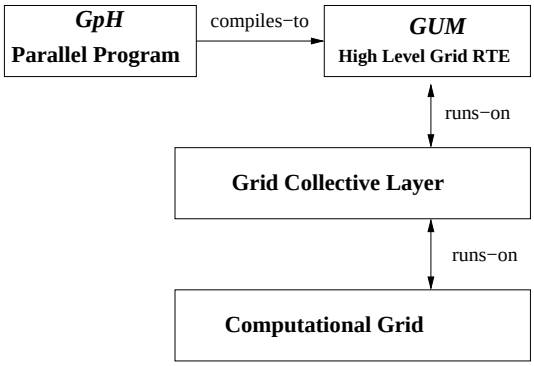


FIG. 4.1. *System Architecture*

GRID-GUM extends the existing GUM memory management, and thread management techniques. In particular, it implements a virtual shared heap over a wide-area network [3]. The communication management in *GRID-GUM* is similar to GUM, but it uses a different communication library: built around MPICH-G2, and hence the Globus Toolkit [8] as middle-ware. While GUM uses a system manager process to start and stop parallel execution, *GRID-GUM* generates an RSL file internally at the beginning of the execution. This file contains: the PE name, port number, and certificate name, environment variables, arguments for the executable program, the directory where the executable program is located in the specified PE, and the executable program's name. This RSL file is used by MPICHG2 to spawn the specified number of PEs.

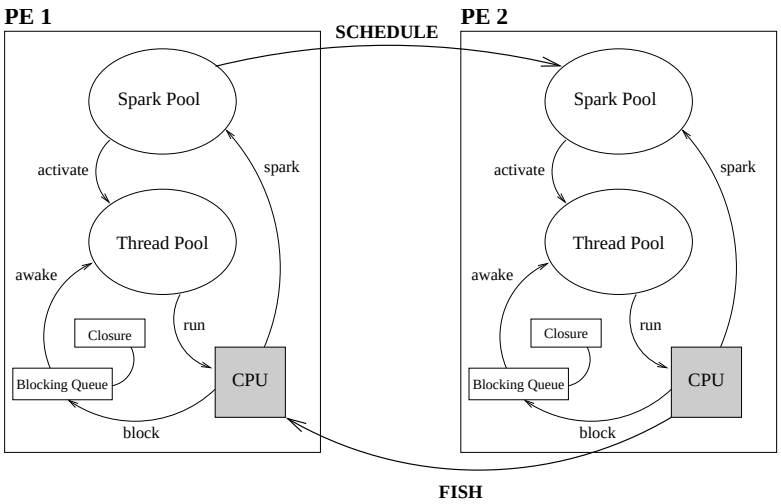


FIG. 4.2. *Interaction of the components of a* GUM *processing element.*

Figure 4.2 illustrates the load distribution mechanism in *GRID-GUM*, depicting the logical components on each PE of the *GRID-GUM* abstract machine. When activated a spark causes a new thread to be generated. Threads that are not currently being executed reside in the thread pool. When the CPU is idle, and the thread pool is empty, a spark will be activated to generate a thread. If a running thread blocks on unavailable data, it is added to the blocking queue of that node until the data becomes available.

The thick arrows between the PEs in Figure 4.2 show load distribution messages exchanged in *GRID-GUM*. Initially all processors, except for the main PE, will be idle, with no local sparks available. PE2 sends a `FISH` message to a random-chosen PE. On arrival of this message, PE1 will search for a spark and, if available, send it to PE2. This mechanism is usually called *work stealing* or *passive load distribution*, since an idle processor has to ask for work. *GRID-GUM* also improves load distribution by using *Limited Thread* mechanism which includes specifying a hard limit on the total number of live threads, i. e. runnable or blocked threads in the thread pool. Figure 4 summarises the *GRID-GUM* load distribution mechanism, it deals locating work (Figure 4.4), and handling work requests (Figure 4.3), where these activities are performed in the main scheduler loop between thread time slices.

```
IF received FISH THEN
 IF sparks availabel THEN
  send spark in SCHEDULE
   to originPE
 ELSE
  IF FISH exceed age
   THEN
   return back to orginPE
  ELSE
   destPE = random PE
    from PEs list
   send FISH to destPE
```

FIG. 4.3. *Work request*

```
IF idle THEN
 IF runnable thread THEN
  evaluate new thread
 ELSE
  IF spark in spark pool THEN
   active new spark
  ELSE
   IF last SCHEDULE from
    mainPE THEN
    destPE = mainPE
   ELSE
    destPE = random PE from
     PEs list
   send FISH to destPE
```

FIG. 4.4. *Work location*

The load distribution mechanisms in GRID-GUM

5. Measurement Framework.

5.1. Hardware Apparatus. The measurements have been performed on five Beowulf clusters: three located at Heriot-Watt Riccarton campus (*Edin1, Edin2,* and *Edin3*), a cluster located at Ludwig-Maximilians University Munich (*Muni*), and a cluster located at Heriot-Watt boarder campus(*SBC*); see Tables 5.1 and 5.2 for the characteristic of these Beowulfs.

All run-times in the coming tables represent the median of three executions to ameliorate the impact of operating system and shared network interaction. In addition, tables include at the bottom, the minimum, maximum and the geometric mean (root mean square) values.

TABLE 5.1
Characteristics of Beowulf Clusters

	CPU MHz	Cache kB	Memory kB
Edin1	534	128	254856
Edin2	1395	256	191164
Edin3	1816	512	247816
SBC	933	256	110292
Muni	1529	256	515500

TABLE 5.2
Approximate Latency between Clusters (ms)

	Edin1	Edin2	Edin3	SBC	Muni
Edin1	0.20	0.27	0.35	2.03	35.8
Edin2	0.27	0.15	0.20	2.03	35.8
Edin3	0.35	0.20	0.20	2.03	35.8
SBC	2.03	2.03	2.03	0.15	32.8
Muni	35.8	35.8	35.8	32.8	0.13

5.2. Software Apparatus. The programs measured in this experiment are classified by the communication degree, which is the number of messages the program sends per second, so we can study the impact of the latency of the network on program behaviour. Six programs are measured in this experiment. Three have low

communication degree, `parFib`, `queens` and `sumEuler`, and the other three have relatively high communication degree, `raytracer`, `matMult`, and `linSolv`.

The `parFib` computes Fibonacci numbers. The `sumEuler` program computes the sum over the application of the Euler totient function over an integer list. The `queens` program places a chess pieces on a board. The `raytracer` calculates a 2D image of a given scene of 3D objects by tracing all rays in a given scene of 3D objects by tracing all rays in a given grid, or window. The `matMult` multiples two matrices. The `linSolv` program finds an exact solution of a linear system of equations. See Table 5.3

TABLE 5.3
Programs Characteristics and Performance

Program	Application Area	Paradigm	Regularity
`queens`	AI	Div-Conq.	Regular
`parFib`	Numeric	Div-Conq.	Regular
`linSolv`	Symb. algebra	Data Para.	Limit irreg.
`sumEuler`	Nume. Analysis	Data Para.	Irregular
`matMult`	Numeric	Divi-Conq.	Irregular
`raytracer`	Graphic	Data Para.	High irreg.

6. ***GRID-GUM*** **Performance.** In developing *GRID-GUM*, a crucial first step was to ensure that GUM could seamlessly support GPH in a GRID environment. In particular, it was important to demonstrate conclusively that the HPC-oriented GUM communication layer could be modified for transparent use in a heterogeneous GRID. As discussed above, GUM communication is based on PVM, where communication in widely used GRID environments like Globus Toolkit is based on special forms of MPI. While there is some evidence that PVM and MPI offer comparable behaviours, it was not known whether the additional GRID control layers might add unacceptable overheads costs to GUM, rendering its use inappropriate for parallel functional programming support in a GRID.

TABLE 6.1
Dynamic Program Properties on 16 PEs

program Name	comm library	No of Threads	Alloc Rate MB/s	comm Degree Pkts/s	Average Pkt Size Byte
`parFib`	PVM	26595	55.3	65.5	5.5
	MPICH	26595	52.7	58.0	5.5
	MPICH-G2	26595	43.2	14.8	5.6
`sumEuler`	PVM	82	52.8	2.09	90.2
	MPICH	82	47.9	1.4	90.3
	MPICH-G2	82	45.7	0.7	90.2
`raytracer`	PVM	350	60.0	46.7	321.7
	MPICH	350	61.4	45.5	320.4
	MPICH-G2	350	49.5	62.9	323.0
`linSolv`	PVM	242	40.3	5.5	290.6
	MPICH	242	40.8	3.1	300.1
	MPICH-G2	242	26.5	2.5	276.3
`matMult`	PVM	144	39.0	67.3	208.8
	MPICH	144	40.1	52.2	213.3
	MPICH-G2	144	40.0	31.2	209.3
`queens`	PVM	24	38.8	0.2	851.8
	MPICH	24	37.0	0.2	818.9
	MPICH-G2	24	34.0	0.1	846.1

TABLE 6.2
Speedup on 16 PEs

program Name	comm library	Runtime Seq sec	Runtime 16 PE sec	Speedup Wall Clock	Speedup Exec	%variance Wall Clock	%variance Exec
`parFib`	PVM	413.7	22.8	14.8	17.1	00%	00%
	MPICH	409.4	20.5	6.8	19.8	54%	-15%
	MPICH-G2	465.1	26.3	2.3	17.6	84%	-2%
`sumEuler`	PVM	1607.1	131.8	11.1	12.1	00%	00%
	MPICH	1585.1	139.2	8.8	11.3	20%	6%
	MPICH-G2	1598.1	188.1	3.5	8.4	68%	30%
`raytracer`	PVM	2855.4	315.3	8.9	9.6	00%	00%
	MPICH	2782.7	365.2	7.8	8.9	12%	7%
	MPICH-G2	2782.7	301.7	6.8	9.2	22%	4%
`linSolv`	PVM	834.2	102.6	6.5	8.9	00%	00%
	MPICH	828.4	110.5	5.5	7.3	15%	17%
	MPICH-G2	828.9	112.2	5.1	7.3	21%	17%
`matMult`	PVM	891.9	150.2	5.9	5.9	00%	00%
	MPICH	891.9	191.9	4.6	4.6	21%	21%
	MPICH-G2	916.3	292.6	3.1	5.0	47%	15%
`queens`	PVM	2802.7	375.1	7.4	7.4	00%	00%
	MPICH	2802.7	390.9	7.1	7.1	4%	4%
	MPICH-G2	2816.4	567.8	4.9	6.2	33%	16%
Min	PVM						
	MPICH					4%	-15%
	MPICH-G2					21%	-2%
Max	PVM						
	MPICH					54%	21%
	MPICH-G2					84%	30%
Geometric Mean	PVM						
	MPICH					26%	13%
	MPICH-G2					49%	16%

6.1. Communication Library Impact. This experiment investigates the impact of using different communication libraries on the performance on a single cluster.

The measurements in this section have been performed on the Edin1 cluster. In Table 6.2, the fifth and sixth columns record the wall-clock and execution speedup. The wall-clock time is the execution time plus the startup time. The seventh and the last columns show the percentage variance of the wall-clock and execution speedup relative to the GUM/PVM implementation speedup.

Overall, the GUM/PVM implementation consistently shows the best wall-clock speedup and GUM/MPICH-G2 the worst marked as the average packet size, in GUM level, shrinks. As shown in measurements in the Table 6.1, the average packet size is relatively small for `parFib`, and `sumEuler`, and the wall-clock speedup variance is big between the different GUM implementations for these programs. For `raytracer`, `matMult`, and `linSolv` the average packet size is significantly larger and the wall-clock speedup variance is smaller.

The main source of overhead for the communication is the time needed for packing and unpacking in the communication libraries. Good performance for small packets is important for GUM, since parallel functional programs have massive amount of fine grained parallelism including many small messages. This is untypical for general parallel applications, and MPI implementations may well be tuned for the common case of large packet sizes. However, the big difference between MPICH and MPICH-G2 is related to the extra startup security checking overhead which Globus Toolkit adds for MPICH-G2

Comparison of the execution-time speedup of the GUM implementations with the different GPH programs shows that no implementation is always better than the others. However, the differences in execution-time speedup are less marked than the differences on the wall-clock speedup.

To summarise:

- For programs with long execution time the performance of GUM is independent of the communication libraries (Table 6.2);

- For small programs GUM with PVM gives the best wall clock speedup and GUM with MPICH-G2 the worst (Table 6.2);
- MPICH-G2 has a high startup cost relative to PVM or MPICH (Table 6.2).

6.2. *GRID-GUM* on Multiple Clusters.

6.2.1. Low Latency Multi-Cluster. This experiment investigates the performance impact of executing GPH programs on multiple heterogeneous clusters with moderate latency interconnect.

TABLE 6.3
Heterogeneous Clusters and Low Latency Interconnect Results

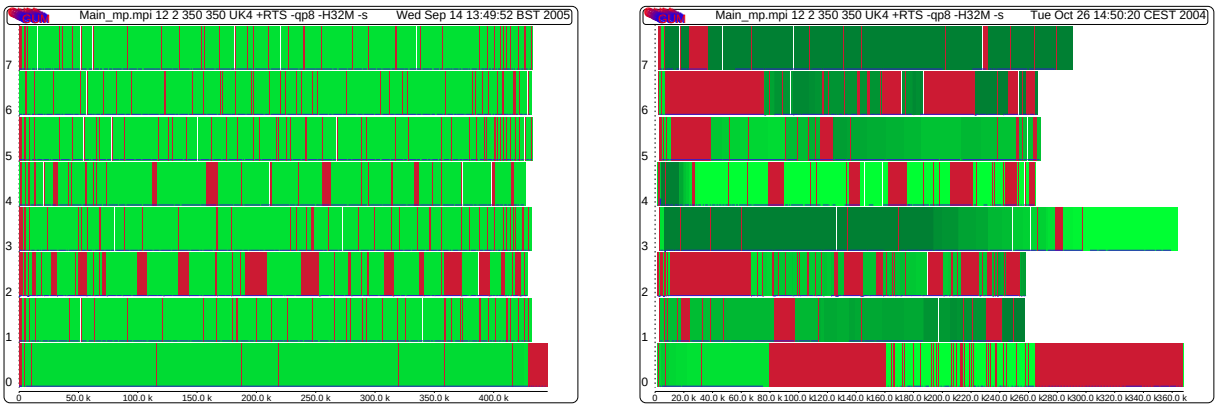
	raytracer			queens(13)		
	Speedup		Rtime	Speedup		Rtime
	F	S	Sec.	F	S	Sec.
F	1.0	3.3	1483.3	1.0	3.2	719.5
S	0.3	1.0	4894.0	0.3	1.0	2324.7
FF	1.9	6.3	772.8	1.8	6.0	384.6
FS	1.2	4.0	1199.4	0.9	3.0	753.5
SS	0.5	1.8	2698.5	0.6	1.9	1176.9
SF	0.7	2.3	2106.1	1.0	3.4	666.3
FFF	2.7	8.9	545.1	2.8	9.3	249.5
FFS	2.0	6.7	728.6	0.9	3.0	768.2
FSS	1.4	4.8	1002.3	0.9	3.1	733.6
SSS	0.8	2.9	1663.0	0.9	2.9	795.7
SSF	1.4	4.6	1047.8	1.1	3.7	627.6
SFF	1.4	4.8	1002.1	1.5	4.8	478.3
FFFF	3.4	11.4	425.9	2.7	9.0	258.2
FFFS	2.7	9.0	538.8	1.4	4.8	483.0
FFSS	2.2	7.2	675.7	1.2	4.0	578.0
FSSS	1.7	5.8	833.1	1.2	4.1	561.6
SSSS	1.1	3.8	1280.9	0.9	3.1	741.6
SSSF	1.6	5.3	916.2	1.2	4.1	560.3
SSFF	1.4	4.8	1006.7	1.2	4.1	563.5
SFFF	1.4	4.6	1046.3	1.9	6.1	375.5
FFFFF	4.0	12.9	376.7	4.0	12.8	181.1
FFFFS	3.5	11.5	422.9	2.8	9.1	254.5
FFFSS	2.9	9.4	519.2	1.3	4.2	544.9
FFSSS	2.4	7.9	615.3	1.3	4.3	530.1
FSSSS	1.9	6.4	755.6	1.2	4.0	577.7
SSSSF	1.7	5.7	850.7	1.2	4.1	560.5
SSSFF	1.8	6.2	786.0	1.5	4.9	474.3
SSFFF	1.8	6.1	790.4	1.9	6.1	375.4
SFFFF	1.9	6.5	747.6	2.2	7.3	316.5

The measurements in Table 6.3 use MPICH communication library on SBC and Edin3 Beowulf clusters described in Table 5.1. Each SBC machine is labelled S (Slow) and each Edin3 machine is labelled F (Fast). Two programs are measured: `raytracer` with relatively high communication degree, and `queens` with relatively low communication degree. The first column shows different combinations of machines. The second and the fifth columns record the speedup using F's sequential runtime for `raytracer` and `queens` respectively. The third and the sixth columns records the speedup using S's sequential runtime, and the fourth and the last columns show the wall-clock time. The first machine in the configuration string is where the program starts.

Table 6.3 shows that, replacing a local machine S by a faster remote machine F decreases the runtime and increases the speedup. For example in Table 6.3, SSS cluster requires 1663.0s to finish the computation of `raytracer`; however, if S machine has been replaced by F remote machine, the runtime is decreased by 37%. Interestingly, this result supports the idea of using a fast remote machine to improve the performance of a GPH parallel program, and it shows that *GRID-GUM* can cope with moderate latency network without modification.

However, it is observable that *GRID-GUM*, with its blind load mechanism, often gives unsatisfactory scheduling in heterogeneous GRID multi-clusters. For example, replacing one of the FFF machines by a slower remote machine S increases the runtime of `queens` from 249.5s to 768.2s, i. e. by a factor of three. Likewise, adding a slower remote machine S to two FF local machines increases the runtime of `queens` from 384.6s to 768.2s i. e. by a factor of two.

GRID-GUM shows relatively poor performance on heterogeneous cluster for many programs, and that is due to poor load management. For example, Figure 6.1 shows *GRID-GUM* per-PE activity profile for `raytracer` on a heterogeneous and a homogeneous cluster. A per-PE activity profile shows the behaviour for each of the PEs (y-axis) over execution time (x-axis). Each PE is visualised as a horizontal line, with darker shades of gray



a) Homogeneous clusters *b*) Heterogeneous cluster

FIG. 6.1. *per PE Activity Profile for* `raytracer`

(green in a colour profile) indicating a larger number of runnable threads. Gaps in the horizontal lines (red areas in the colour profile) indicate idleness.

Figure 6.1.a depicts the performance on homogeneous cluster where all PEs have the same CPU speed. Figure 6.1.b depicts the performance on heterogeneous cluster where there are four fast machines (0-3) and four slow machines (4-7). All PEs in Figure 6.1.a are uniformly loaded, and finish at the same time, in contrast the PEs in Figure 6.1.b have numerous idle periods, and finish at different times. Figure 6.1.b also shows long idle periods at the beginning of the computation, where only a small amount of parallelism is available, and blocking on data that is remotely evaluated will cause the entire PE to remain idle until new work is obtained (see the start of PE 6). Matching the profile in Figure 6.1.a, the fast processors in Figure 6.1.b (0-3) show a fairly balanced load and finish at about the same time. Towards the end only PE 3 has useful work, and the main PE 0 has to wait for it to finish. Considering the runtime of the heterogeneous cluster, 368.0s, is almost two times greater than the runtime of the homogeneous cluster, 220.0s.

To summarise:

- Replacing a local PE with a faster remote PE reduces execution time (Table 6.3);
- *GRID-GUM*'s load balancing mechanism does not deliver good scheduling in a heterogeneous GRID multi-clusters (Figure 6.1);
- In a moderate latency configuration, latency is not the dominating factor, since *GRID-GUM* can overlap communication with computation, provided a sufficient amount of parallelism is available (Table 6.3).

6.2.2. High Latency Multi-Cluster. This experiment investigates the performance impact of executing GPH programs on multiple homogeneous clusters with a high latency interconnect. We measure programs with both low and high communication degrees.

The measurements in Table 6.4, and 6.5 use MPICH-G2 communication library on the Muni and Edin2 Beowulf clusters described Table 5.1. Each Muni machine is labelled M and each Edin2 machine is labelled E

Five programs have been tested: two programs with relatively low communication degree `parFib`, and `sumEuler`, and three programs with relatively high communication degree `raytracer`, `linSolv`, and `matMult`, see Table 6.1.

For programs with a low communication degree, Table 6.4 shows that adding a remote machine M decreases the runtime. Even on multi-clusters configurations with very high latency between the clusters, the additional computational power outweighs the expensive but infrequent communication. It also shows that replacing a local machine E by a remote machine M does not grossly deteriorate performance. For example, in Table 6.4, in an EEE configuration `sumEuler` requires 899.7s to finish, machine M is added $EEEM$ the runtime decreases by 26.0%. Furthermore, replacing a local machine E by a remote machine M, yielding a EEM configuration, shows little change in the runtime (3.5%). In short, using remote machines in high latency communications does not have impact on the performance of low communication degree programs.

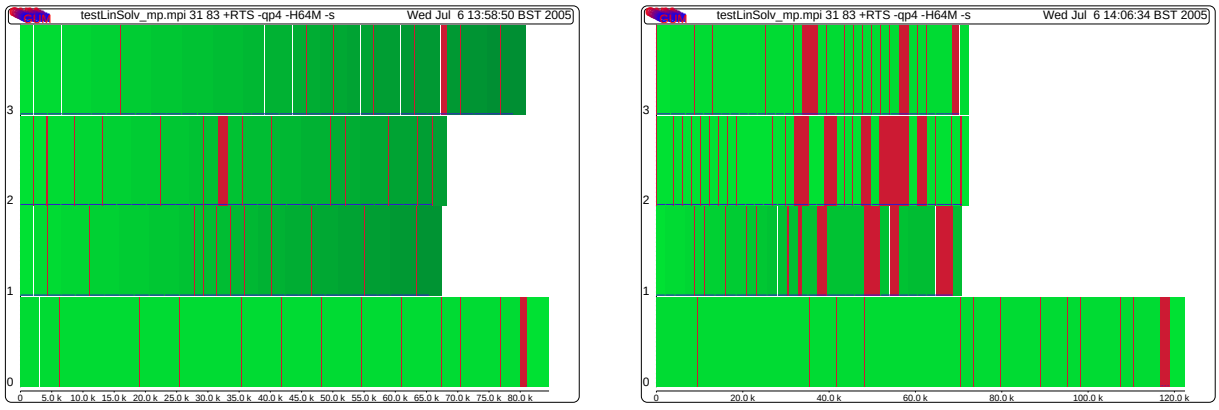
Table 6.5 shows, programs with a high communication degree, replacing a local machine with a slightly faster remote machine increases the runtime and decreases the speedup. For instance `linSolv` on two lo-

TABLE 6.4
Low Communication Degree Programs

	parFib(45)			sumEuler		
	Rtime	Spedup		Rtime	Spedup	
	Sec.	E	M	Sec.	E	M
M	867.5	1.2	1.0	3138.5	1.0	1.0
E	1070.1	1.0	0.8	3227.6	1.0	0.9
MM	431.0	2.2	2.0	1270.4	2.5	2.4
EM	480.6	2.2	1.8	1308.8	2.4	2.3
EE	536.8	1.9	1.6	1332.8	2.4	2.3
MMM	298.8	3.5	2.9	869.9	3.7	3.6
EMM	331.1	3.2	2.6	838.7	3.8	3.7
EEM	338.9	3.1	2.5	867.9	3.7	3.6
EEE	374.8	2.8	2.3	899.7	3.5	3.4

	parFib(45)			sumEuler		
	Rtime	Spedup		Rtime	Spedup	
	Sec.	E	M	Sec.	E	M
MMMM	241.9	4.4	3.5	629.7	5.1	4.9
EMMM	251.3	4.2	3.4	670.7	4.8	4.6
EEMM	268.0	3.9	3.2	665.8	4.8	4.7
EEEM	274.9	3.8	3.1	665.5	4.8	4.7
EEEE	292.9	3.6	2.9	662.2	4.8	4.7
MMMMM	205.9	5.0	4.2	523.2	6.1	5.9
EMMMM	212.7	5.0	4.0	544.0	5.9	5.7
EEMMM	226.2	4.7	3.8	553.7	5.8	5.6
EEEMM	224.7	4.7	3.8	620.8	5.1	5.0
EEEEM	234.0	4.5	3.7	588.4	5.4	5.3
EEEEE	251.3	4.2	3.4	570.8	5.6	5.4

cal machines EE takes 174.9s, if one of the local machine is replaced by a remote machine EM, the runtime increases by 41.4%. Note that for all programs the runtime increases when adding a remote machine in such a way. Furthermore, a configuration of the form $EMM\ldots M$ is always worst among the configurations with the same number of PEs. This is because the local machine E, which has all the work in the beginning of the execution, has to communicate with the other machines through a high latency network, which becomes a bottleneck in the execution. Finally, configurations of the form $E\ldots E$ or $M\ldots M$ are usually the best configurations, because all machines communicate with others through the low latency network.



a) Homogeneous low latency interconnect

b) Heterogeneous high latency interconnect

FIG. 6.2. *per PE Activity Profile for* `linSolv` *on multi-Clusters*

GRID-GUM shows relatively poor performance on high latency interconnect multi-clusters for many programs. For example, Figure 6.2 shows *GRID-GUM* per-PE activity profile for `linSolv` on a homogeneous low and a heterogeneous high latency interconnect multi-clusters. Figure 6.2.a depicts the performance on homogeneous low latency interconnect cluster. Figure 6.2.b depicts the performance on heterogeneous high latency interconnect where PE 0 & PE 1 and PE 2 & PE 3 are connected pairwise by a low latency network, and with a high latency network between the pairs.

In Figure 6.2.b the PEs exhibit significantly more idle time, i. e. gaps in the horizontal line, and complete at different times. In contrast, the work is fairly evenly balanced in Figure 6.2.a. The idle time in Figure 6.2.b is due to PEs waiting for data to without other threads execute.

TABLE 6.5
High Communication Degree Programs

	raytracer			matMult			linSolv		
	Rtime	Speedup		Rtime	Speedup		Rtime	Speedup	
	Sec.	E	M	Sec.	E	M	Sec.	E	M
M	903.8	1.1	1.0	265.8	0.9	1.0	290.0	1.0	1.0
E	1027.8	1.0	0.8	259.9	1.0	1.0	299.3	1.0	0.9
MM	548.6	1.8	1.4	228.8	1.1	1.1	196.5	1.5	1.4
EM	624.6	1.6	1.4	393.0	0.6	0.6	232.0	1.2	1.2
EE	545.7	1.8	1.6	227.8	1.1	1.1	164.9	1.8	1.7
MMM	383.7	2.6	2.3	133.0	1.9	1.9	139.4	2.1	2.0
EMM	535.8	1.9	1.6	297.7	0.8	0.8	231.8	1.2	1.2
EEM	494.9	2.0	1.8	201.9	1.2	1.3	141.1	2.1	2.0
EEE	387.5	2.6	2.3	137.8	1.8	1.9	136.8	2.1	2.1
MMMM	312.8	3.2	2.8	121.9	2.1	2.1	119.5	2.5	2.4
EMMM	497.6	2.0	1.8	295.0	0.8	0.9	142.5	2.1	2.0
EEMM	421.7	2.4	2.1	213.9	1.2	1.2	134.9	2.2	2.1
EEEM	377.9	2.7	2.3	145.9	1.7	1.8	120.9	2.4	2.3
EEEE	326.4	3.1	2.7	114.8	2.2	2.3	117.1	2.5	2.4
MMMMM	287.8	3.5	3.1	108.6	2.3	2.4	104.4	2.8	2.7
EMMMM	473.8	2.1	1.9	290.8	0.8	0.9	147.0	2.0	1.9
EEMMM	413.7	2.4	2.1	228.8	1.1	1.1	142.1	2.1	2.0
EEEMM	378.7	2.7	2.3	150.9	1.7	1.7	104.9	2.8	2.7
EEEEM	329.9	3.1	2.7	125.1	2.0	2.1	107.7	2.7	2.7
EEEEE	279.8	3.6	3.2	95.9	2.7	2.7	102.9	2.9	2.8

To summarise:

- For high communication degree programs *GRID-GUM* delivers poor performance on high latency multi-clusters (Table 6.5);
- For low communication degree programs *GRID-GUM* can delver good performance on high latency multi-clusters (Table 6.4);
- The poor performance of *GRID-GUM* on high latency multi-clusters is primarily due to poor load management (Figure 6.2).

7. ***GRID-GUM2.*** Based on the results in previous section, it is essential to modify *GRID-GUM* for execution on a computational GRID, (*GRID-GUM2*). *GRID-GUM2* uses the monitored information to provide a good load distribution over the GRID using the following policies:

- An idle PE sends a FISH message only to a PE that has high load relative to its CPU speed.
- PEs have a preference for obtaining work from PEs that currently have low communication latency.
- The recipient PE switches from passive to active load distribution if a FISH message received from another cluster.

The new *GRID-GUM2* mechanism has two main components: information collection and adaptive load distribution. The information collection is supported by a monitoring mechanism to provide the current state information of the GRID network. The monitoring mechanism performs during the whole course of execution. It collects static information like CPU speed at the start of program in *PEStatic* table (Figure 7.1), and dynamic information such as load and latency during the execution in *PEDynamic* and *ComMap* tables (Figures 7.2 and 7.3) respectively. The adaptive load distribution of *GRID-GUM2* comprises the following aspects:

- Resource-level load distribution: programs executed do not required specific resource, present on only same PEs. Idle PEs use load distribution mechanism in*GRID-GUM2* to seek work from PEs relatively

PE	CPU Speed	Time Stamp
A	550 MHz	13:40:01
D	550 MHz	13:45:00
C	350 MHz	12:40:03
B	350 MHz	13:44:03
F	350 MHz	14:40:03

FIG. 7.1. *PEStatic Table*

PE	Load	time_stamp
A	2000	14:13:49
D	3000	14:13:59
B	10000	14:12:22

FIG. 7.2. *PEDynamic Table*

PE	Latency	Last Update
F	0.75 msec	12:45:20
G	2.05 msec	12:24:50
C	10.00 msec	12:50:25

FIG. 7.3. *ComMap Table*

```
IF received fish THEN
 update tables with data
  from fishing PE
 IF sparks availabelTHEN
  IF fishing PE is local THEN
   send sparks in sechedule
    to fishing PE+local data
  ELSE
   send spark(s) in super−schedule
    to fishing PE+local data
 ELSE
  IF another PE has spark
   forward fish+local data
    to busiest local PE
```

FIG. 7.4. *Work Request*

```
IF idle THEN
 send fish+local data
  to busiest PE from tables
 IF runnable−thread THEN
  execute runnable−thread
 IF spark in the spark−pool THEN
  create runnable−thread
  execute runnable−thread
 ELSE
  send fish+local data
   to busiest PE from tables
```

FIG. 7.5. *Work Location*

The Load Distribution Mechanisms in GRID-GUM2

heavily loaded.

- Dependent load distribution: *GRID-GUM2* aims for an efficient load distribution mechanism to a single parallel program with dependent tasks.
- Decentralised information services: *GRID-GUM2* maintains a decentralised scheme where every PE is responsible for maintaining state information of some nearby PEs and share it with other PEs.
- Dynamic load distribution: *GRID-GUM2* assumes that limited knowledge about the load and PEs are available *a priori*, and load distribution decisions have to be made during the execution.
- Decentralised load distribution organisation: *GRID-GUM2* distributes the load distribution decision to every PE. Therefore, each PE acts as both a load distributer and a computational resource.
- Redistribution support: *GRID-GUM2* supports work placement which enhance system reliability and flexibility.
- Adaptive load distribution: *GRID-GUM2* is a mainly passive load distribution system where lightly loaded PEs have to explicitly ask for work from PEs with excess load. However, if an idle PE requests work from a PE residing outside its cluster and the request originated from relatively powerful cluster, it changes from a passive to an active system and the recipient PE sends more work to the idle PE.

The core of *GRID-GUM2* load distribution can be summarised as work location (Figure 7.4), and work request handling (Figure 7.5).

8. *GRID-GUM2* Performance on Heterogeneous Architecture. This experiment investigates the performance impact of using the adaptive load distribution of *GRID-GUM2* on multiple heterogeneous clusters with moderate latency interconnect.

The measurements in Table 8.1 use *GRID-GUM* and *GRID-GUM2* on Edin1 and Edin2 Beowulf clusters described in Table 5.1. Four GpH programs are measured in this experiment: `queens`, `sumEuler`, `linSolv` and `raytracer` described in Section 5. In Table 8.1, The second and third columns record the run-time using *GRID-GUM* and *GRID-GUM2* in seconds respectively. The last column shows the percentage improvement of

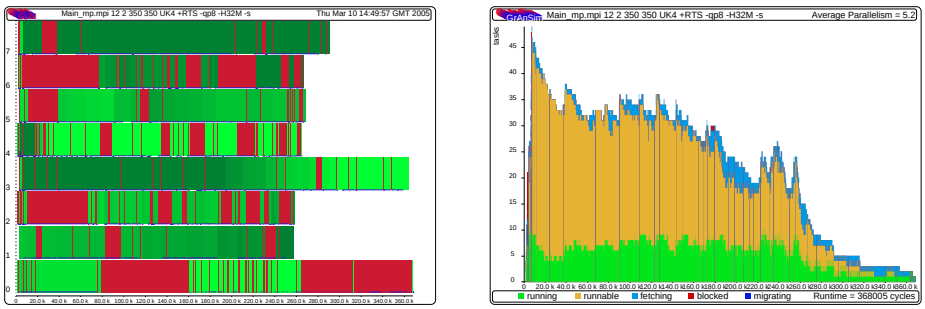
GRID-GUM2.

TABLE 8.1
Performance On Heterogeneous Architecture

Program	Run-time (s)		Improvement %
	GRID-GUM	*GRID-GUM2*	
`queens`	668	310	53%
`sumEuler`	570	279	51%
`linSolv`	217	180	17%
`raytracer`	1340	572	57%
Min			17%
Max			57%
Geometric Mean			47.3%

Table 8.1 shows that, *GRID-GUM2* outperforms *GRID-GUM* on multiple heterogeneous clusters with moderate latency interconnect as far as the execution time is concerned. *GRID-GUM2* shows run-time improvements between 17% and 57%. The greatest improvement are given with the most dynamic program, `raytracer`. Through the rest of this sub-section we consider studying in more details the behaviour of `raytracer` in multi-clusters heterogeneous architecture.

`raytracer` has highly irregular execution, and consequently is very sensitive to changes in parallel environment. Figure 8.1 shows per-PE and overall activity profiles for `raytracer`, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7). A per-PE activity profile shows the behaviour for each of the PEs (y-axis) over execution time (x-axis). An overall activity profile shows the behaviour of the program at each instant of its execution.



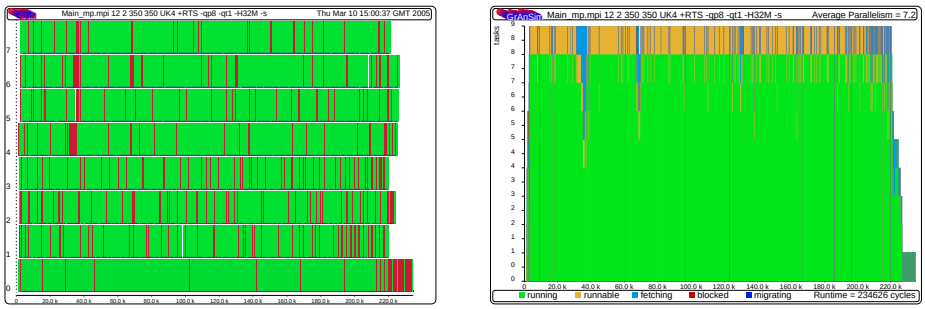
a) Per-PE activity profile *b*) Overall activity profile

FIG. 8.1. *GRID-GUM:* `raytracer` *with* 350*X*350 *Image on a Heterogeneous multi-Clusters*

Figure 8.1.a shows a poor load distribution of *GRID-GUM* with `raytracer` to calculate an image with resolution 350×350 using eight heterogeneous machines, i. e. four fast and four slow machines. PEs as depicted in Figure 8.1 have numerous idle period and finish at different time. From Figure 8.1.b, it is observable that there are a considerable number of runnable threads waiting to be evaluated at most of the execution time. This may explain the poor load distribution in *GRID-GUM*. PEs with slow CPU speed in a heterogeneous architecture in *GRID-GUM* show the same demand of seeking work as PEs with fast CPU speed. This concludes that PEs with slow CPU speed accumulate and activate sparks as PEs with fast CPU speed. If a spark has been activated, it remains in its local PE as runnable or blocked thread in the thread pool and it can not be evaluated by another PE. Considering that PEs have different capabilities of evaluating their own threads explains the reason that there are many runnable threads are waiting to be evaluated while there are some PEs are idle.

GRID-GUM provides explicit control over the load distribution by specifying a hard limit on the total number of live threads, i. e. runnable or blocked threads. Figure 8.2 shows per-PE and overall activity profiles

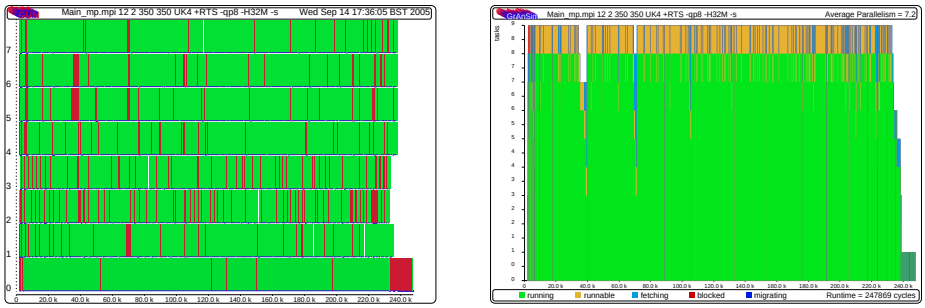
for `raytracer` to calculate an image with resolution 350×350, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7). *GRID-GUM* in this experiment uses a hard limit of 1 on the total number of live threads in the thread pool.



a) Per-PE activity profile *b*) Overall activity profile

FIG. 8.2. *GRID-GUM with Thread Limitation:* `raytracer` *with* 350*X*350 *Image on a Heterogeneous multi-Clusters*

In Figure 8.2, *GRID-GUM* with thread limitation shows an efficient load distribution in a heterogeneous architecture with moderate latency interconnect. it completes the image manipulation in 327 s, while the version of *GRID-GUM* does not employ thread limitation requires 441 s. Expectedly, for the same problem *GRID-GUM*2 has similar performance, i. e. 338 s, with *GRID-GUM* using thread limitation (Figure 8.3).



a) Per-PE activity profile *b*) Overall activity profile

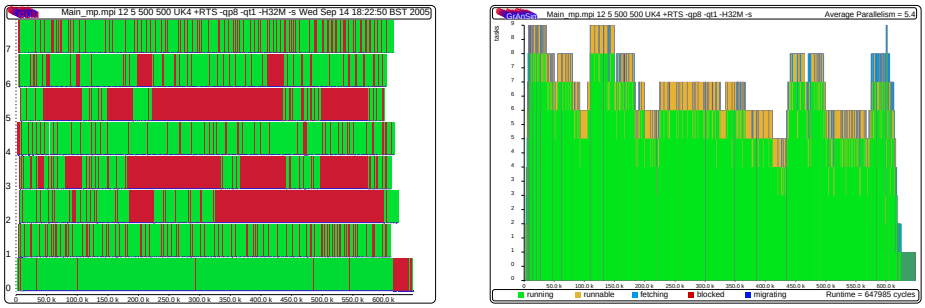
FIG. 8.3. *GRID-GUM2:* `raytracer` *with* 350*X*350 *Image on a Heterogeneous multi-Clusters*

However, *GRID-GUM*'s load distribution efficiency regress when the size of the input increased even with thread limitation. Figure 8.4 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution 500×500, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7). *GRID-GUM* in this experiment uses a hard limit of 1 on the total number of live threads in the thread pool.

PEs in Figure 8.4.a finish at the same time, but they still have numerous idle periods which deteriorate the performance. This idle periods are caused by the dependencies between threads in `raytracer`. These dependencies are effected badly by the thread limitation, which causes PEs to remain idle waiting for certain threads to be evaluated. Figure 8.4.b shows that the idle periods are not caused by lack of tasks to be evaluated. Generally speaking, thread limitation has a serious impinge on many programs performance. Figure 8.5 shows per-PE profiles for `linSolv` with and without thread limitation on 8 homogeneous machines from Edin1 Beowulf cluster.

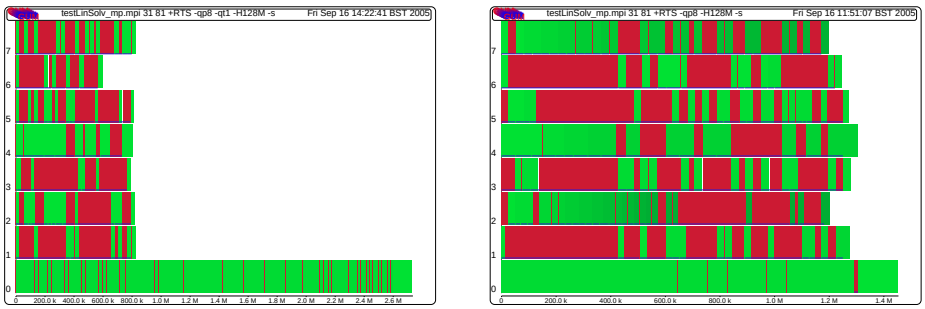
From Figure 8.5, *GRID-GUM* delivers better performance with `linSolv` without using thread limitation. *GRID-GUM* requires 2802 s to finish `linSolv` computation using thread limitation, unlike when thread limitation is excluded *GRID-GUM* requires only 1521 s to finish the same computation in the same platform.

However, *GRID-GUM*2 shows more effective load distribution in heterogeneous architecture in comparison with *GRID-GUM*'s load distribution. Figure 8.6 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution 500×500, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7).



a) Per-PE activity profile *b*) Overall activity profile

FIG. 8.4. *GRID-GUM with Thread Limitation:* `raytracer` *with* $500X500$ *Image on a Heterogeneous multi-Clusters*



a) With thread limitation *b*) With no thread limitation

FIG. 8.5. *GRID-GUM:* `linSolv` *on a Heterogeneous multi-Clusters*

PEs in Figure 8.6.a are fairly balanced and and finish at about the same time. Figure 8.6.b shows that *GRID-GUM*2 scores a good average parallelism in 8 PEs, 6.9, and generates enough tasks for all PEs at each instant of the execution time. Finally, *GRID-GUM*2 outperforms *GRID-GUM* with thread limitation in `raytracer` when the image resolution increases from 350×350 to 500×500, the execution time for *GRID-GUM*2 and *GRID-GUM* with thread limitation is 572 s and 814 s, respectively.

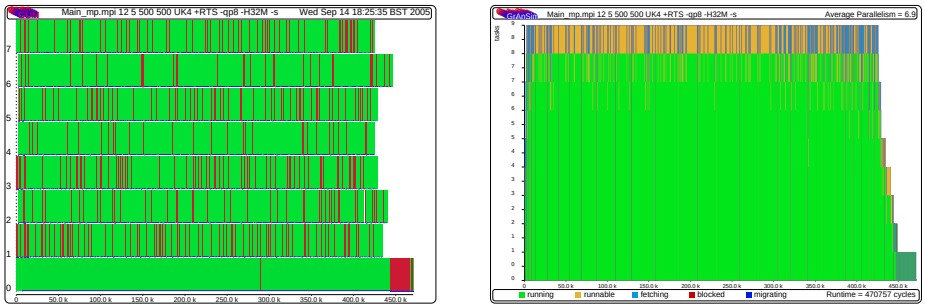
To summarise:

- *GRID-GUM*2 shows efficiency and automatic management of data and work on heterogeneous multi-clusters GRID environment (Table 8.1);
- For some programs, thread limitation improves the performance of *GRID-GUM* on heterogeneous multi-clusters but not for all (Figures 8.2 and 8.5);
- *GRID-GUM*2 outperforms *GRID-GUM* and *GRID-GUM* with thread limitation for large input sizes (Figures 8.6 and 8.4).

9. Related Work. The most closely related to our philosophy of semi-implicit management of parallelism in a high level language is the ConCert system [10] system and the Hemlock compiler [11], which translates a subset of ML to machine code, for execution on a GRID architecture. In contrast to our work, parallelism is expressed via explicit synchronisation.

Under the topic of meta-computing several projects, like Harness [12], aim at provide functionality similar to *GRID-GUM*2. The characteristic difference to *GRID-GUM*2 is the automatic management of parallelism within one parallel program.

Alt *et al* apply skeletons to computational GRIDs [13]. This work focuses on providing the application user with skeletons to capture common patterns of GRID abstractions. However, our aim is to provide more general programming language support for parallelism through an implementation that incorporates new implicit dynamic coordination-management strategies. Aldinucci *et al* also apply skeletons to computational GRIDs [14].This work focuses on providing a skeleton to centralise load management in the GRID environment. However, our aim is to solve load scheduling on the GRID by developing a dynamic decentralised load schedule.



a) Per-PE activity profile *b*) Overall activity profile

FIG. 8.6. *GRID-GUM2:* `raytracer` *with* 500×500 *Image on a Heterogeneous multi-Clusters*

10. Conclusion. We have presented and measured two GRID-enabled runtime environments for the GPH high-level parallel programming language.

Measurements of *GRID-GUM* showed that for large programs, the performance of GUM on a single cluster is largely independent of the communication library used. Despite being designed for homogeneous clusters, *GRID-GUM* delivers good and predictable speedups on GRID multi-clusters with a low latency interconnect. In contrast, on GRID multi-clusters with heterogeneous architecture, *GRID-GUM* does not deliver good performance due to poor scheduling. In addition, on GRID multi-clusters with a high latency interconnect, Grid-GUM only delivers acceptable speedups for low communication degree programs.

We have presented the initial design of *GRID-GUM2* that incorporates new load management mechanisms, informed by the *GRID-GUM* results. *GRID-GUM2* achieves good parallel performance for a typical set of symbolic applications running on two heterogeneous clusters connected via the Globus Toolkit, realising a small but typical computational GRID. The improved performance is achieved by dynamically distributing work between the machines on top of a virtual shared memory implementation. No explicit thread placement or scheduling has to be done by the programmer. In particular, our system makes contributions towards load distribution on such wide-area networks

We conclude that, with appropriate load management strategies, acceptable performance can be obtained on hereogeneous computational GRIDs from a distributed virtual shared heap implementation of a high-level parallel language.

REFERENCES

[1] H-W. LOIDL, F. RUBIO, N. SCAIFE, K. HAMMOND, S. HORIGUCHI, U. KLUSIK, R. LOOGEN, G. J. MICHAELSON, R. PEÑA, Á. J. REBÓN PORTILLO, S. PRIEBE AND P. W. TRINDER, *Comparing Parallel Functional Languages: Programming and Performance*, in Higher-order and Symbolic Computation, Kluwer Academic Publishers, 16(3),2003.

[2] GHC,*The Glasgow Haskell Compiler*, Department of Computing Science, University of Glasgow (`http://www.dcs.gla.ac.uk/`), January 1998, "The Glasgow Haskell Compiler compiles code written in the functional programming language Haskell" URL: `http://www.dcs.gla.ac.uk/fp/software/ghc/`

[3] P. W. TRINDER, K. HAMMOND, J. S. MATTSON JR., A. S. PARTRIDGE AND S. L. PEYTON JONES, *GUM: a Portable Parallel Implementation of Haskell*, in PLDI'96—Conf. on Programming Language Design and Implementation, 1996, Philadephia USA.

[4] T. L. CASAVANT AND J. G. KUHL, *A Taxonomy of Scheduling in General-Purpose Distribution Computing Systems*, in IEEE Transactions on Software Engineering, 14(2),1988, ISSN 0098-5589, pages 141–154, IEEE Press, Piscataway NJ USA.

[5] Y-T. WANG, AND R. J. T. MORRIS, *Load Sharing in Distributed Systems* , In Scheduling and Load Balancing in Parallel and Distributed Systems, 1995, Shirazi, A. and Hurson, A. R. and Kavi, K. M., eds, IEEE Transactions on Software Engineering, pp. 7–20, ACM.

[6] D. L. EAGER, E. D. LAZOWSKA AND J. ZAHORJAN, *A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract)*, in SIGMETRICS '85: Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems, 1985, ISBN 0-89791-169-5, pp. 1–3, Austin Texas United States, ACM Press.

[7] S. L.PEYTON JONES, C. CLACK, J. SALKILD AND M. HARDIE, *GRIP—a High-Performance Architecture for Parallel Graph Reduction*, in Intl. Conf. on Functional Programming Languages and Computer Architecture, pp. 98–112, September 1987, LNCS 274, Portland Oregon, Springer-Verlag.

[8] I. FOSTER, AND C. KESSELMAN, *Globus: A Metacomputing Infrastructure Toolkit*, in "The International Journal of Supercomputer Applications and High Performance Computing", 11(2), pp. 115–128, 1997.

[9] A. Al Zain, P. Trinder, H-W. Loidl and G. Michaelson, *Grid-GUM: Towards Grid-Enabled Haskell*, in Draft Proceedings of IFL'04—Intl. Workshop on the Implementation of Functional Languages, Septamber 2004, Lübeck Germany.
[10] Trustless Grid Computing in ConCert, *B-Y. Evan Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning*, In Proceedings of the GRID 2002 Workshop, 2536 of LNCS, Springer-Verlag, 2001.
[11] T. Murphy VII, *Hemlock and Concert v2 Framework*, Talk at Carnegie Mellon University, August 2003
[12] M. Beck, J. Dongarra, G. Fagg, A. Geist, P. Gray, M. Kohl, J. Migliardi, K. Moore, T. Moore P. Papadopoulos, S. Scott and V. Sunderam *HARNESS: A Next Generation Distributed Virtual Machine*, in Future Generation Computer Systems, 15(5/6):571-582, October 1991, Special Issue in Metacomputing.
[13] M. Alt, H. Bischof and S. Gorlatch, *Program Development for Computational Grids Using Skeletons and Performance Prediction*, in CMPP'02—Int. Workshop on Constructive Methods for Parallel Programming, June 2002.
[14] M. Aldinucci, M. Dnelutto and Dünnweber, *Optimization Techniques for Implementing Parallel Sckeletons in Grid Environments*, in CMPP'04—Intl. Workshop on Constructive Methods for Parallel Programming, July 2004, Stirling Scotland
[15] M. Litzkow, M. Livny and M. Mutka, *Condor- A Hunter of Idle Workstations*, in Proc. the 8^th InternationalConference of Distributed Computing Systems, San Jose, California, June 1988.
[16] J. Frey and T. Tannenbaum and M. Livny and I. Foster and S. Tuecke, *Condor-G: A Computation Management Agent for Multi-Institutional Grids*,in HPDC10 — Tenth International Symposium on High Performance Distributed Computing, August, 2001, IEEE Press.
[17] F. Berman and R. Wolski, *The AppLeS Project: A Status Report*, 1997.
[18] A. S. Grimshaw, M. J. Lewis, A. J. Ferrari and J. F. Karpovich, *Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems*, Department of Computer Science, University of Virginia, 1998, Technical Report, CS-98–12.
[19] F. Berman, G. Fox and T. Hey, *The Grid: past, present, future*, in Grid Computing—Making the Global Infrastructure a Reality, pp. 9–50, John Wiley & Sons, Ltd, West Sussex, England, 2003.
[20] I. Foster and C. Kesselman, *The Globus project: a status report*, in Future Generation Computer Systems, 15(5–6), pp. 607–621, 1999.
[21] A. Geist, A. Beguelin, J. Dongerra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine*, MIT, 1994.
[22] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT, second ed. , 1999
[23] W. Gropp, E. Lusk, N. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI Message-Passing Interface standard*, in Parallel Computing, 1996, 22(6), pp. 789–828.
[24] G. A. Geist, J. A. Kohl and P. M. Papadopoulos, *PVM and MPI: A comparison of features*, in Calculateurs Parallels, 8(2), 1996.
[25] N. Karonis, B. Toonen and I. Foster, *MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface.*, in Journal of Parallel and Distributed Computing, 2003.
[26] I. Foster, C. Kesselman and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, in Int. J. Supercomputer Applications, 2001.
[27] S. Zhou,X. Zheng, J. Wang and P. Delisle, *Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, in Software—Practise and Experience, 23(12), pp. 1305–1336, 1993.
[28] Sun Microsystems, *Grid-Engine Project*, 2001, `http://gridengine.sunsource.net/`
[29] P. W. Trinder, K. Hammond, H-W. Loidl and S. L. Peyton Jones, *Algorithm + Strategy = Parallelism*, in Journal of Functional Programming, 1998, 8(1), pp. 23–60.

Appendix A: `sumEuler`.

```
module Main(main) where

import System(getArgs)
import Strategies

sumTotient :: Int ->---lower limit of the interval
              Int ->---upper limit of the interval
              Int ->---chunk size
              Int
sumTotient lower upper c =
  sum ( map (sum . map euler) (splitAtN c [upper, upper-1 .. lower])
       `using` parList rnf)

euler :: Int -> Int
euler n = length (filter (relprime n) [1 .. n-1])


relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

mkList :: Int -> Int -> [Int]
mkList lower upper = reverse (enumFromTo lower upper)

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
                where (ys,zs) = splitAt n xs
main = do args <- getArgs
          let
            lower = read (args!!0) :: Int---lower limit of the interval
            upper = read (args!!1) :: Int---upper limit of the interval
            c     = read (args!!2) :: Int---chunksize
          putStrLn ("Sum of Totients between [" ++
                    (show lower) ++ ``.." ++ (show upper) ++ ``] is `` ++
                     show (sumTotient lower upper c))
```

Edited by: Frédéric Loulergue
Received: October 20, 2005
Accepted: February 1st, 2006

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 27–37. http://www.scpe.org
ISSN 1895-1767

DYNAMIC MEMORY MANAGEMENT IN THE *LOCI* FRAMEWORK

YANG ZHANG AND EDWARD A. LUKE*

Abstract. Resource management is a critical concern in high-performance computing software. While management of processing resources to increase performance is the most critical, efficient management of memory resources plays an important role in solving large problems. This paper presents a dynamic memory management scheme for a declarative high-performance data-parallel programming system—the *Loci* framework. In such systems, some sort of automatic resource management is a requirement. We present an automatic memory management scheme that provides good compromise between memory utilization and speed. In addition to basic memory management, we also develop methods that take advantages of the cache memory subsystem and explore balances between memory utilization and parallel communication costs.

Key words. Memory management, declarative languages, parallel programming, software synthesis

1. Introduction. In this paper we discuss the design and implementation of a dynamic memory management strategy for the declarative programming framework, *Loci* [5, 6]. The *Loci* framework provides a rule-based programming model for numerical and scientific simulation similar to the *Datalog* [12] logic programming model for relational databases. In *Loci*, the arrays typically found in scientific applications are treated as relations, and computations are treated as transformation rules. The framework provides a planner, similar to the FFTW [3] library, that generates a schedule of subroutine calls that will obtain a particular user specified goal. *Loci* provides a range of automatic resource management facilities such as automatic parallel scheduling for distributed memory architectures and automatic load balancing. The *Loci* framework has demonstrated predictable performance behavior and efficient utilization of large scale distributed memory architectures on problems of significant complexity with multiple disciplines involved [6]. *Loci* and its applications are in active and routine use by engineers at various NASA centers in the support of rocket system design and testing.

The *Loci* planner is divided into several major stages. The first stage is a dependency analysis which generates a dependency graph that describes a partial ordering of computations from the initial facts to the requested goal. In the second stage, the dependency graph is sub-divided into functional groups that are further partitioned into a collection of directed acyclic graphs (DAGs). In the third stage, the partitioned graphs are decorated with resource management constraints (such as memory management constraints). In the forth stage a proto-plan is formed by determining an ordering of DAG vertices to form computation super-steps. (In the final parallel schedule, these steps are similar to the super-steps of the Bulk Synchronous Parallel (BSP) model [13, 10, 2].) The proto-plan is used to perform analysis on the generation of relations by rules as well as the communication schedule to be performed at the end of each computation step in the fifth and sixth stages (existential analysis and pruning), as described in more detail in this recent article [6]. Finally the information collected in these stages is used to generate an execution plan in the seventh stage. Dynamic memory management is primarily implemented as modifications to the third and fourth stages of *Loci* planning.

2. Related Work. The memory system and its management has been studied extensively in the past. These studies are on various different levels. On the software level, memory management can be roughly categorized into allocation techniques and management strategies. Allocation techniques mostly deal with how memory is requested and returned to the operating system in order to efficiently satisfy application requests. Memory management strategies often study how and when to recycle useless memory. Allocation is usually performed by the "allocator," which is typically implemented as a library component (such as the `malloc` routine in the standard C library). The central themes in various allocation techniques are fragmentation and locality problems. If care is not taken, then the allocator could build up large internal fragmentation with significant inaccessible memory. The locality property in the allocator can greatly affect the cache and page misses and hence also contributes to the program performance. Wilson et al. [15] has an excellent survey for various allocation techniques. The memory management strategies can be subdivided mainly into two directions: one is to managing memory manually; while the other direction is to automatically reclaim useless memory. Manual memory management is usually performed by explicit programming. Programmer has full control over memory recycling. There has been much debate concerning various aspects of the advantages and disadvantages for

*Department of Computer Science and Engineering, and Computational Simulation and Design Center, Mississippi State University, Mississippi State, MS 39762. Questions, comments, or corrections may be directed to the first author at fz15@cse.msstate.edu

manual memory recycling. But a general consensus is that for large complex systems, a manual strategy is not encouraged due to its complex interaction with other software components. Automatic memory management frees the programmers from bookkeeping details of reclaiming memory and is typically a built-in feature in many modern languages such as *Java*, *ML*, *Smalltalk*, etc. The most prevalent technique for automatic memory recycling is "garbage collection" where the run-time system periodically reclaims useless memory [14]. Recent studies proposed "region inference" as another technique for automatic memory recycling. Region inference [11] relies on static program analysis and is a compile-time method and uses the region concept. The compiler analyzes the source program and infers the allocation. In addition to being fully automatic, it also has the advantage of reducing the run-time overhead found in garbage collection.

When designing the memory management subsystem for *Loci*, we are mostly interested in designing a memory management strategy and not in low level allocator designs. The programming model in *Loci* is declarative, which means the user does not have direct control of allocation. Also one major goal of the *Loci* framework is to hide irrelevant details from the user. Therefore we are interested in designing an automatic memory management scheme. Garbage collection typically works better for small allocations in a dynamic environment. While in *Loci*, the data-structures are often static; and allocations are typically large. Thus, the applicability of garbage collection to this domain is uncertain. Another problem of garbage collection is that the time required for collecting garbage cannot be predicted easily. While there has been research work on real-time garbage collection [4, 9] that attempt to address such issues, we find a memory management scheme without garbage collection to be straightforward and easy to reason about. Therefore instead of applying traditional garbage collection techniques, we have adopted a strategy that shares some similarities to the region inference techniques as will be described in the following sections. We also note interactions between the parallel scheduling of tasks and memory management strategies. Similar interactions have been observed in recent studies in continuous data streams [1] where it is demonstrated that operator scheduling order in the context of continuous data streams can affect overall memory requirements. They suggested a near-optimal strategy in the context of stream models. Although this is in a context that differs from our data-parallel programming domain, it shares some similarities with our approaches in balancing the memory utilization and parallel communication costs within the *Loci* framework.

3. Basic Dynamic Memory Management. In *Loci*, the aggregations of attributes found in scientific computing are treated as binary relations and are stored in *Loci* provided value containers. These value containers are the major source of memory consumption. Therefore the management of allocation and deallocation of these containers is the major focus of our memory management scheme. A simple way to manage the lifetime of these containers is *preallocation*. In this approach we take advantage of the *Loci* planner's ability to predict the sizes of the containers in advance. In the preallocation scheme, all containers are allocated at the beginning and recycled only at the end of the schedule. While this scheme is simple and has little run-time overhead, it does not offer any benefits for saving space. Scientific applications for which *Loci* is targeted tend to have large memory requirements. The primary goal of the management is therefore to reduce the peak memory requirement so that larger problems can be solved on the same system. Preallocation obviously fails this purpose.

After the user submits a request, the *Loci* planner generates a dependency graph. This graph describes the relationship between rules that obtains the specified goal. The dependency graph usually contains cycles caused by the specification of iteration, conditional execution blocks, and rule recursion. To simplify scheduling, the dependency graph is partition to a hierarchical graph where each level contains a DAG. Cycles have been removed in this graph and replaced by super-nodes that represent the semantics (e.g. iteration or recursion). This hierarchical graph is referred to as the multi-level graph. Thus, most of *Loci* scheduling is reduced to scheduling a DAG of rules. A simple approach to incorporating appropriate memory scheduling would be to incorporate relevant memory management operations into the multi-level graph. Then, when the graph is compiled, proper memory management instructions are included into the schedule and will be invoked in execution. We refer this process of including memory management instructions into the dependency graph as graph decoration. Thus memory management for *Loci* becomes the graph decoration problem. For example, Fig. 3.1 shows a decoration for a simple DAG. However, the multi-level dependency graph for a real application is likely to be complex. For example, multiple nested iterations and conditional specifications, recursions, etc. could also be involved. A global analysis of the graph is performed to determine the lifetime of all containers in the final schedule [16].

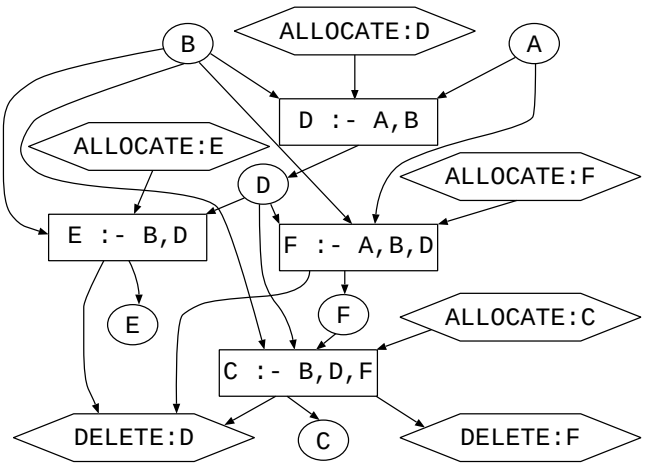


FIG. 3.1. *Memory Management by means of Graph Decoration*

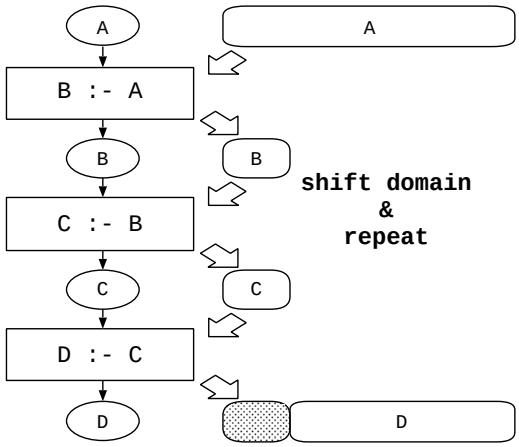


FIG. 4.1. *The Chomping Idea*

4. Chomping. Chomping is a technique we used in *Loci* to optimize the cache performance. The idea of chomping is borrowed from the commonly known loop scheduling technique: strip mining. In *Loci*, relations, the primary data abstractions, are collections of attributes that are stored in array-like containers that represent aggregations of values. Since these containers dominate the space consumed by *Loci* applications, they are ideal candidates for cache optimization by data partitioning. Data partitioning also creates further chance for memory savings in addition to the basic memory management implemented in *Loci*. Consider the rule chain in Fig. 4.1. Relation A is the source to the chain and D is the final derived relation; B and C are intermediate relations. We can break the rules in the chain into small sub-computations. In each of these sub-computation, only part of the derived relations are produced. This implies for any intermediate relations, only partial allocation of their container is required. Because these partial allocations can be made small, they enhance cache utilization and can further reduce memory requirements. Breaking computations into smaller intermediate segments not only reduces absolute memory allocation requirements, but also helps to reduce fragmentation by reusing a pool of small uniformly sized memory segments.

4.1. Chomping Implementation. The implementation of chomping extends the partitioning second stage in the *Loci* planner. In each level in the multi-level dependency graph generated by the *Loci* planner, all suitable rule chains for chomping are first identified. Then each chain is replaced by a special chomping rule and is handled separately in the graph compilation phase. The replacement is illustrated in Fig. 4.2. This allows smooth integration of chomping and dynamic memory management implemented in the *Loci* planner. As we presented in section 3, memory management in the *Loci* planner is implemented as a graph decoration

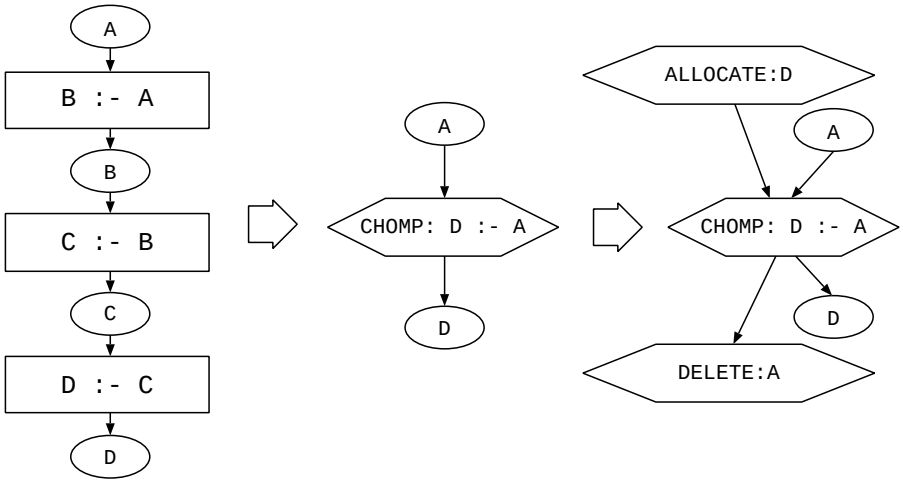


FIG. 4.2. *Implementation of Chomping*

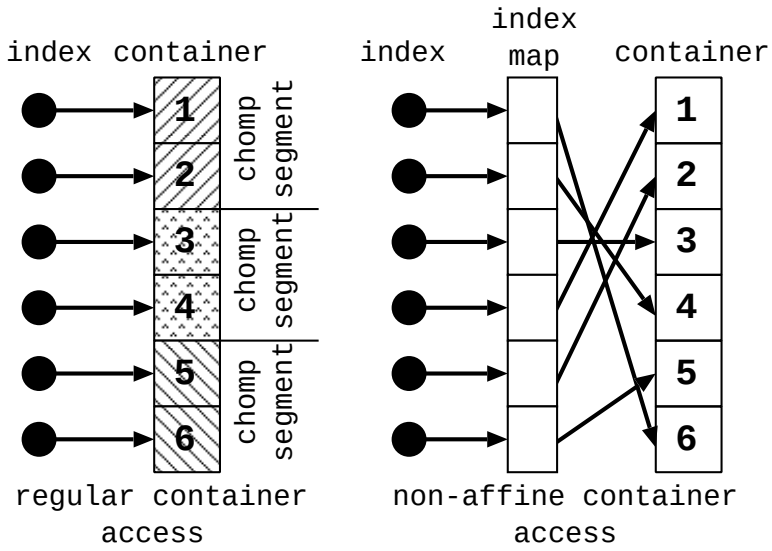


FIG. 4.3. *Memory Reference Patterns*

problem. The graph decorator does not have to know the chomping chain. For example, for the replaced graph in Fig. 4.2, the decorator can proceed as normal without the knowledge of the chomping chain. The memory allocation of the chomped relations B and C are handled internally in the chomping rule.

Obviously, the central problem in the implementation is how we identify suitable rule chains that can be chomped in a DAG in the multi-level dependency graph. Because of the existence of non-affine memory references in *Loci* rules, we cannot group arbitrary rules into rule chains that can be chomped. Consider the examples of memory access shown in Fig. 4.3. For a regular container access, there is a pre-determined access order to the domain of the container. For the case shown in Fig. 4.3, the access order to the domain is $[1, 2, 3, 4, 5, 6]$. Since this pattern is pre-determined, we can allocate memory for domain $[1, 2]$ first and perform the computation on this sub-domain of the container. Then we can shift this sub-domain by a distance of 2 and obtain the following sub-domain $[3, 4]$, and finally $[5, 6]$. Therefore the container and the rules associated can be chomped directly. For a non-affine container access, essentially we have the pattern of $A[B[i]]$. A is the container and B is the index map. The index map is often unknown until run-time. Therefore the access pattern to the container is also unknown until run-time. For the case shown in Fig. 4.3, the access order to the domain is $[6, 4, 3, 1, 2, 5]$. We have no direct way to allocate and shift sub-domains as in the regular container access. As

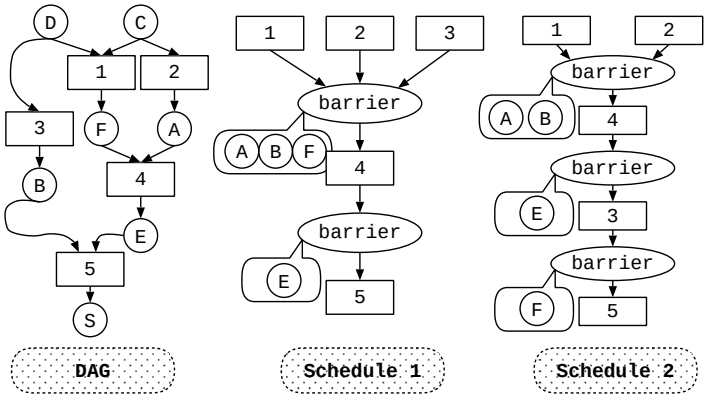


FIG. 5.1. *Different Scheduling for a DAG*

a result, the container and its associated rules cannot be chomped directly. In *Loci*, we use a heuristic search to identify suitable chains in the multi-level dependency graph and apply chomping only to them. The running time and the searching results of our algorithm are of satisfactory for large *Loci* applications. In section 6, we include an empirical evaluation of this heuristic chomping searching algorithm.

4.2. The Decision of Chomping Size. The total allocation size for chomped relations in a chomping rule is referred to as chomping size in the *Loci* planner. For example, the chomping size for the chomping chain in Fig. 4.1 is the total memory allocation for segments for relations B and C. Ideally, as in matrix blocking algorithms, the chomping size should be approximately the size of the data cache in order to utilize the cache performance. However, in *Loci*, we typically set it to be approximately half the size of the data cache. Chomping in *Loci* does not completely resemble typical cache optimization techniques such as matrix blocking algorithms. The source and target relations for a chomping rule chain is not chomped. They could be large and rules in the middle of the chomping chain could also have access (or non-affine access in the worst case) to the source relations. These potentially destroy the cache benefits obtained through chomping. We set the chomping size smaller than the data cache size with the hope to alleviate some of these problems. In our implementation, users can also specify a particular chomping size for the *Loci* planner. Chomping may create some further chances for program optimization, we will touch some of these in our conclusion section.

5. Memory Utilization and Parallel Communication Costs. In section 3, we transformed the memory management into a graph decoration problem. However the graph decoration only specifies a dependencies between memory management and computation. It is up to the *Loci* planner to generate a particular execution order that satisfies this dependence relationship. From the memory management point of view, the order to schedule allocation and deallocation affects the peak memory requirement of the application. On the other hand, the *Loci* planner can produce a data-parallel schedule. In the data-parallel model, after each super-step, processors need to synchronize data among the processes. From the communication point of view, different schedules may create different numbers of synchronization points. While the number of synchronization points does not change the total volume of data communicated, increased synchronization does reduce the opportunity to combine communication schedules to reduce start-up costs and latency. Thus with respect to parallel overhead, less synchronization is preferred.

Figure 5.1 shows the effect of different scheduling of a DAG. Schedule one is greedy on computation, a rule is scheduled as early as possible. Therefore schedule one has fewer synchronization points. Schedule two is greedy on memory, a rule is scheduled as late as possible. Therefore derived relations are spread over more super-steps, hence more synchronization points are needed.

A trade-off therefore exists in the *Loci* planner. In order to optimize memory utilization and reduce peak memory requirement, the planner will typically generate a schedule with more synchronization points, and therefore increase the communication start-up costs and slow down the execution. Attempting to minimize the synchronization points in a schedule results in a fast execution, but with more memory usage. Such trade-off can be customized under different circumstances. For example, if memory is the limiting factor, then a memory optimization schedule is preferred. In this case, speed is sacrificed for getting the program run within limited

resources. On the other hand, if time is the major issue, then a computation greedy schedule is preferred, but users have to supply more memory to obtain speed. In the *Loci* planner, we have implemented two different scheduling algorithms. One is a simple computation greedy scheduling algorithm, which minimizes the total synchronization points. The other one is a memory greedy scheduling algorithm. It relies on heuristics to attempt to minimize the memory usage. Users of *Loci* can instruct the planner to choose either of the two policies.

The scheduling infrastructure in the *Loci* planner is priority based. *Loci* planner schedules a DAG according to the weight of each vertex. In this sense, scheduling policies can be implemented by providing different weights to the vertices. For a computation greedy schedule, we simply set the same weight for each vertex in the graph, since vertices with same weight will be scheduled together according to the graph topology. This effectively schedules all possible rules together to form a super-step and hence minimizes the barrier points needed to synchronize intermediate results among processes.

```
PrioGraph(gr)
 1  l ← NIL
 2  for vi ∈ V
 3       do a ← AllocNum(vi)
 4          d ← DelNum(vi)
 5          o ← TargetOutEdgeNum(vi)
 6          l ← Append(l, (vi, a, d, o))
 7  prio ← 0
 8  for i ← 1 to Length(l)
 9       do s ← l[i]
10          if s.a = 0
11             then p[s.vi] ← prio
12                  Erase(l, l[i])
```

```
 1  prio ← 1
 2  Sort(l, Ascend(a))
 3  StableSort(l, Descend(d))
 4  for i ← 1 to Length(l)
 5       do s ← l[i]
 6          if s.d ≠ 0
 7             then p[s.vi] ← prio
 8                  Erase(l, l[i])
 9                  prio ← prio +1
10  Sort(l, Ascend(o))
11  for i ← 1 to Length(l)
12       do s ← l[i]
13          p[s.vi] ← prio
14          prio ← prio +1
```

We also provide a heuristic for assigning vertices weight that attempts to minimize the memory utilization for the schedule. The central idea of the heuristic is to keep low memory usage in each scheduling step. Given a DAG with memory management decoration, rules that do not cause memory allocation have the highest priority and are scheduled first. They are packed into a single step in the schedule. If no such rules can be scheduled, then we must schedule rules that cause allocation. The remaining rules are categorized. For any rule that causes allocation, it is possible that it also causes memory deallocation. We schedule one such rule that causes most deallocations. If multiple rules have the same number of deallocations, we schedule one that causes fewest allocations. Finally, we schedule all rules that do not meet the previous tests, one at a time with the fewest outgoing edges from all relations that it produces. This is based on the assumption that the more outgoing edges a relation has in a DAG, the more places will it be consumed, hence the relation will have a longer lifetime.

We used a sorting based algorithm in *Loci* for computing vertex priority based on the heuristics described above for memory minimization, which is shown in the procedure PrioGraph. Given a graph, we start off from building a list of statistical information for each vertex. Line 3 to line 5 compute the allocation number, the deallocation number, and the number of outgoing edges for all target relations respectively for every rule (for a relation, these numbers are all 0). Then all vertices that do not have allocation number get a priority of 0, which represents the highest priority. Then we sort the remaining list first according to the ascending order of the allocation number (line 2) and then the descending order of deallocation number (line 3). After this, all the remaining rules will be ordered according to their deallocation and allocation number. We assign appropriate priority to each rule that causes deallocation. The remaining rules are sorted again according to the number of outgoing edges for target relations (line 10) and priorities are assigned accordingly.

6. Experimental Results. In this section, we present some of our measurements for the work discussed in the previous sections. First of all, *Loci* planning is carried out at run-time. Our work in memory management incurs some additional costs to the planner. We performed a measurement first for the planner itself in order to evaluate the planning performance. Table 6.1 shows our measurement of various planning stages discussed in

previous sections. The measurement is performed on a typical Linux workstation for an average complex *Loci* application. We can conclude that the planning overhead is virtually negligible since typical running time for *Loci* applications range from hours to several days on large parallel machines. For substantially larger problems (e.g., a complex unstructured grid), the total planning time will increase correspondingly. But the planning for the work addressed in the paper only depends on the number of rules and relations in an application and does not relate to the input problem size. The measurements here should be a good suggestion for practical problems we are currently considering.

TABLE 6.1
Loci planner statistics

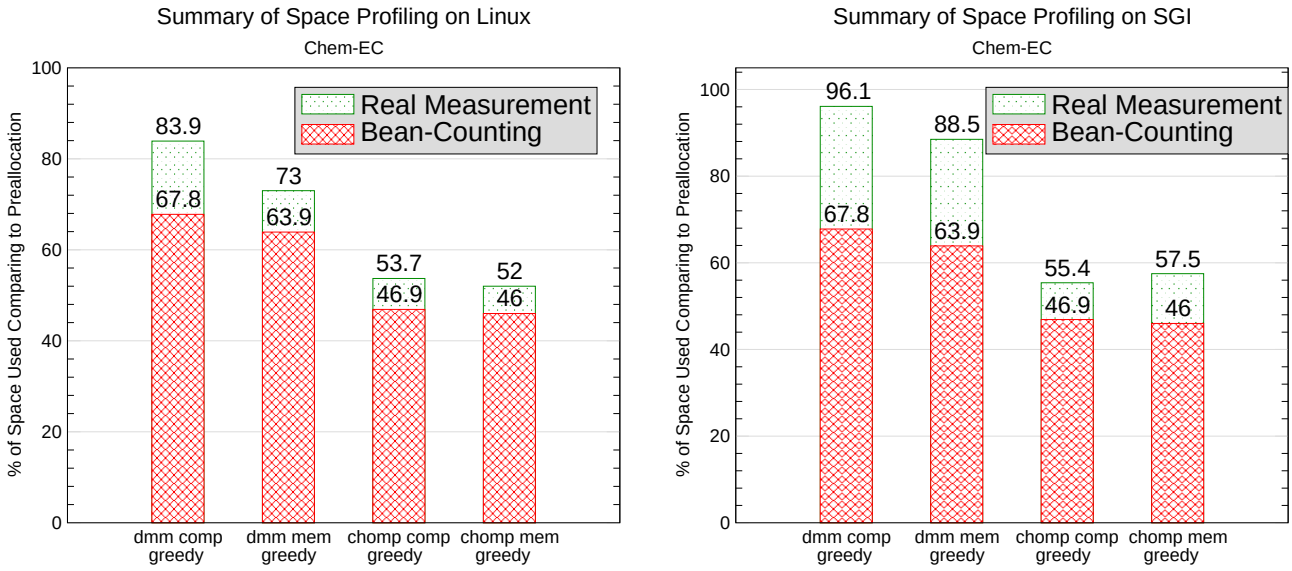
	unit: second
decoration	0.5402
chomping chain searching	0.3760
computation greedy schedule	0.0103
memory greedy schedule	0.1350
total *Loci* planning time	10.9706

We used the CHEM program as the benchmark in our profiling. CHEM [7, 8] is a finite-rate non-equilibrium Navier-Stokes solver for generalized grids fully implemented using the *Loci* framework. CHEM can be configured to run in several different modes, they are abbreviated as Chem-I, Chem-IC, Chem-E, and Chem-EC in the following figures and tables. An IBM Linux Cluster (total 1038 1GHz and 1.266GHz Pentium III processors on 519 nodes, 607.5 Gigabytes of RAM), and various Linux and SGI workstations are used in the measurement. In addition to taking the measurement of the real memory usage, we also record the bean-counting memory usage numbers. (By bean-counting we mean tabulating the exact amount of memory requested from the allocator. It is shown as a reference as we use GNU GCC's allocator in *Loci*.) In most of the measurements, we are comparing the results with the preallocation scheme mentioned in section 3, as the preallocation scheme represents the upper-bound for space requirement and the lower-bound for run-time management overhead.

We did extensive profiling of the memory utilization on various architectures. Figure 6.1(a) shows a measurement of Chem-EC on a single node on the Linux cluster. Figure 6.1(b) shows the same measurement on an SGI workstation. The "dmm" in the figure means the measurement was performed with the dynamic memory management enabled; "chomp" means chomping was also activated in the measurement in addition to basic memory management. As can be found from the figures, when combining with memory greedy scheduling and chomping, the peak memory usage is reduced to at most 52% (on Linux) of preallocation peak memory usage. The actual peak memory also depends on the design of the application. We noticed that for some configurations, the difference between the real measurement and the bean-counting is quite large. We suspect that this is due to the quality of the memory allocator. We also found that under most cases, using chomping and memory greedy scheduling improves the memory fragmentation problem. We suspect this is attributable to allocations that are much smaller and regular, thus the same allocations may be more effectively reused.

Figure 6.2(a) shows one timing result for chomping on a single node on the Linux cluster. Figure 6.2(b) shows the same measurement on an SGI workstation. The results showed different chomping sizes for different CHEM configurations. Typically using chomping increases the performance, although no more than 10% in our case. The benefit of chomping also depends on the *Loci* program design, the more computations are chomped, the more benefit we will have. The box in Fig. 6.2(a) and Fig. 6.2(b) shows the speed of dynamic memory management alone when compared to the preallocation scheme. This indicates the amount of run-time overhead incurred by the dynamic memory management. Typically they are negligible. The reason for the somewhat large overhead of Chem-I under "dmm" on Linux machine is unknown at present and it is possible due to random system interactions.

To study the effect of chomping under conditions where the latencies in the memory hierarchy are extreme, we performed another measurement of chomping when virtual memory is involved. To invoke virtual memory, we intentionally execute CHEM on a large problem such that the program had significant access to disk through



(a) Space Profiling on Linux

(b) Space Profiling on SGI

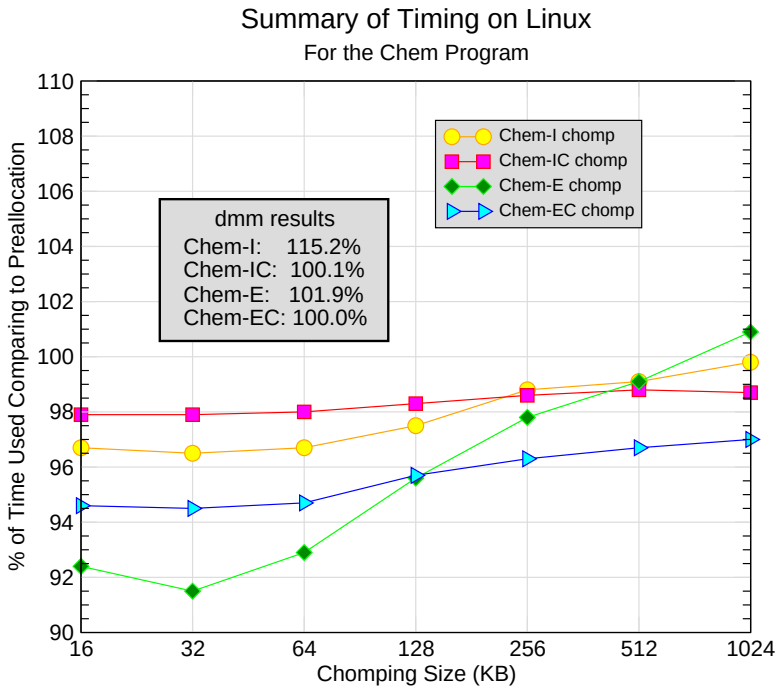
FIG. 6.1. *Space Measurement*

virtual memory. We found in this case, chomping has superior benefit. Schedule with chomping is about 4 times faster than the preallocation schedule or the schedule with memory management alone. However the use of virtual memory tends to destroy the performance predictability and thus it is desirable to avoid virtual memory when possible. For example, a large memory requirement can be satisfied by using more processors. Nevertheless, this experiment showed an interesting feature of chomping. Chomping may be helpful when we are constrained by system resources.

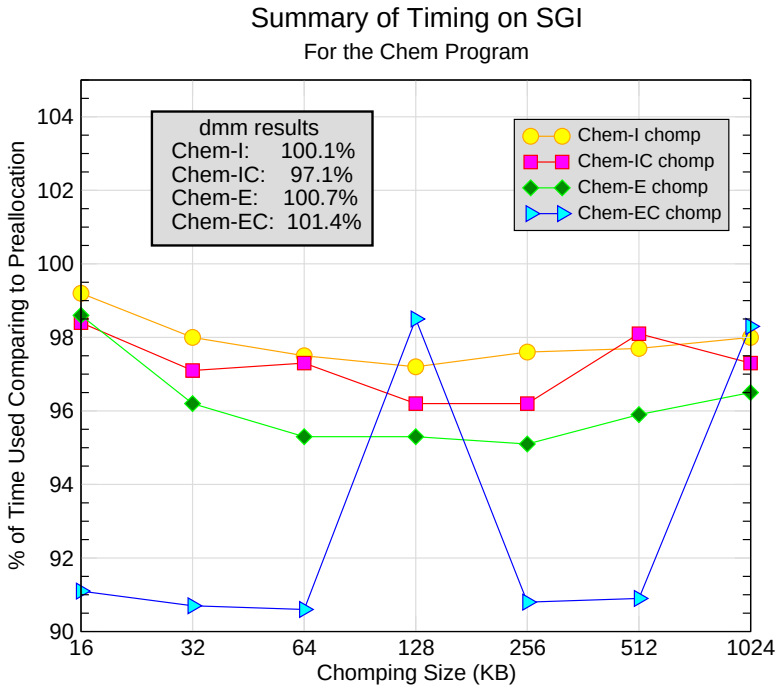
We are also interested to see how well the searching algorithm for chomping chain (as discussed in section 4.1) performs for real applications. We did a measurement for the searching for the CHEM program as shown in table 6.2. The "chomping candidates" in the table refers to relations in the program that do not involve non-affine accesses. As discussed in section 4.1, they represent the upper bound of the relations that we can possibly chomp. But the constraints of relations and rules in the graph may force us to discard some of them. We do not know whether the searching results are optimal or not. But the results shown in the table are close to the upper bound and we consider them to be good enough for practical use. We also took a measurement of the size of the total chomped relations. Interestingly, for this measurement, the percentage of size is larger than the percentage of number in total relations. This further shows the importance of chomping. If a *Loci* program is designed appropriately, from the memory management point of view, doing chomping alone would eliminate a large portion of memory requirement. Typically the peak memory is determined by the number of relations that needs to be in the memory simultaneously. If most of these relations can be chomped, then memory requirement can be potentially reduced greatly in addition to performance benefits.

TABLE 6.2
Statistics of Chomping

	CHEM-I	CHEM-IC	CHEM-E	CHEM-EC
total relations	192	196	162	166
chomping candidates	47	49	49	51
chomped relations	40	42	44	47
% of the size of chomped relations in total relations	32.25	32.39	44.74	51.03



(a) Timing on Linux



(b) Timing on SGI

FIG. 6.2. *Timing Measurement*

Finally we present one result of the comparison of different scheduling policies in table 6.3. The measurement was performed on 32 processors of our parallel cluster. We noticed the difference of peak memory usage between computation greedy and memory greedy schedule is somewhat significant, however the timing results are almost identical albeit the large difference in the number of synchronization points. We attribute this to the fact that CHEM is computationally intensive, the additional communication start-up costs do not contribute significantly

TABLE 6.3
Mem vs. Comm under dmm on Linux Cluster

	memory usage (MB)		sync		time
	real	bean-counting	points	time (s)	ratio
comp greedy	372.352	174.464	32	3177.98	1
mem greedy	329.305	158.781	50	3179.24	1.0004

to the total execution time. This suggests for computationally intensive applications, the memory greedy scheduling is a good overall choice, as the additional memory savings do not incur undue performance penalty. For more communication oriented applications, the difference of using the two scheduling policies may be more obvious. In another measurement, we artificially ran a small problem on many processors such that parallel communication is a major overhead. The results are presented in table 6.4. We found the synchronization points in the memory greedy schedule is about 1.6 times more than the one in computation greedy schedule and the execution time of memory greedy schedule increased roughly about 1.5 times. Although this is an exaggerated case, it provided some evidence that such a trade-off does exist. However, for scaling small problems, conserving memory resources should not be a concern and in this case the computation greedy schedule is recommended.

TABLE 6.4
Mem vs. Comm under chomping on Linux Cluster (A Small Case)

	bc(MB)	sync points	time (s)	time ratio
comp greedy	1.08	32	1155.55	1
mem greedy	1.05	52	1699.33	1.47

7. Conclusions. This study presented a new dynamic memory management technique implemented in a novel declarative parallel programming framework, *Loci*. The approach utilizes techniques to improve both cache utilization and memory bounds. In addition, we studied the impact of memory scheduling on parallel communication overhead. Results show that memory management is effective and is seamlessly integrated into the *Loci* framework. By utilizing the chomping technique, which is similar to strip-mining in traditional loop optimizations, we were able to reduce both memory bounds and run times of *Loci* applications. In addition, we illustrate that the aggregation performed by *Loci* also facilitates memory management and cache optimization. We were able to use *Loci*'s facility of aggregating entities of like type as a form of region inference. The memory management is thus simplified as managing the lifetime of these containers amounted to managing the lifetimes of aggregations of values. In this sense, although *Loci* supports fine-grain specification [6], the memory management does not have to be at the fine-grain level. This has some similarity with the region management concept. The initial graph decoration phase resembles the static program analysis performed by the region inference memory management, although much simpler and is performed at run-time.

We also observed an interesting phenomenon with our chomping technique. While we expected it to increase performance and to a small extent reduce memory requirements, its benefits in memory reduction were greater than expected. Apparently this result comes from two sides. The first one is due to heap fragmentation: the uniformly sized chomps were more efficiently managed by the allocator. Obviously there are interesting and complex interactions between allocation policy and heap management. The SGI platform seems to have significant benefit from the improved memory fragmentation by chomping. The second reason is due to that the memory allocations for chomping is virtually negligible. Therefore the more relations are chomped, the less we pay for their memory space allocations. We found that we were able to chomp many relations in our CHEM program and the aggregate size of all these relations is large.

We also note that the performance gains in real applications achieved by using chomping is significantly less than potential gains. While the CHEM application saw a performance increase on the order of 10 percent in some cases, simple example programs that made significant use of the chomping facility saw performance boosts of as much as a factor of five. We suspect that the cache performance suffers from large non-chain access as well as non-affine accesses to other data in the middle of the chomping chain as discussed in section 4.2. We believe that these issues may be mitigated by identifying common non-affine references and factoring them out

of the computations through program transformations. However, this transformation may require replicating some work to achieve full effect, limiting the potential performance boost.

While we have demonstrated a trade-off between an efficient memory schedule and communication barriers in the parallel program, we have not provided an automated way of selecting the appropriate policy. This is currently under user control. It would be fairly simple to use a model-based approach to decide when a memory efficient schedule would cost more than some small percentage of overall run-time and then select the appropriate policy automatically. However, preliminary investigations into replicating work to eliminate communication barriers appears in many cases to eliminate most of the barriers introduced by the memory efficient schedule. While this replication technique is still under development, we believe, based on our preliminary results, that the combination of work replication and a memory efficient schedule may provide the best of both techniques and eliminate the need for a more sophisticated policy selection mechanism.

Acknowledgment. We thank the financial support from the National Science Foundation (ACS-0085969), NASA GRC (NCC3-994), and NASA MSFC (NAG8-1930). In addition we are grateful to the anonymous reviewers for their insightful suggestions and comments.

REFERENCES

[1] B. Babcock, S. Babu, M. Datar, and R. Motwani, *Chain: Operator scheduling for memory minimization in data stream systems*, in Proceedings of the ACM International Conference on Management of Data (SIGMOD 2003), San Diego, California, June 2003.

[2] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004.

[3] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, Seattle, WA, May 1998, pp. 1381–1384.

[4] H. Lieberman and C. Hewitt, *A real time garbage collector based on the lifetimes of objects*, Communications of the ACM, 26(6) (1983), pp. 419–429.

[5] E. A. Luke, *Loci: A deductive framework for graph-based algorithms*, in Third International Symposium on Computing in Object-Oriented Parallel Environments, S. Matsuoka, R. Oldehoeft, and M. Tholburn, eds., no. 1732 in Lecture Notes in Computer Science, Springer-Verlag, December 1999, pp. 142–153.

[6] E. A. Luke and T. George, *Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis*, Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming, 15 (2005), pp. 477–502. Cambridge University Press.

[7] E. A. Luke, X. Tong, J. Wu, and P. Cinnella, *Chem 2: A finite-rate viscous chemistry solver – the user guide*, tech. report, Mississippi State University, 2004.

[8] E. A. Luke, X. Tong, J. Wu, L. Tang, and P. Cinnella, *A step towards "shape-shifting" algorithms: Reacting flow simulations using generalized grids*, in Proceedings of the 39th AIAA Aerospace Sciences Meeting and Exhibit, AIAA, January 2001. AIAA-2001-0897.

[9] S. M. Nettles and J. W. O'Toole, *Real-time replication-based garbage collection*, in Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation, Albuquerque, NM, June 1993, pp. 217–226.

[10] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, *Questions and answers about BSP*, Scientific Programming, 6 (1997), pp. 249–274.

[11] M. Tofte and L. Birkedal, *A region inference algorithm*, Transactions on Programming Languages and Systems (TOPLAS), 20 (1998), pp. 734–767.

[12] J. Ullman, *Principles of Database and Knowledgebase Systems, Volume I*, Computer Science Press, 1988.

[13] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

[14] P. R. Wilson, *Uniprocessor garbage collection techniques*, in Proceedings of International Workshop on Memory Management, St. Malo, France, 1992, Springer-Verlag.

[15] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, *Dynamic storage allocation: A survey and critical review*, in Proceedings of International Workshop on Memory Management, Kinross, Scotland, 1995, Springer-Verlag.

[16] Y. Zhang, *Dynamic memory management for the Loci framework*, master's thesis, Mississippi State University, Mississippi State, Mississippi, May 2004.

Edited by: Frédéric Loulergue
Received: October 3, 2005
Accepted: February 1st, 2006

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 39–49. http://www.scpe.org
ISSN 1895-1767

A PARALLEL RULE-BASED SYSTEM AND ITS EXPERIMENTAL USAGE IN MEMBRANE COMPUTING *

DANA PETCU†

Abstract. Distributed or parallel rule-based systems are currently needed for real applications. The proposed architecture of such a system is based on a wrapper allowing the cooperation between several instances of the rule-based system running on different computers of a cluster.

As case study a parallel version of the Java Expert System Shell is built. Initial tests show its efficiency when running classical benchmarks. Moreover, this parallel version of Jess is successfully used to accelerate current simulators for membrane computing.

Key words. Rule-based system, cluster computing, membrane computing

1. Introduction. The suites of benchmarks, like the one reported in [30], show that the rule-based or production systems, running on the current hardware, need hours to give the solutions when the number of rules to be fired are measured in thousands. In this context, parallel, distributed or grid version of those systems are welcome. The strategies to improve the speedup using multiple processors were already discussed in the last twenty years. Details are given in Section 2.

The strategy adopted here to build parallel rule-based systems has a high degree of flexibility: to construct a wrapper for the parallel system which allows cooperation between its sequential instances. The wrapper must be the same or slightly different for most of the available production or rule-based systems.

Jess, Java Expert System Shell [23], has been taken as an example for this study mainly due to its openness towards the computing environment (communication via sockets, ability to create Java objects and call Java methods). It is a rule-based programming environment written in Java. Inspired by Clips expert system shell [13], it has grown into a complete environment of its own, being today one of the fastest rule engines available. Jess uses the Rete algorithm [19] to compile rules, an efficient mechanism for solving the difficult many-to-many matching problem. With Jess one can build software that has the capacity to reason using knowledge supplied in the form of declarative rules. Jess is used in agent frameworks, expert systems from medicine to cryptography, intelligent tutoring, robotics, and so on [17].

The structure and functionality of Parallel Jess are detailed in Section 3. Some preliminary tests on Parallel Jess were performed showing its efficiency when using a cluster of workstations. A report on these experiments is presented in Section 4.

Our motivation to build a parallel version of a rule-based system comes from a practical request: the need of a faster rule-based simulator for membrane computing. Membrane computing (or P-systems) is a domain that was qualified as emergent research front in computer science [7]. The class of P systems promise polynomial time solutions to NP-complete problems, the clue being the ability to generate an exponential working space in a linear time by means of bio-inspired operations such as cell division, membrane creation, or content replication.

A P system is a distributed parallel computing model inspired from the way the living cells process chemical compounds, energy, and information. P systems are inherently parallel and, in many variants, they also exhibit an intrinsic non-determinism, hard to be caught by sequential computers. The web page [38] contains an extensive bibliography of hundreds of papers devoted to P systems.

There are several attempts to simulate P systems on existing computers. They have both didactic and scientific values. Details can be found in Section 5.

We started from an existing Clips simulator described in [32] which allows the study of the evolution of P systems with active membranes based on production system techniques. The set of rules and the configurations in each step of the evolution are expressed as facts in a knowledge base. In Section 6 we first prove that a splitting technique of the membranes in several Java threads running embedded Jess can lead to faster simulation. Then we use the Parallel Jess to further speedup the simulation.

Further directions of concept development and some conclusions can be found in Section 7.

*This work was partially supported by the projects CEEX-I03-47-2005-ForMOL and CNCSIS-949-2004-CompGrid funded by Romanian Ministry of Research.

†Computer Science Department, Western University of Timişoara, and Institute e-Austria Timişoara, B-dul Vasile Pârvan 4, 300223 Timişoara, Romania (petcu@info.uvt.ro).

2. Parallel production or rule-based systems. A production system (PS) consists in a working memory, a set of rules and an inference engine. The working memory is a global database of data (elements) representing the system state. A rule is a condition-action pair. The inference engine is based on a three-phase cyclic execution model of condition evaluations (matching), conflict-resolution and action firing. An instantiation is a rule with a set of working memory elements. A conflict set is the set of all instantiations. Firing an instantiation can add, delete or modify elements in the working memory.

In a sequential environment, conflict-resolution selects one instantiation from the conflict set for firing. In a parallel environment, multiple instantiations can be selected to fire simultaneously.

Encompassing a high degree of parallelism, the time performance of production systems can be improved through parallel processing. The interest in parallel production systems has been raised after 1984 and the first parallel implementations were already available at the beginning of the last decade. Amaral in [1] presents a comprehensive synthesis of the efforts made before 1994 in providing parallel firing systems. We mention here only the most representative contributions: PESA—a parallel architecture for PSs; Rubic—multiprocessor for PSs, multiple rule firing PSs on hypercube; PARULEL—parallel rule processing using meta-rules; PPL—parallel production language.

Concerning the Jess predecessor, Clips, there are several parallel and distributed implementations produced mainly before 1994: for parallel shared memory architectures [8] and [40]; for hypercubes [22, 29]; for parallel distributed memory architectures—dClips [25] and PClips [26]; for distributed system architectures, [20] and using PVM [27]. For example, in the version reported in [22] and developed in 1994 to run on Intel hypercubes new added commands allows parallel calls. A complete version of Clips runs on each node of the hypercube, only rule-level parallelism is supported, and parallel commands enable the assertion and retraction of facts to and from remote nodes working memory.

Unfortunately, none of the above mentioned parallel and distributed versions of rule-based systems are anymore available in the public domain.

The interest in distributed rule-based engines has appear again in last five years in connection with Jess. Recently Jess was used in conjunction with a Java implementation of an actor model [16] to write distributed artificial intelligence applications. In the computational environment each Jess is an active independent computational entity with the ability to communicate with other Jess instances. The OKEANOS middleware [41] provides an infrastructure for mobile agents that access computational services and communicate by passing messages. The agents are implemented in Java and contain rule-based knowledge interpreted by Jess. They consist of two parts: one is responsible for managing messages and for transforming them into Jess-based declarative rules, and the other one, the Jess engine, interprets the incoming rules, new facts are added to the knowledge base, or existing facts are retracted from it, depending on the content of the incoming messages. The resulting knowledge base of a Jess-agent determines its state.

The parallel matching approach parallelizing only the match phase leads to a limited speedup by the sequential execution of rules [43]. The multiple rule firing approach parallelizing the match phase and the act phase by firing multiple rules in parallel is more promising, but supplementary costs are due to synchronization needs. Special techniques like copy-and-constraints, compatible rules, analysis of data dependency graph have been used with success to increase the parallelism. Those techniques are general and do not exploit the parallelism specific to the application domains, as it is the case here. The task-level parallelism approach based on the functional decomposition of the problem into a hierarchy of tasks can lead to better results that the above mentioned ones, but the techniques tend to be ad hoc [43].

In this context we are interested to build a parallel distributed memory version of Jess based on task parallelism.

3. Parallel Jess. In our tests, Jess was chosen over several other rule-based systems because of its active development and support, tight interaction with Java programs, and expressiveness. The Jess rule language includes elements not present in many other production systems, such as arbitrary combinations of boolean conjunctions and disjunctions. Its scripting language is powerful enough to generate full applications entirely within the Jess system [17]. The core Jess language is still compatible with Clips, in that many Jess scripts are valid Clips scripts and vice-versa. Jess adds many features to Clips, including backwards chaining, working memory queries, and the ability to manipulate and directly reason about Java objects. Jess is also a powerful Java scripting environment, from which one can create Java objects and call Java methods without compiling any Java code.

As stated above, we are interested in building a parallel version of Jess based on task parallelism. The target architecture that we consider is a homogeneous cluster of workstations. Extensions towards Grid architecture are not excluded.

Building a rule-based application based on socket communication facilities can distract the user from its main aim, to solve a concrete problem. To avoid this, a middleware is needed.

For the Parallel Jess structure we adopted a modular scheme composed by Instances, Connectors, and Messengers. The Instances are the Jess kernels.

The modular approach has been taken in consideration to simplify the task of migration towards other production systems or other message passing protocols. In the case of a such migration, the Connector must be rewritten to adopt the new production system, or the Messenger must be rewritten to adopt the new message passing protocol. A graphical representation of the proposed structure is given in [35].

The task farming model is used. Each Jess instance has a unique identifier stored in a Jess variable p. The are two types of Jess Instances: the one controlled by the user normally using a local Jess interface—the master or farmer, labeled 0 –, and the ones controlled by the user via the system—the slaves or workers, labeled starting from 1. While the master's Jess runs in the interactive version, a worker's Jess is running in a Java embedded version. The first one is launched by the user. The later one is possible to run on a remote computing platform and it is launched by a Java code.

The first part of the Jess wrapper for the parallel version, the Connector, is written in Java. Each Jess instance has one corresponding Connector. The Jess instance acts as a client and contacts via socket its Connector, the server. The Connector uses the standard Java ServerSockets methods. If the connection is established, the Connector interprets the Jess special incoming requests. Those requests are concerning either a communication or an action on other Jess instance: send or receive an information, launch or kill other instances. A message in transit is a string containing a command written in Jess language.

The second part of the Jess wrapper is the Messenger. Each Messenger is associated with one Connector and its purpose is to execute the commands received by the Connector, and to communicate with the Messengers associated with the other Connector and Jess instances. The Messenger is written in Java and JPVM [24], a Java implementation of Parallel Virtual Machine.

JPVM was selected instead mpiJava or other similar environments for message passing, due to its ability to dynamically create and destroy tasks. Adopting a PVM variant, the user is absolved of the duties to nominate the hosts on which the Jess instances are running, to treat sequentially the incoming messages, or to check the status of the machines on which the Jess instances are running. Asynchronous incoming message delivery, checks for incoming high priority messages, or hierarchies of Jess instances are possible.

The set of new commands added to Jess language is the minimal one required to implement a message passing interface. These commands are described in Table 3.1.

A very simple example of using Parallel Jess is provided in Figure 3.1. The commands given by the user to the Jess interface are displayed. JPVM daemon must be already activated. First, the user loads the new Jess function definitions by specifying the ParJess file. The local Connector & Messenger are started, and the connection between the Jess instance and the Connector is establish by calling the connection function. Then two new Jess instances are launched. Depending on the current JPVM configuration those Jess instances

TABLE 3.1
Jess functions defined in ParJess.clp

Function	Description
connection	Establishes the socket connection between the Jess instance and the Connector
kernels n	Launches n embedded Jess instances
kill n	Stops the nth embedded Jess instance
send n t s	Sends to the nth Jess instance the message labeled t and containing the string s (non-blocking function); if n is -1, the message is broadcast to all Jess instances
recv n t	Returns a string representing a message labeled t and received from the nth Jess kernel; if n is -1, it is accepted the first message from any kernel; if t is -1, it is accepted the message with any label; if no message has arrive, the execution is blocked until the message arrives
prob n t	Tests if a message labeled t send by the nth instance has arrive (non-blocking function); returns "t" or "f"; the meaning of -1 is the same as in the case of recv
stop	Stops all embedded Jess instances, closes the Connector-to-Messenger socket connection

```
                    Jess> (batch ‘‘ParJess.clp")
                                TRUE
                    Jess> (connection)
                                Connection established
                    Jess> (kernels 2)
                                2 kernels launched
                    Jess> (send -1 1 ‘‘(sqrt (+ ?*p* 1))")
                                Multicast successful
                    Jess> (recv 1 2)
‘‘1.4142135623730951"
                    Jess> (recv 2 2)
‘‘1.7320508075688772"
                    Jess> (stop)
                                Connection closed
```

FIG. 3.1. *A simple example*

can run on the same machine or on remote machines. A simple command to perform $\sqrt{p+1}$ where p is the instance identifier is send to the new Jess instances. The first number in the send command indicates the destination instance; -1 is used to broadcast the request to all instances. The label 1 in the send command express the fact that it is the first request addressed to the destination instance. The results from each instance are received in a sequence of three commands. The label 2 used in each receive command means that is the second action concerning the specified instance. The results are received in the form of strings. Finally the three Jess instances (one master and two slaves) are killed, the local socket connection is closed, and all the Connectors and Messengers are shutdown.

At start, each embedded Jess instance receives, from the master instance, its identifier and the list of the identifiers of the other instances in order to be able to communicate with them. The file ParJess is loaded implicitly and the workers enter in an infinite loop in which they wait to receive and execute commands from the Jess master and the other workers. The master messages have higher priority compared to the worker ones.

4. Benchmarks. To measure the efficiency of Parallel Jess several tests are needed. We used the classical benchmark from [30].

The particular test problem presented here is the Miss Manners problem. It is the problem of finding an acceptable seating arrangement for guests at a dinner party, by attempting to match people with the same hobbies, and to seat everyone next to a member of the opposite sex. The classical solution employs a depth-first search approach to the problem. The variables of the problem are the number of guests and chairs, the maximum and minimum numbers of hobbies (e.g. 128 guests, 128 chairs, max 3 hobbies, min 2 hobbies). The computation stops when a solution is found or there is no solution.

A Jess version of the solution can be found at [18]. The data are generated randomly, the number of guests of opposite sex being equal. For each guest, name, sex and list of hobbies are given. In one of the easiest cases, e.g. maximum 3 hobbies and minimum 2 hobbies the depth-first search is building a solution relative fast. But the time to obtain the solution is increasing exponentially with the number of guests and chairs. For example, for a sample of initial data, on a PIV at 2.2 GHz with 512 Mb RAM, the problem for 64 guests is solved in 7 seconds, the one for 128 guests in 110 seconds, while the one for 256 guests in 1801 seconds. If the problem is more complicated, e.g. the minimum number of hobbies is 1, the depth-first search explores several branches of the search tree until it reach a solution. For example, for another sample of initial data generated with maximum 2 hobbies and minimum 1 hobbies, the problem for 64 guests is solved in 2103 seconds, while the one for 128 guests in 7361 seconds.

The solution process is divided into p tasks as follows. It is assumed that $2p$ divides the number of guests.

In a preprocessing phase of the initial data, the data set is split into p equal fragments. Then a search is performed looking if there are at least p special guests having the same sex and the maximal number of hobbies. If the answer is yes, they are distributed each to a distinct data fragment (interchanges are possible). If no, take the ones with the most close number to the maximal number of hobbies.

Each task receives a data fragment and the lower and the upper numbers of the seats that will be treated. The special guest selected in the first phase is seating on the first chair assigned to the task which treats the data fragment. The special guest is communicated to the task which treats the left neighbor data fragment. Same rules are applied for the depth-first search on each task, but on the different data fragment. The task

TABLE 4.1
Run time improvement using Parallel Jess for Miss Manners (256 guests, max 3 and min 2 hobbies) running on one computer

No. of instances	Running time	No. of fired rules/ instance
1	2321 seconds	33406 rules
2	214 seconds	8510 rules
4	35 seconds	2206 rules
8	11 seconds	590 rules

search is complete only if at least one hobby of the guest seating on the last chair assigned to task is on the list of the special guest communicated by the task treating the right neighbor fragment of data. Finally the pairs guest-chair provided by each task are collected by the master instance.

The above mentioned Jess source was modified accordingly. A batch file was written to launch the p Jess instance, similar to the one from Figure 3.1. Each instance loads the modified Jess source and according to its instance identifier processes a fragment of the data and send and/or receive the information about the special guests sitting on the chairs nearby the ones treated by the Jess instance.

The cluster environment used in the experiments consists of 8 IBM PCs at 1.5 GHz and 256 Mb RAM connected by a Myrinet switch with a peak speed for communications of 2Gb/s. When several instances of Jess are running on one of the cluster PCs, the solution is provided faster than in the case of using only one instance—Table 4.1 proves this fact. Moreover Table 4.2 shows that due the fact that the communication is needed only at the beginning and at the ending of the above described tasks, the parallel implementation efficiency is expected to be near to the ideal value.

5. P-systems and the available simulators. In the area of biology-inspired computing, a recently introduced model has taken inspiration from the structure and the functioning of living cells: Păun in [31] proposed an abstraction of the cell architecture and the way biological substances are modified or moved among compartments. Each compartment, delimited and separated from the rest by a membrane, can be seen as a computing unit having its own data and its local program (reactions). All compartments considered as a whole (the cell) can be seen as an unconventional computing device characterized by a membrane structure, where membranes can be hierarchically placed inside a unique external membrane delimiting the entire cell. All membranes are semi-permeable barriers, which either allow some substances to move in or out and consequently change their location in the membrane structure, or block the movement of some other substances.

This cell interpretation has its mathematical formalization in P systems, also called membrane systems, where a membrane structure can be described by a finite string of well matching parentheses. The substances and reactions are represented by objects and evolution rules. Objects are described as symbols or strings over a given alphabet, evolution rules are given as rewriting rules. The rules act on objects, by modifying and moving them, and they can also affect the membrane structure, by dissolving or dividing the membranes. A computation in P systems is obtained by starting from an initial configuration, identified by the membrane structure, the objects and the rules initially present inside it, and then letting the system to evolve. The application of rules is performed in a nondeterministic and maximal parallel manner: all the applicable rules have to be used to modify all objects which can be the subject of a rule, and this is done in parallel for all membranes. A universal clock is assumed to exist. The computation halts when no rule can be further applied. The output is defined in terms of the objects sent out to the external membrane or collected inside a specified membrane.

An example of the mathematical representation of a P-system is given in Figure 5.1. The different variants of P systems found in the literature are generally thought as generating devices. Almost all implementations of P system simulators are considering deterministic P systems.

TABLE 4.2
Run time improvement (and speedup) using the cluster nodes

No. of instances	1	2		4		8	
	Time	Time	S_2	Time	S_4	Time	S_8
1	2321 s	-	-	-	-	-	-
2	214 s	112 s	1.91	-	-	-	-
4	35 s	19 s	1.84	11 s	3.18	-	-
8	11 s	6 s	1.83	4 s	2.75	3 s	3.67

In the following experiments we have considered the particular P system described in [32] and in Figure 5.2 to solve the validity problem—given a boolean formula in conjunctive normal form, to determine whether or not it is a tautology. If we consider the problem input in the form

$$\wedge_{i=1}^{m} \vee_{j=1}^{k_i} x_{ij} \quad \text{where } x_{ij} \in \{X_1, \ldots, X_n, \overline{X}_1, \ldots, \overline{X}_n\}$$

the P system solves the NP-complete problem in $5n + 2m + 4$ evolution steps (in the external membrane it is obtained 'Yes' or 'No'). The number of membranes (the P system degree) increases by division from only 3 initially (two internal ones, plus the external one) to $2^n + 2$ at the computation end. This example is of particular interest for parallel simulation using dynamic task creation.

In the attempt to implement membrane computing on the usual computer one needs to simulate nondeterminism on a deterministic machine.

There are several attempts to simulate P systems on the existing sequential computers: a Visual C++ simulation for P systems with active membranes and catalytic P systems allowing graphical simulation and step-by-step observations of the membrane system behavior is reported in [10]; a Java implementation is reported in [28]; an implementation of transition P systems in Haskell is discussed in [3, 5]; another implementation of transition P system was done in MzScheme [4]; rewriting P systems and P systems with symport/antiport rules were described as executable specifications in Maude in [2, 39]; in [36] the membrane system is programmed in VHDL; an implementation of Cayley P Systems was written in MGS [21]; transition P systems and deterministic P systems with active membranes were simulated also in Prolog [14, 15, 37]; recognizer P systems with active membranes, input membrane and external output were simulated in Clips and used to solve two NP-complete problems in [32, 33, 34].

By simulating parallelism and nondeterminism on a sequential machine one can lose the power and attractiveness of P system computing. Therefore the simulations on multiple processors are useful. A parallel and a cluster implementation for transition P systems in C++ and MPI were reported in [9, 11, 12]. A distributed implementation based on Java RMI was also described in [42].

Unfortunately, the above mentioned simulators are capable to handle only P systems solving small problems. The computational power of a P system is not proved by small problems for which we already have faster algorithms, but by large problems where the NP is becoming an issue. Surely the space expansion requested by a P system evolution is a big problem, but a step forward is to use a large amount of computing devices in the simulation.

6. Using Parallel Jess in a P system simulation. We consider that the existing Clips implementations (compatible with Jess) are written in a language close to the mathematical description of the P systems. Also they provide a natural internal mechanism allowing multiple rule firing, and they are easily adaptable to different variants of P systems.

In this context, two developing directions were foreseen: the first one deals with the simulator improvement by using multiple computing units (reported here), and the second one deals with the improvement of the Clips implementation allowing wider set of P system variants to be simulated, queries strategies for P system status, objects, membranes, or rules, and an easy to use interface for describing and studying new P systems (reported in [6]). Hopefully the two improvements will lead to a faster P system simulator than the current available ones.

P-system: $(\Sigma, H, \mu, \omega_1, \ldots, \omega_N, R)$
N: the number of initial membranes (system degree)
Σ: the alphabet of symbol-objects
H: finite set of membrane labels
μ: the membrane structure
$\omega_1, \ldots, \omega_N$: strings over Σ, the initial multisets of objects, placed in each membrane of μ
R: a finite set of evolution rules of the following forms:
–object evolution rules (inside evolution): $[a \to b]_h^p$
–send-in rules (incoming objects): $a[]_h^{p_1} \to [b]_h^{p_2}$
–send-out rules (outgoing objects): $[a]_h^{p_1} \to b[]_h^{p_2}$
–dissolution rules (membrane disolved): $[a]_h^p \to b$
–division rules (membrane multiplication): $[a]_h^{p_1} \to [b]_h^{p_2}[c]_h^{p_3}$
where $a, b, c \in \Sigma$, $h \in H$, $p, p_1, p_2, p_3 \in \{+, -, 0\}$

FIG. 5.1. *A generic P-system with active membranes*

$N = 2$
$\Sigma = \{x_{i,j}, \overline{x}_{ij} : 1 \le i \le m,\ 1 \le j \le n\} \cup \{c_k : 1 \le k \le m+1\} \cup \{d_k : 1 \le k \le 2n+2m+2\}$
$\cup \{e_k : 0 \le k \le 3\} \cup \{r_{ik} : 0 \le i \le m,\ 1 \le k \le 2n\} \cup \{\text{Yes, No}\}$
$H = \{1, 2\}$
$\mu = [[]_2]_1$
$\omega_1 = \{e_1\},\ \omega_2 = \{d_1, <\text{the input symbols}>\}$
$R = \{[d_k]_2^0 \to [d_k]_2^+ [d_k]_2^- : 1 \le k \le n\} \cup \{[x_{i1} \to r_{i1}]_2^+,\ [\overline{x}_{i1} \to r_{i1}]_2^-,\ [x_{i1} \to]_2^-,\ [\overline{x}_{i1} \to]_2^+ : 1 \le i \le m\} \cup \{[x_{ij} \to x_{i\,j-1}]_2^+,\ [x_{ij} \to x_{i\,j-1}]_2^-,\ [\overline{x}_{ij} \to \overline{x}_{i\,j-1}]_2^+, [\overline{x}_{ij} \to \overline{x}_{i\,j-1}]_2^- : 1 \le i \le m,\ 2 \le j \le n\} \cup \{[d_k]_2^+ \to d_k[]_2^0,\ [d_k]_2^- \to d_k[]_2^0, : 1 \le k \le n\} \cup \{d_k[]_2^0 \to [d_{k+1}]_2^0, : 1 \le k \le n-1\} \cup \{[r_{ik} \to r_{i\,k+1}]_2^0 : 1 \le i \le m,\ 1 \le k \le 2n-1\} \cup \{[d_k \to d_{k+1}]_1^0 : n \le k \le 3n-3\} \cup \{[d_{3n-2} \to d_{3n-1} e_0]_1^0,\ e_0[]_2^0 \to [c_1]_2^-,\ [d_{3n-1} \to d_{3n}]_2^0\} \cup \{[d_k \to d_{k+1}]_1^0 : 3n \le k \le 3n+2m\} \cup \{[r_{1\,2n}]_2^- \to r_{1\,2n}[]_2^+\} \cup \{[r_{i\,2n} \to r_{i-1\,2n}]_2^+ : 1 \le i \le m\} \cup \{r_{1\,2n}[]_2^+ \to [r_{0\,2n}]_2^-\} \cup \{[c_k \to c_{k+1}]_2^+ : 1 \le k \le m\} \cup \{[c_{m+1}]_2^- \to c_{m+1}[]_2^+,\ [c_{m+1}]_1^0 \to c_{m+1}[]_1^+,\ \{d_{3n+2m+1}[]_2^+ \to [d_{3n+2m+2}]_2^+,\ d_{3n+2m+1}[]_2^- \to [d_{3n+2m+2}]_2^-,\ [d_{3n+2m+2}]_2^- \to d_{3n+2m+2}[]_2^-,\ [d_{3n+2m+2}]_1^+ \to d_{3n+2m+2}[]_1^-,\ [d_{3n+2m+2}]_1^0 \to d_{3n+2m+2}[]_1^-\} \cup \{[e_k \to e_{k+1}]_1^+ : 1 \le k \le 2\} \cup \{[e_2]_1^+ \to Yes[]_1^+,\ [e_3]_1^- \to No[]_1^-,\ [e_1]_1^- \to No[]_1^-\}$

FIG. 5.2. *A P-system for solving the validity problem*

TABLE 6.1
Communication actions

Rule	Case	Membrane	Actions
Object evolution	$[a \to b]_h^p$	m	modify m
			send children(m) "ev" (a, b)
		child(m)	recv m "ev"
			modify m
Send-in	$a[]_h^{p_1} \to [b]_h^{p_2}$	m	modify m, father(m)
			send father(m) "in" (a)
		father(m)	recv m "in"
			modify father(m)
Send-out	$[a]_h^{p_1} \to b[]_h^{p_2}$	m	modify m, father(m)
			send father(m) "out" (b)
		father(m)	recv m "out"
			modify father(m)
Dissolve	$[a]_h^p \to b$	m	modify m, father(m), children(m)
			send all "ds" (m, father(m), children(m))
		all	recv "ds" all
			modify m, father(m), children(m)
Division	$[a]_h^{p_1} \to [b]_h^{p_2}[c]_h^{p_3}$	m	modify m, father(m), children(m)
			send all "dv" (m, father(m), children(m))
		all	recv "dv" all
			modify m, father(m), children(m)

The aim of the experiment presented here is to measure the efficiency of Parallel Jess in a cluster environment when it is applied to a particular problem, the one involving P systems for which task parallelism is easy to be detected.

Different membranes can be distributed on different machines of a cluster. They can evolve accordingly the object evolution rules independently. New rules are added to express the message exchange. The send-in and send-out rules are requesting message exchanges, one message containing the object which goes out or comes in the membrane. The membrane towards which an object is migrating must probe constantly the existence of a new incoming message from the children of that membrane. The father membranes must communicate towards their children membranes any change in their content which can affect the send-in or send-out rules to be fired in the children membranes. Table 6.1 specifies the communication actions to be taken.

The division or the dissolution of a membrane means a dynamical change of the communication structure. This problem can be solved by Parallel Jess but it is not implemented yet in the Jess-based P system simulator.

We resumed the experiments reported in [32] describing partially a Clips simulator and a solution to the validity problem using P-systems. Starting from the Clips code, a Jess code for the simulator was written.

The numbers of rules to be fired are similar to those from the classical production system benchmarks like [30]. Table 6.2 specifies the number of rules to be fired in the case of checking the validity of a boolean expression with $m = n$. The requested time is measured on a machine of the cluster mentioned in the previous section. We concentrate our attention on the part of the simulation after the final division occurs in the P

TABLE 6.2
Test problem dimension

$m \times n$	Membranes	Rules fired after the last division	Time for firing those rules	Reaching the full membrane configuration	Total time of simulation
2×2	6	335	1 s	1 s	2 s
3×3	10	779	13 s	3 s	16 s
4×4	18	1835	428	6 s	434 s
5×5	34	?	Out of memory	49 s	?

```
MASTER

(batch ‘‘ParJess.clp")
(defglobal ?*t*=(time))
(connection)
(kernels 4)
(send -1 1 ‘‘(batch \"W\")"
(recv 1 2)
(stop)
(-(time) ?*t*)
```

```
WORKER (W)

 (batch ‘‘simulator")
(load-facts ‘‘Psyst")
(initialize)
(run)
```

FIG. 6.1. *Master and workers batch files in the case of a P-system and 4 workers*

system (the most consuming part of the simulation), i. e. when we have already $2^n + 2$ membranes.

The Parallel Jess code activated by the user is very simple and it is depicted in Figure 6.1. Table 6.3 describes shortly the code added in order to ensure the correct distribution and evolution of the membranes.

TABLE 6.3
Changes in the simulator code

What	How
Membrane template	New slot ‘owner’
Evolution, send-in/out rules	Fired only if the Jess instance owns the membrane
Send-in rule	Send to the father the object to be erased from its content
Send-out rule	Send to the father the object to be added to its content
New phase: send-recv	After evolution, send-in-out, messages are received
New rule: send-father-status	If the content of the father was changed, the new content is send to all membrane children
New rule: recv-father-status	As response to an incoming message indicating the father change, the local copy of the father is changed
New rule: recv-child-erase	As response to an incoming message indicating the child change, the incoming object is erased from father content
New rule: recv-child-add	As response to an incoming message indicating the child change, the incoming object is added to the father content

Each Jess instance reads the simulator rules, all the facts (the particular rules to be applied) and the membrane structure and contents. Each membrane is owned by a Jess instance. An instance can own one or several membranes. Rules are fired by the instance only if they are referring to an owned membrane. Each instance has copies of other instance membranes, with the actual content or an old one. A change in the content of the father membrane can lead to a change in the agendas of the membrane children. Sources and destination numbers used in the send and receive commands are referring to the membrane owners. The receiving rules do not have a blocking effect, unless the membrane receiver cannot evolve further without any incoming messages: the message arrival is tested again for different kinds of messages (due to evolution, send-in or send-out rules) until a message is received from another membrane owner or from the master instance. The send and receive are activated only if the two membranes involved in the exchange have different owners.

A significant time reduction of the running time of the P system simulator was obtained using the Parallel Jess version running on only one machine. Table 6.4 shows some examples: the shorter time is underlined. The number of Jess instances working concurrently on the same machine and leading to the lowest simulation time depends on the problem dimension. It seems that, at least in the test cases, for a $m \times m$ problem it is recommended to use m working Jess instances. To explain this phenomena, we remember that the basic

TABLE 6.4
Simulation time: the Jess instances are running on one machine

Membranes	Instances			
	1	2	3	4
6	1 s	1 s	3 s	3 s
10	13 s	5 s	5 s	10 s
18	428 s	45 s	25 s	33 s
34	Memory out	1054 s	325 s	200 s

rules have been changed: they can be fired only if the Jess instance owns the membrane. So the time for the matching process is considerably lower in each instance than in the case when only one Jess instance treats all the membranes.

A small time variation was registered when the membrane distribution to different instances was changed.

TABLE 6.5
Simulation time, speedup and efficiency: the Jess instances are running on different machines of the cluster

Instances	Machines	Membranes			
		6	10	18	34
1	1	1 s	13 s	428 s	Mem. out
2	1	1 s	5 s	45 s	1054 s
	2	1 s	3 s	23 s	528 s
	S_2	1	1.7	1.9	2
	E_2	0.51%	0.65%	0.95%	0.99%
3	1	3 s	5 s	25 s	325 s
	3	3 s	2 s	10 s	126 s
	S_3	1	2.5	2.5	2.6
	E_3	0.33%	0.83%	0.83%	0.87%
4	1	3 s	10 s	33 s	200 s
	4	2 s	3 s	9 s	52 s
	S_4	1.5	3.3	3.7	3.8
	E_4	0.37%	0.82%	0.91%	0.96%

Further reduction of the simulation time is expected when the Jess instances are running on different machines. This expectation is confirmed by the tests. Table 6.5 refers to some of them. The speedup S_p and the efficiency E_p of the parallel implementation are registered in this table. Those values are close to the ideal ones. It is easy to see the normal increase of the speedup with the number of rules to be fired. We expect to obtain better results for larger dimension for the validity problem.

7. Conclusions and further improvements. Several approaches to construct parallel rule-based engines were discussed in this paper. Following one approach, we have initiated the development of Parallel Jess, a wrapper for Jess enabling it to cooperate with other Jess instances running in a cluster environment.

At this stage, Parallel Jess exists as a demo system. Changing the message passing interface from JPVM to MPI will allow the system migration towards Grids to connect more than one cluster running several Jess instances.

A version for parallel Clips will be soon derived using the same wrapping model. A first step to do that has been already undertaken: to enrich Clips with a socket communication facility.

Further extensions to other languages for production systems is the subject for discussions.

We demonstrated the efficiency of Parallel Jess on a concrete application involving P systems. The P-system simulator using the Parallel Jess will be further developed to include facilities for membrane dissolution and division; tests are necessary to compare the current parallel implementation with other existing ones.

Further experiments are needed, not necessarily related to P systems.

REFERENCES

[1] J. N. AMARAL, *A parallel architecture for serializable production systems*, Ph.D. Thesis, University of Texas, Austin, 1994, available at `ftp://ftp.capsl.udel.edu/pub/people/amaral/tese.ps.gz`

[2] O. ANDREI, G. CIOBANU AND D. LUCANU, *Rewriting P systems in Maude*, Pre-procs. 5th Workshop on Membrane Computing, WMC5, Milano, Italy, 2004, available at `http://psystems.disco.unimib.it/procwmc5.html`

[3] F. Arroyo, C. Luengo, A. V. Baranda and L. F. de Mingo, *A software simulation of transition P systems in Haskell*, in Procs. 3rd Workshop on Membrane Computing WMC2, Curtea de Arges, Romania, 2002, Gh. Paun et al. (eds.), Springer, LNCS 2597 (2003), pp. 19–32.

[4] D. Balbontin Noval, M. J. Perez-Jimenez, F. Sancho-Caparrini, *A MzScheme implementation of transition P systems*, in Procs. 3th Workshop on Membrane Computing, WMC2, Curtea de Arges, Romania, 2002, Gh. Paun et al. (eds.), Springer, LNCS 2597 (2003), pp. 58–73.

[5] A. V. Baranda, F. Arroyo, J. Castellanos, R. Gonzalo, *Towards an electronic implementation of membrane computing: A formal description of non-deterministic evolution in transition P systems*, in Procs. 7th Workshop on DNA Based Computers, Tampa, Florida, 2001, N. Jonoska and N.C. Seeman (eds.), Springer, LNCS 2340 (2002), pp. 350–359.

[6] C. Bonchiş, G. Ciobanu, C. Izbaşa and D. Petcu, *A Web-based P systems simulator and its parallelization*, in Procs. UC 2005, C.S. Calude et al. (Eds.), LNCS 3699 (2005), pp. 58–69.

[7] C. S. Calude and G. Păun, *Computing with cells and atoms: after five years*, CDMTCS Research Report Series, CDMTCS-246, 2004, available at `http://www.cs.auckland.ac.nz/CDMTCS/researchreports/246cris.pdf`

[8] Y. Cengeloglu, S. Khajenoori and D. Linton, *A framework for dynamic knowledge exchange among intelligent agents*, in Procs. AAAI Symposium, 1994.

[9] G. Ciobanu, R. Desai and A. Kumar, *Membrane systems and distributed computing*, in Membrane Computing, Procs. WMC2, Curtea de Arges, Romania, 2002, Gh. Paun et al. (eds.), Springer, LNCS 2597 (2003), pp. 187–202.

[10] G. Ciobanu and D. Paraschiv, *Membrane software. A P system simulator*, Fundamenta Informaticae 49, no. 1-3 (2002), pp. 61–66.

[11] G. Ciobanu and G. Wenyuan, *A parallel implementation of the transition P systems*, in Pre-procs. 4th Workshop on Membrane Computing, Taragona, Spain (2003), pp. 169–184.

[12] G. Ciobanu and G. Wenyuan, *P systems running on a cluster of computers*, in Procs. 4th Workshop on Membrane Computing, WMC3, Taragona, Spain, 2003, C. Martin-Vide et al. (eds.), Springer, LNCS 2933 (2004), pp. 123-139.

[13] *Clips, A tool for building expert systems*, available at `http://www.ghg.net/clips/Clips.html`

[14] A. Cordon-Franco, M. A. Gutierrez-Naranjo, M. J. Perez-Jimenez and F. SanchoCaparrini, *A Prolog simulator for deterministic P systems with active membranes*, in Procs. 1st Brainstorming Week on Membrane Computing, Taragona, Spain (2003), pp. 141–154.

[15] A. Cordon-Franco, M. A. Gutierrez-Naranjo, M. J. Perez-Jimenez, A. Riscos-Nunez and F. Sancho-Caparrini, *Implementing in Prolog an effective cellular solution for the Knapsack problem*, in Procs. 4th Workshop on Membrane Computing, WMC3, Taragona, Spain, 2003, C. Martin-Vide et al. (eds.), Springer, LNCS 2933 (2004), pp. 140–152.

[16] D. Fezzani, J. Desbiens, *Epidaure: A Java distributed tool for building DAI applications*, in Procs. EuroPar'99, P.Amestoy et al (eds), Springer, LNCS 1685 (1999), pp. 785–789.

[17] E. Friedman-Hill, *Jess in action: rule-based systems in Java*, Manning Publications, 2003.

[18] E. Friedman-Hill, *Jess: manners and waltz test for Jess* (2003), available at `http://www.mail-archive.com/jess-users@sandia.gov/msg05113.html`

[19] C. L. Forgy, *Rete: A fast algorithm for the many pattern/many object pattern match problem*, Artificial Intelligence 19 (1982), pp. 17–37.

[20] D. Gagne, A. Garant, *DAI-Clips: Distributed, asynchronous, interacting Clips*, Procs. Clips'94 (electronic version), 3rd Conf.on Clips, Lyndon B. Johnson Space Center (1994), pp. 297–306, available at `http://www.ghg.net/clips/Clips.html`

[21] J. L. Giavitto, O. Michel, J. Cohen, *Accretive rules in Cayley P systems*, in Procs. 3th Worshop on Membrane Computing, WMC2, Curtea de Arges, Romania, 2002, Gh. Paun et al. (eds.), Springer, LNCS, 2597 (2003), pp. 319-338.

[22] L. O. Hall, B. H. Bennett, I. Tello, *PClips: Parallel Clips*, Procs. Clips'94 (electronic version), 3rd Conf.on Clips, Lyndon B. Johnson Space Center, 1994, pp. 307–314, available at `http://www.ghg.net/clips/Clips.html`

[23] *Jess, the rule engine for Java platform*, available at `http://herzberg.ca.sandia.gov/jess/`

[24] *JPVM, The Java Parallel Virtual Machine*, 1999, available at `http://www.cs.virginia.edu/~ajf2j/jpvm.html`

[25] P. Y. Li, *dClips: A distributed Clips implementation*, Procs. AIAA Computing in Aerospace Conference, San Diego, 1993, AIAA Paper 93-4502-CP.

[26] R. Miller, *PClips: A distributed expert system environment*, Procs. Clips Users Group Conf, 1990.

[27] L. Myers, K. Pohl, *Using PVM to host Clips in distributed environments*, Procs. Clips'94 (electronic version), 3rd Conf.on Clips, Lyndon B. Johnson Space Center, 1994, pp. 177–186, available at `http://www.ghg.net/clips/Clips.html`

[28] I. A. Nepomuceno-Chamorro, *A Java simulator for basic transition P systems*, in Procs. 2nd Brainstorming Week on Membrane Computing, Sevilla, Spain, 2004, available at `http://www.gcn.us.es/Brain/bravolpdf/SIM21.pdf`

[29] L. O'Higgins Hall, L. Prasad, E. Jackson, *Parallel Clips for current hypercube architectures*, Procs. FLAIRS'93, Ft. Lauderdale, 1993.

[30] *OPS5 benchmark suite*, available at `http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/` (1993), and `http://www.pst.com/benchcr2.htm` (2003).

[31] G. Păun, *Computing with membranes*, in Journal of Computer and System Sciences, 61, pp. 108–143, 2000 and TUCS Report No. 208, 1998.

[32] M.J. Perez-Jimenez, F.J. Romero-Campero, *A Clips simulator for recognizer P systems with active membranes*, 2002, available at `http://www.gcn.us.es/Brain/bravolpdf/Clips.pdf`

[33] M. J. Perez-Jimenez, F. J. Romero-Campero, *Solving the BinPacking problem by recognizer P systems with active membranes*, in Procs. 2nd Brainstorming Week on Membrane Computing, 2004, Sevilla, Spain, Gh. Paun et al (eds.) (2004), pp. 414–430.

[34] M. J. Perez-Jimenez, F. J. Romero-Campero, *Trading polarizations for bi-stable catalysts in P systems with active membranes*, in Pre-procs. 5th Workshop on Membrane Computing WMC5, Milano, Italy, (2004), pp. 327–342.

[35] D. Petcu, *Parallel Jess*, in Procs. ISPDC 2005, 4-6 July 2005, Lille, IEEE Computer Society Press, Los Alamitos (2005), pp. 307–314.

[36] B. Petreska, C. Teuscher, *A reconfigurable hardware membrane system*, in Procs. 4th Workshop on Membrane Computing, WMC3, Taragona, Spain, 2003 C. Martin-Vide et al (eds.), Springer, LNCS 2933 (2004), pp. 269–285.
[37] P. Prakash Mohan, *Computing with membranes*, 2001, available at `http://psystems.disco.unimib.it/download/rep.zip`
[38] *P systems Web page*, 2004, available at `http://psystems.disco.unimib.it`
[39] Z. Qi, C. Fu, D. Shi and J. You, *Specification and execution of P systems with symport/ antiport rules using rewriting logic*, Pre-procs. 5th Workshop on Membrane Computing, WMC5, Milano, Italy, (2004).
[40] G. Riley, *Implementing Clips on a parallel computer*, Procs. SOAR '87, NASA/Johnson Space Center, Houston, TX, 1987.
[41] R. D. Schimkat, W. Blochinger, C. Sinz, M. Friedrich and W. Küchlin, *A service-based agent framework for distributed symbolic computation*, in Procs. HPCN 2000, M. Bubak et al (eds.), LNCS 1823 (2000), pp. 644–656.
[42] A. Syropoulos, E. G. Mamatas, P. C. Allilomes and K. T. Sotiriades, *A distributed simulation of transition P systems*, in Procs. 4th Workshop on Membrane Computing, Taragona, Spain, 2003, C. Martin-Vide et al. (eds.), Springer, LNCS 2933 (2004), pp. 357–368.
[43] S. Wu, D. Miranker and J. Browne, *Towards semantic-based exploration of parallelism in production systems*, TR-94-23, 1994, available at `http://historical.ncstrl.org/litesite-data/utexas_cs/tr94-23.ps.Z`

Edited by: M. Tudruj, R. Olejnik.
Received: February 24, 2006.
Accepted: July 30, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 51–59. http://www.scpe.org
ISSN 1895-1767

AN EFFICIENT FAULT-TOLERANT ROUTING STRATEGY FOR TORI AND MESHES*

M. E. GÓMEZ, P. LÓPEZ AND J. DUATO†

Abstract.
In massively parallel computing system, high performance interconnection networks are decisive to get the maximum performance. While routing is one of the most important design issues of interconnection networks, fault-tolerance is another issue of growing importance in these machines, since the huge amount of hardware increases the probability of failure. This paper proposes a mechanism that provides both, scalable routing and fault-tolerance, for commercial switches to build direct regular topologies, which are the topologies used in large machines. The mechanism is very flexible and the hardware required is not complex. Furthermore, it allows a high number of faults having a minimal effect on performance.

Key words. Fault-tolerance, memory-effective routing, regular topologies, adaptive routing.

1. Introduction. In large parallel computers high-performance interconnection networks are decisive for reaching the maximum performance. While routing is one of the most important design issues of interconnection networks, fault-tolerance is another issue of growing importance, since the huge amount of hardware of large machines increases the probability of failure. Failures in the interconnection network may isolate a large fraction of the machine. Two fault models have been defined in order to deal with permanent faults: static and dynamic. In a static fault model, all the faults are known in advance when the machine is (re)booted. It needs to be combined with checkpointing techniques in order to be effective. In a dynamic fault model, once a new fault appears, some techiques are applied in order to appropriately avoid the faulty component without stopping the machine. Several approaches have been used to tolerate faults in the interconnection network. Replicating components incurs in a high extra cost. Another technique is based on reconfiguring the routing tables. This technique is extremely flexible but it may kill performance. However, most of the solutions proposed in the literature are based on fault-tolerant routing algorithms able to find an alternative path when a packet can encounter a fault. We proposed [5] a fault-tolerant routing mechanism for direct topologies that incurs in a minimal decrease of performance in the presence of faults, and reaches a reasonably high level of fault-tolerance, without disabling any healthy node and without requiring too much extra hardware.

As commented, in addition to fault-tolerance, routing is another important issue. The routing strategy determines the path that each packet follows between a source–destination pair. Routing is deterministic if only one path is provided, or adaptive, if several paths are possible between a source-destination pair. Routing strategies can be also classified depending on the place where routing decisions are taken. When using source routing, the source node calculates the path prior to packet injection and stores it in the packet header. When using distributed routing, each switch computes the next output port. The packet header only contains the destination node. Distributed routing has been used in most hardware routers for efficiency reasons.

Distributing routing is implemented following two different approaches. In the first approach, some hardware in the switches computes the output port as a function of the current and destination nodes and the status of the output port. With the use of clusters of workstations, routing based on forwarding tables was introduced. In this approach, there is a table at each switch that contains, for each destination node, the output port that must be used. The main advantage of table-based routing is that any topology and any routing algorithm can be implemented with the same commercial switches, but it is not scalable.

High-performance switch-based point-to-point interconnects are used in large cluster-based machines. In these machines, the topology is regular. Either direct networks (tori and meshes) or indirect multistage networks are the usual topology. In these large machines, source routing is inefficient due to the increased header size, and routing based on forwarding tables is also inefficient due to the table size. On the other hand, since general purpose switches are used, specifically designed routing hardware is not feasible. We proposed [6] a routing strategy for switch-based networks, Flexible Interval Routing (FIR), based on Interval Routing that allows to implement the most commonly-used deterministic and adaptive routing algorithms in meshes and tori, with a total memory requirements $O(\log N)$.

*This work was supported by the Spanish MCYT under Grant TIC2003-08154-C06-01.
†Dept. of Computer Engineering, Universidad Politécnica de Valencia, Camino de Vera, 14, 46071–Valencia, Spain, megomez@disca.upv.es

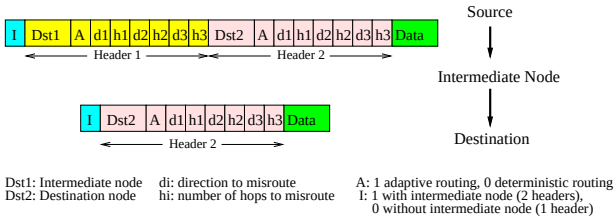


FIG. 2.1. *Packet info for the fault-tolerant methodology.*

In this paper we present an integration proposal, combining the FIR routing strategy [6] and the aforementioned fault-tolerant mechanism [5] to provide a fault-tolerant and memory-efficient routing strategy for direct networks (meshes and tori). As a result, we will provide a distributed, fault-tolerant and scalable routing strategy for general purpose switches. The rest of the paper is organized as follows. To make the paper self-contained, section 2 briefly presents the fault-tolerant methodology and section 3 describes the FIR scheme. Section 4 presents Fault-Tolerant FIR, the integration proposal. Section 5 evaluates it and finally, some conclusions are drawn.

2. Fault-Tolerant Methodology. The methodology provides fault-tolerance both in meshes and tori. It assumes a static fault model. The methodology is focused only at the computation of the new routing info[1]. The initial (i. e., without faults) routing algorithm routes packets by using Duato's fully adaptive routing with at least two virtual channels (at least one adaptive and one escape) per physical channel. The adaptive channel(s) enables routing through any minimal path. The escape channel guarantees deadlock freedom based on the bubble flow control mechanism. The methodology provides a fault-free path for each source-destination pair. To do that in the presence of faults, it uses intermediate nodes for routing between some source-destination pairs. Packets are first sent from the source node (S) to an intermediate node (I), and later, from this node to the final destination node (D)[2]. In both subpaths, minimal adaptive routing is used. If possible, the I node is selected inside the minimal adaptive cube defined by S and D. In this way, both subcubes defined by S and I, and by I and D, are inside this cube, but they are smaller and in this way the failure is avoided. In both subcubes (S-I and I-D) packets can be adaptively routed. In order to avoid deadlocks between both subpaths at the I node, we propose the use of two different escape channels. One of them will be used as escape channel for the S-I subpath and the second one for the I-D subpath. With an I node, the methodology is the 1-fault tolerant. In order to tolerate more than one failure we use two additional mechanisms: misrouting and switching off adaptive routing. Misrouting forces routing packets several hops along different directions (up to three directions in our methodology). Once misrouting is consumed, then minimal routing (adaptive or deterministic if adaptive routing is switched off) is applied. To guarantee deadlock-freedom, misrouting must use the directions according to the order established by the deterministic routing. The methodology uses the $X+Y+Z+X-Y-Z-$ direction-order routing, which is deadlock-free and provides routing flexibility since it allows routing packets in both directions of the same dimension providing non-minimal paths.

A packet routed through an I node requires two subheaders (see Figure 2.1). The first one is used in the subpath towards the I node, and the second one in the one towards the final destination. At the I node, the first subheader is removed. Packet subheaders also include control info about misrouting (direction and hops, up to three misroutings are allowed) and switching off adaptive routing (one bit). The routing info must also be stored at each source node, but not at the switches. For every destination, the possible I node (if required) and info about misrouting and switching off adaptive routing must be stored. The amount of required memory is low.

3. FIR: Flexible Interval Routing. Interval routing (IR) was proposed in [7]. IR groups those destination nodes reachable from the same output port into an interval. The intervals corresponding to different output ports of a switch are non-overlapping. Each packet is forwarded through the output port whose interval includes the destination of the packet. It is sufficient to store the bounds of each interval and to perform a paralell comparision to implement it, thus, memory space required is $2 \times d \times \log(N)$ bits per switch[3]. In a

[1] Detection of faults, checkpointing, and distribution of routing info is out of the scope of the methodology.
[2] Packets are not ejected from the network at the I node.
[3] d being the number of switch ports or switch degree and N the network size.

previous paper[6], we proposed an extension of the IR scheme, the Flexible Interval Routing (FIR), which can implement deterministic and adaptive routing on meshes and tori. Each output port has an associated interval, indicated by two registers, First Interval (FI) and Last Interval (LI). But, in order to add flexibility, we associate additional registers to the output ports. In regular topologies, each node is identified by its coordinates in each network dimension, to check if a given dimension can be used for routing, the coordinates of the current and destination nodes are compared. So in FIR, with each output port is associated a Mask Register (MR). This register has the size of n bits[4] and selects the bits of the packet destination address corresponding to the dimension of the output port. These bits are set to 1 in the MR and are the ones that will be compared with FI and LI. This is showun in Figure 3.1 taking into account cyclic intervals. If the masked destination address is inside the interval, then the hardware returns 1. This operation is done in parallel in all the output ports and the results of the comparisons for all the output ports can be stored in the Allowed Register (AR), unique for the switch, with a size of d bits.

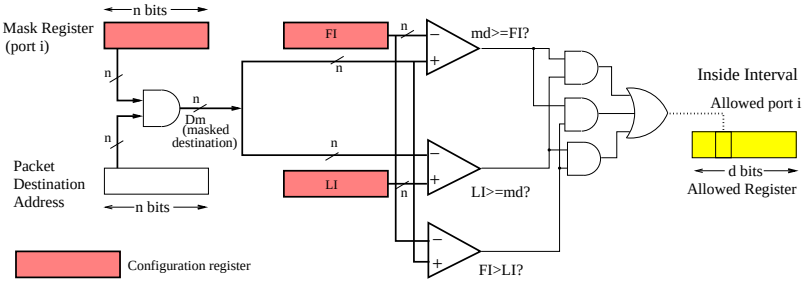


FIG. 3.1. *Hardware to generate the Allowed bit for each output port.*

Unlike IR, our proposal permits that the different intervals associated to the different output ports of a given switch overlap and, therefore, more than one output port can be allowed[5]. However, to guarantee deadlock freedom, some routing restrictions must usually be applied. In regular topologies, this is usually ensured by traversing network dimensions following some order. These routing restrictions are implemented in FIR by means of an additional register associated to each output port, the Routing Restrictions Register (RRR). It establishes, for each output port, which other output ports of the switch should be chosen prior to this one if they are allowed. This register has one bit per each output port. In a given output port i, the j bit in its RRR indicates if the output port j has highest preference (bit set to 1) or not (0) than output port i. Thus, the routing decision for a given output port i is obtained taking into account the Allowed bits of the other output ports and the bits in its RRR. If there is other output port with highest preference (bit set to 1 in the RRR) that also has its Allowed bit set to 1, this output port will not be returned to route the packet. Figure 3.2 presents the required hardware for output port i. This is done in paralell in all the output ports of the switch. The results can be stored in a Routing Register (RR), unique per switch, with a size of d bits. The RR indicates the result of the routing function. In case of adaptive routing, it may contain more than one 1.

If virtual channel multiplexing [2] is used, the configuration registers (LI, FI, MR and RRR) will be associated to virtual channels. Moreover, the RRR will have one bit per virtual channel and it will define the preferences among virtual channels. This is also the case for the registers associated to the whole switch (the Allowed and Routing Registers), that will represent virtual channels. Therefore, FIR requires, at each switch, $d \times v$ FI, LI, MR and RRR configuration registers, and one RR and AR registers of size $d \times v$, where v is the number of virtual channels per output port. The configuration of the configuration FIR registers establishes the routing algorithm. In [6] we present how to set them for the most popular routing algorithms used in meshes and tori networks. Following, we present some illustrative examples for this paper.

3.1. Illustrative Examples. To begin, a simple example explaining the register configuration for a 16-node 2-D mesh (4×4) and deterministic routing is presented. The destination addresses use 4 bits, and also the LI, FI and MR. Assuming no virtual channel multiplexing, the RRR requires one bit per each output port, so for a 2-D mesh, 4 bits are needed. Figure 3.3.(a) shows the configuration of the FIR registers when XY

[4] $n = \log(N)$, N being the network size
[5] This can be used to provide adaptive routing.

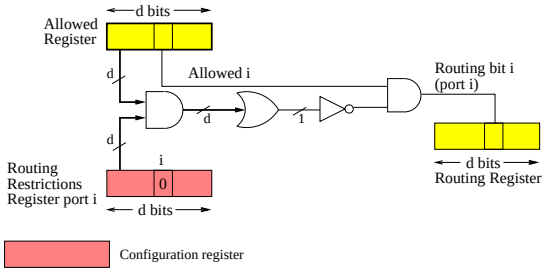


FIG. 3.2. *Routing bit for port i.*

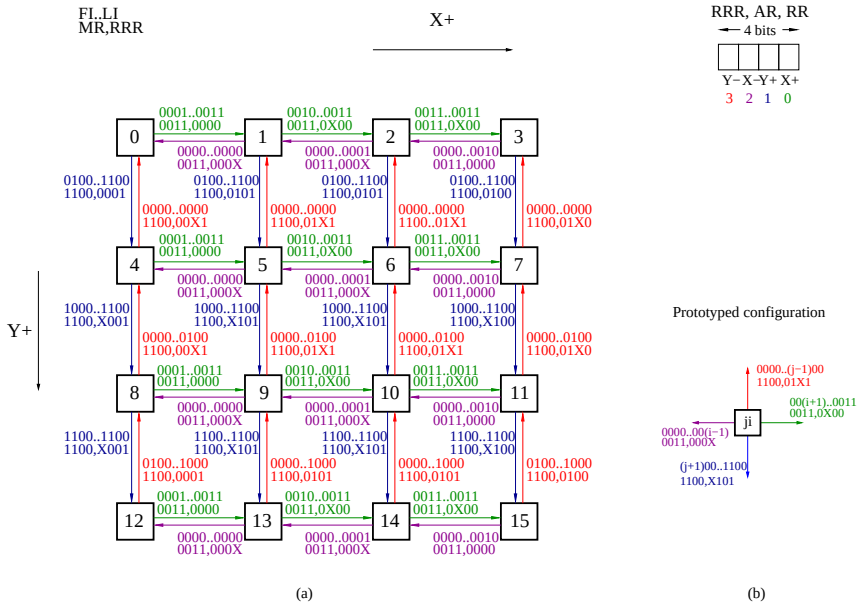


FIG. 3.3. *FIR registers configuration in a 2-D mesh with XY routing.*

routing is used. Figure 3.3.(b) shows the prototyped configuration for a central switch ji, i being the two least significant bits of the switch address and j the two most significant bits. The MR chooses the dimension bits in the destination address. This is, the two most significant bits of the destination address for the Y output ports, and the two least significant bits for the X links. Finally, the RRRs implement the XY routing. To do that, the RRR in the Y output port has the bits corresponding to the X output ports set to 1 to give preference to these output ports.

There are mainly two alternatives to prevent deadlocks in the rings of each dimension in torus networks. An alternative is the use of two virtual channels[6]. Another alternative that does not require the use of virtual channels is the bubble flow control [1]. With the bubble mechanism, when a packet is injected into the network or moves from one dimension to another dimension, two free buffers are necesary to guarantee deadlock freedom. In our proposal, an additional register associated with each output port is required to support the bubble mechanism (the Bubble Register -BR-), with one bit per input port (including the injection ports). The input ports that only need one free buffer to send through the current output port will have its associated bit set to 1. If two buffers are required by the input port in the output port, then the corresponding bit must be set to 0. Figure 3.4 shows the hardware that implements the bubble condition in the routing decision. So, the Routing bit corresponding to output port i will be obtained from the AR, its RRR, the bit associated to the input port in its BR and the space available at the output port.

FIR can implement adaptive routing. In this case, more than one bit in the RR can be set to 1. We consider fully adaptive routing based on Duato's protocol [4]. In this routing algorithm, each physical link is

[6]See [6] for a detailed FIR registers configuration.

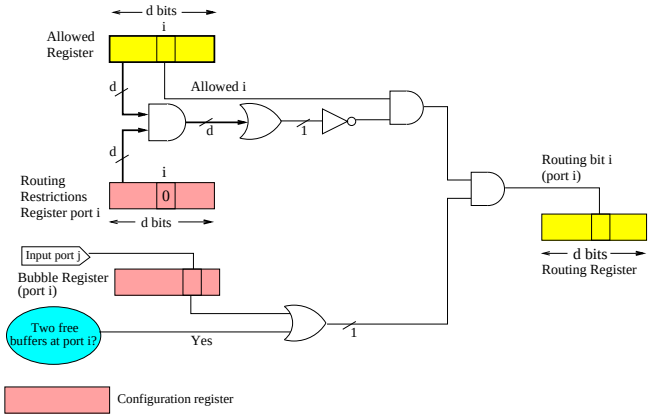


FIG. 3.4. *Bubble condition hardware in output port i.*

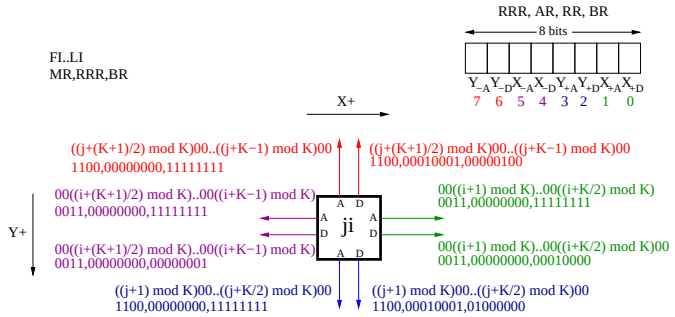


FIG. 3.5. *LI, FI, MR, RRR and BR for a* 4×4 *Torus with adaptive routing and the bubble flow control mechanism.*

split into several virtual channels, the adaptive and the escape ones. Figure 3.5 shows the prototyped register configuration for a 4×4 torus using a dimension order routing as deterministic subfunction in the escape channels. The escape channels must meet the bubble condition, so the BR must be configured. RRRs associated to the adaptive channels do not establish any order among VCs. Thefore, the routing function returns the adaptive channels that provide minimal routing and the deterministic channel returned by the deterministic subfunction.

4. FT-FIR: Fault-Tolerant FIR. In this section, we show how FIR can provide fault-tolerance using the above described fault-tolerant methodology at the expense of a few extra amount of memory and hardware. As commented before, the fault-tolerant methodology requires at least three virtual channels per physical channel. Two of them are used as escape channels with the bubble flow control, E1 and E0. E0 is used by the packets in the S-I phase, and E1 is used by the packets in the I-D phase. At least an additional adaptive channel is required to provide adaptive routing. Figure 4.1 shows the FIR registers configuration in a switch of a 2-D torus network. RRRs establish the deterministic routing in the escape channels ordering the different directions. $X+Y+X-Y-$ routing is used in order to be able to misroute packets. $X+$ direction is given the highest preference by setting the bits corresponding to the $X+$ escape channels set to 1 in the RRRs corresponding to the other directions. $Y-$ direction has the lowest preference, by setting the bits corresponding to the escape channels of the other directions set to 1 in its RRR. The new switch hardware will do the selection of the proper escape channel considering the I bit in the packet header as shown below. Only the escape channel corresponding to the current routing phase will be allowed. The RRRs associated with the adaptive channels do not establish any preference among the VCs. This means that the routing function returns all the adaptive channels that provide minimal routing and the corresponding deterministic channel following the deterministic routing subfunction. The bubble condition in the escape channels is implemented by the BR[7].

We have presented the FIR register configuration, now we explain how the FIR switch hardware must be modified to provide fault-tolerance. As commented above, packet subheaders provides information about

[7]The bits corresponding to the injection channels are not shown. They are set to 0, since two buffers are required.

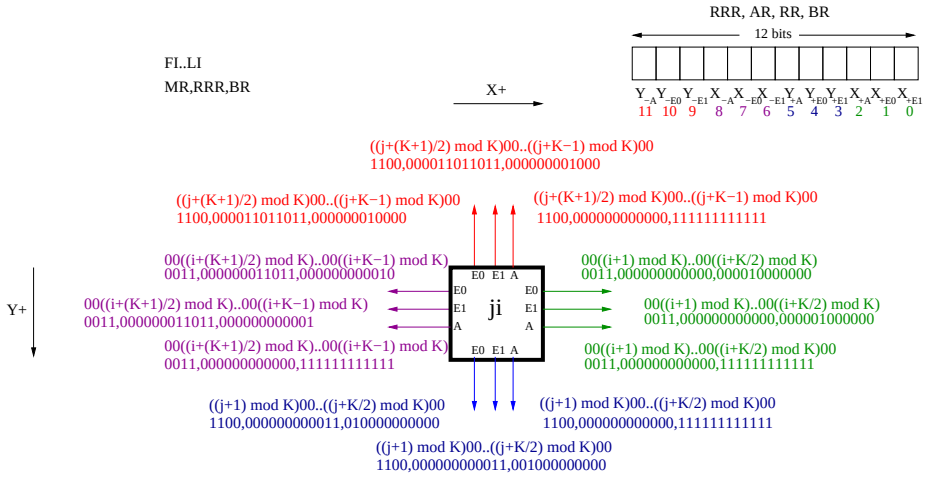


FIG. 4.1. *FI, LI, MR, RRR and BR configuration for a* 4×4 *torus and two escape channels.*

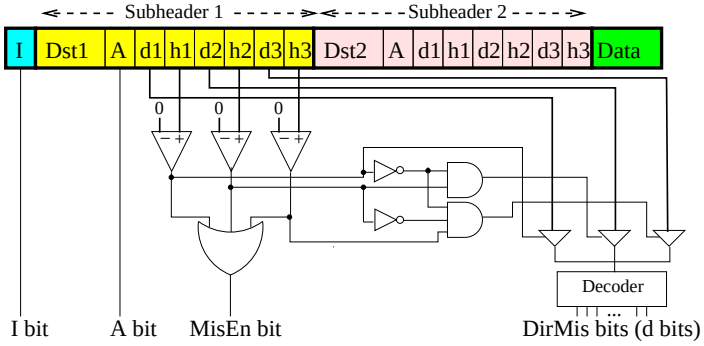


FIG. 4.2. *Hardware to obtain the fault-tolerance control bits from the packet header.*

intermediate nodes, disabling adaptivity and misrouting. From the packet header, in addition to the destination identifier, four control bits are generated (see Figure 4.2). These bits are used by the switch hardware associated with fault-tolerance. The I and A bits are obtained from the same bits of the packet subheaders. The $MisEn$ bit is set to one if the packet must be misrouted. This is true if any of the hops fields is different of 0 in the subheader corresponding to the current routing phase. $MisEn$ can be obtained by using three comparators, one per hops field in the packet subheader. The $DirMis$ field indicates the direction to misroute the packet. It has a number of bits equal to the switch degree and contains a "1" in the position that corresponds to the current direction (port) that must be used to misroute the packet. A decoder can be used to obtain the $DirMis$ bits. Its input is the first direction to misroute, which is obtained by using a multiplexer (implemented by the tristate gates and their control logic in the Figure). Figure 4.2 shows the processing of the first subheader. When the I is reached, this subheader is deleted and then the second subheader is processed as the first one.

Packets must select the proper escape channel considering the I bit in the packet header. To do that, we propose the extension shown in Figure 4.3.(a) for Figure 3.1. In this figure the Allowed bit is obtained for the escape channels, E0 and E1, considering the I bit. This bit indicates the routing phase in which the packet is. E0 is allowed only if packet is traveling to the I node, that is if I is 1. E1 only if packet is traveling to its final destination, that is if I is 0. The H bit shown in Figure 4.3 is a new configuration bit associated to each virtual channel. It indicates which escape channel corresponds to the current virtual channel. H is set to 0 in E0 escape channels and to 1 in E1 escape channels. On the contrary, the adaptive channels are not affected by the I bit.

Other mechanism consists in disabling adaptivity. If the A bit is set to 0 in any of the packet subheaders, then the packet must be routed in a deterministic way in that phase and therefore it must use the deterministic virtual channel corresponding to the proper escape channel. So, as shown in Figure 4.3.(b), if the A bit is

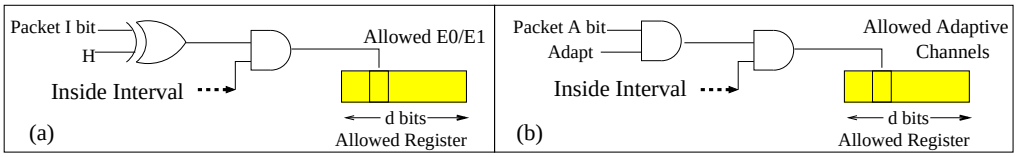


FIG. 4.3. *(a). Selection of the adequate escape channel taking into account the I bit. (b). If A=0, then the adaptive channels are not allowed.*

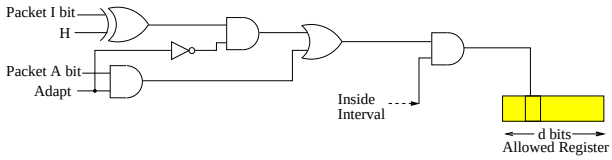


FIG. 4.4. *Combination of Figure 4.3.(a) and Figure 4.3.(b): Allowed bit taking into account the I and A bits of the packet header.*

set to 0, the adaptive channels are disabled. *Adapt* is another configuration bit associated with each virtual channel. It indicates whether the current virtual channel is adaptive (set to 1 in the adaptive virtual channels, and to 0 in the escape ones). The hardware shown in Figure 4.3.(a) is applied in the escape virtual channels. Figure 4.4 extends Figure 3.1 merging Figure 4.3.(a) and Figure 4.3.(b) to obtain the hardware for a generic virtual channel.

Therefore, two new switch configuration bits, H and *Adapt*, are required per virtual channel in order to configure the different virtual channels either as escape (E0 or E1) or as adaptive. To do that, we propose adding two registers to each switch with a number of bits equal to the number of virtual channels of the switch, that is, $d \times v$ bits. In the first register, the Adaptive Register, the bits corresponding to the adaptive channels are set to 1. In the second register, the Escape Register, the bits corresponding to escape channels E1 are set to 1.

Next, we consider the last mechanism of the fault-tolerant methodology: misrouting. When misrouting a packet, it is routed in a deterministic way, so the adaptive channels will be disabled, and the corresponding escape channel will be used instead. Figure 4.5.(b) obtains the Allowed bit for the adaptive channels when misrouting. The escape channels to use depends on the current routing phase (E0 or E1) and belongs to the first direction to misroute, even if the packet destination is not inside the interval bounds of that direction, since misrouting can provide non-minimal paths. As the packet travels along its path, the number of hops associated to that direction is decreased at the packet subheader, and when the number of hops is 0, then the next direction to misroute indicated in the packet subheader is followed. Figure 4.5.(a) shows how escape channels are allowed taking into account the $MisEn$ and $DirMis$ bits. Once the hops of all the directions to misroute at the current routing phase are consumed, the $MisEn$ bit is set to 0, and therefore packets are routed following minimal path, taking into account the FI..LI registers. Figure 4.6 combines Figure 4.5.(b) and Figure 4.5.(a).

Therefore we can say that FIR allows to implement the fault-tolerant methodology. To do that two registers (Adaptive and Escape Registers) are added to each switch with a number of bits equal to the number of virtual channels in the switch. The switch hardware is only a little more complex. The hardware shown in Figure 4.6 must be added to Figure 3.1 to obtain the Allowed bits in the new switch. Remember that in any routing strategy, some logic is also required to manage the three mechanisms the fault-tolerant methodology is composed of.

5. Evaluation of FTFIR. In this section, we evaluate the Fault Tolerant FIR strategy. We are interested in analyzing its fault-tolerance, its performance, the amount of memory required and the routing delay.

The fault-tolerant degree and performance of the proposal is the same obtained by the fault-tolerant methodology (evaluated in [5]) since the proposed FT-FIR scheme is a different hardware implementation. This methodology is 7-fault tolerant. However, the percentage of tolerated fault combinations is greater than 99,9% up to 10 failures. Concerning performance, in the presence of failures, the performance degradation is small. As an example, network throughput degrades less than 5.5% when injecting 6 random failures in a 512-node Torus.

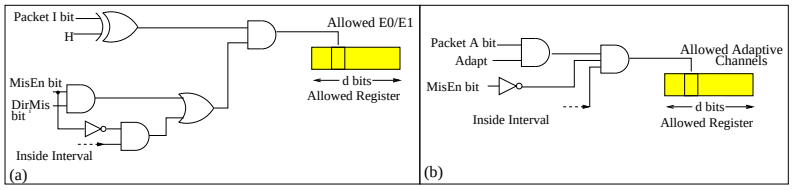


FIG. 4.5. *(a). When using misrouting, if the first direction indicated in the packet subheader corresponds to the direction of the current output channel, then the corresponding escape channel is allowed. (b). The adaptive channels are not allowed, If the packet must be misrouted in the current routing phase (MisEn=1).*

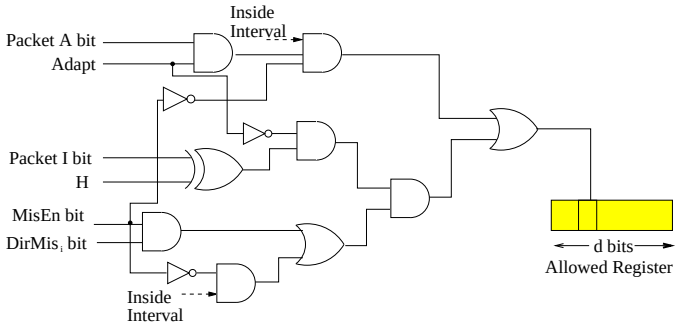


FIG. 4.6. *Combination of Figures 4.5.(a) and 4.5.(b): Allowed bit taking into all the fault-tolerant info contained in the packet header.*

Now we compare the amount of memory required at the switches by the FT-FIR strategy and an implementation based on forwarding tables[8]. Assume that we have a network composed of N nodes, build with switches with d ports. The links attached to each port may be split into up to v virtual channels. Finally, the routing algorithm offers a maximum of r routing options. The FT-FIR approach needs to associate five configuration registers with each virtual channel, three of them (FI, LI and MR) of size $\log(N)$ bits and two (RRR and BR) of size $d \times v$ bits. No matter if the routing is deterministic or adaptive. In addition, two configuration registers are associated to the whole switch, Adaptive and Escape Registers, of size $d \times v$ bits in order to provide fault-tolerance. Therefore, the total number of bits required to implement the FT-FIR strategy is $C_{\text{FTFIR}} = d \times v \times (3 \times \log(N) + 2 \times d \times\ v) + 2 \times d \times\ v$ bits. So, its cost remains being $O(\log(N))$ as in FIR. On the other hand, routing based on forwarding tables requires a table with as many entries as the number of nodes, and each entry must contain the port(s) returned by the routing function. Hence, the cost of this alternative is $C_{\text{FT}} = N \times \log(d \times v) \times r$ bits in each switch, which means that the cost is $O(N)$, and not scalable with the network size.

Routing delay of FIR is given by the time required to check if the destination address is inside the interval (all the ports make these comparisons concurrently) thus obtaining the Allowed bits plus the time to merge these bits with the routing restrictions (RRR). In the FT-FIR, this delay is slightly increased by the additional constraints of the fault-tolerant methodology. However, as part of this hardware may work in parallel with the interval comparison, the increase in routing delay will be small. On the other hand, some hardware would be required to support the fault-tolerance mechanism in a conventional routing scheme. This additional hardware will also increase the routing delay.

6. Conclusions. This paper proposes Fault-Tolerant Flexible Interval Routing (FT-FIR), a fault-tolerant adaptive routing strategy for commercial switches, . The strategy provides both, fault-tolerance and scalable routing for commercial switches in regular direct topologies. It is scalable because it requires relatively few amount of hardware and memory. It uses only 7 registers per each virtual channel to route packets, so the total requirements of memory is $O(\log(N))$. Moreover, it tolerates a relatively large number of faults while inflicting a minimal decrease of performance in the presence of faults.

[8]The fault-tolerant information is stored in the source nodes, but not at the switches.

REFERENCES

[1] C. Carrion, R. Beivide, J. A. Gregorio, and F. Vallejo. *A Flow Control Mechanism to Avoid Message Deadlock in K-ary N-Cube Networks*. Forfth International Conference on High Performance Computing, pp. 332-329, December 1997.

[2] W. J. Dally, Virtual-channel flow control, *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194-205, March 1992.

[3] W. J. Dally and H. Aoki. *Deadlock-free adaptive routing in multicomputer networks using virtual channels*. IEEE Trans. on Parallel and Distributed Systems, vol 4, no 4. pp 466-475, April 1993.

[4] J. Duato, S. Yalamanchili and L. Ni. Interconnection Networks. An Engineering Approach. *Morgan Kaufmann*, 2004.

[5] M. E. Gómez, J. Duato, J. Flich, P. López, A. Robles, N. A. Nordbotten, T. Skeie, and O. Lysne. *A New Adaptive Fault-Tolerant Routing Methodology for Direct Networks*, in Proc. International Conference on High Performance Computing, 2004.

[6] M. E. Gómez, P. López, J. Duato. *A Memory-Effective Routing Strategy for regular Interconnection Networks*, to appear in Proc. Int. Parallel and Distributed Processing Symposium, 2005. Best Paper Award in the Architecture Track.

[7] N. Santoro and R. Khatib. Routing without routing tables. *Tech. report SCS-TR-6, School of Computer Science*, Carleton University, 1982. Also as: Labelling and Implicit Routing in Networks, Computer Journal 28(1), 1985, pp. 5-8.

Edited by: M. Tudruj, R. Olejnik.
Received: February 24, 2006.
Accepted: July 30, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 61–73. http://www.scpe.org
ISSN 1895-1767

AFPAC: ENFORCING CONSISTENCY DURING THE ADAPTATION OF A PARALLEL COMPONENT

JÉRÉMY BUISSON*, FRANÇOISE ANDRÉ†, AND JEAN-LOUIS PAZAT‡

Abstract. Grid architectures are execution environments that are known to be at the same time distributed, parallel, heterogeneous and dynamic. While current tools focus solutions for hiding distribution, parallelism and heterogeneity, this approach does not fit well their dynamic aspect. Indeed, if applications are able to adapt themselves to environmental changes, they can benefit from it to achieve better performance. This article presents Afpac, a tool for designing self-adaptable parallel components that can be assembled to build applications for Grid. This model includes the definition of a consistency criterion for the dynamic adaptation of SPMD components. We propose a solution to implement this criterion. It has been evalued using both synthetic and real codes to exhibit the behavior of several proposed strategies.

Key words. dynamic adaptation, consistency, parallel computing

1. Introduction. Grid is an architecture that can be considered as a large-scale federation of pooled resources. Those resources might be processing elements, storage, and so on; they may come from parallel machines, clusters, or any workstation. One of the main properties of Grid architectures is to have variable characteristics even during the lifetime of an application. Resources may come and go; their capacities may vary. Moreover, resources may be allocated then reclaimed and reallocated as applications start and terminate on the Grid. To sum up, Grid is an architecture that is at the same time distributed, parallel, heterogeneous and dynamic. This is why we think distributed assemblies of parallel self-adaptable components is a suitable model for Grid applications.

In our context, we just see the component as a unit that explicitly specifies the services it provides ("provide ports") and the ones it requires ("use ports"). Required resources are one kind of required services. Service specifications can be qualitative (type specification); they can also be quantitative (quality of the provided or used services). A parallel component is simply a component that encapsulates a parallel code. For example, GridCCM [15] extends CORBA Component Model to support parallel components. A self-adaptable component is a component that is able to modify its behavior depending on the changes of the environment: it may use different algorithms that use services differently and provide different qualities of services. The choice of the algorithm is a component-dependent problem and cannot be solved in a general way at some middleware level. Nevertheless, some generic mechanisms exist in adaptable components that are independent of the component itself. Our research focuses on the design of an adaptation framework for parallel components that provides all those common mechanisms. One of the main problems consists in providing a mechanism for choosing a point in the execution to perform the adaptation in a consistent manner. This article aims at addressing this problem.

Section 2 presents our model of dynamic adaptation for parallel codes to set the context of this article up. Section 3 gives few mathematical definitions that are useful in the remaining text of this paper. Sections 4 and 5 describe how the future of the execution path is predicted to choose the adaptation point at which the adaptation code is inserted. Section 6 discusses the results obtained from our experiments. Section 7 compares our work to others in the area of dynamic adaptation. It also shows similarities with computation steering and fault tolerance. Finally, section 8 concludes this paper and presents some perspectives.

2. Model of self-adaptable components. The dynamic adaptability of a component is its ability to modify itself according to constraints imposed by the execution environment on which it is deployed. It aims at helping the component to give the best performance given its allocated resources. Moreover, it makes the component aware of changes in resource allocation. Resources may be allocated and reclaimed dynamically during the lifetime of the component.

2.1. Structure of a self-adaptable component. With our model of dynamic adaptation Dynaco [3], parallel self-adaptable components can split in several functional boxes (some of them being independent of the

*IRISA/INSA de Rennes, Campus de Beaulieu, 35042 Rennes, France (jbuisson@irisa.fr)
†IRISA/Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France (fandre@irisa.fr)
‡IRISA/INSA de Rennes, Campus de Beaulieu, 35042 Rennes, France (pazat@irisa.fr)

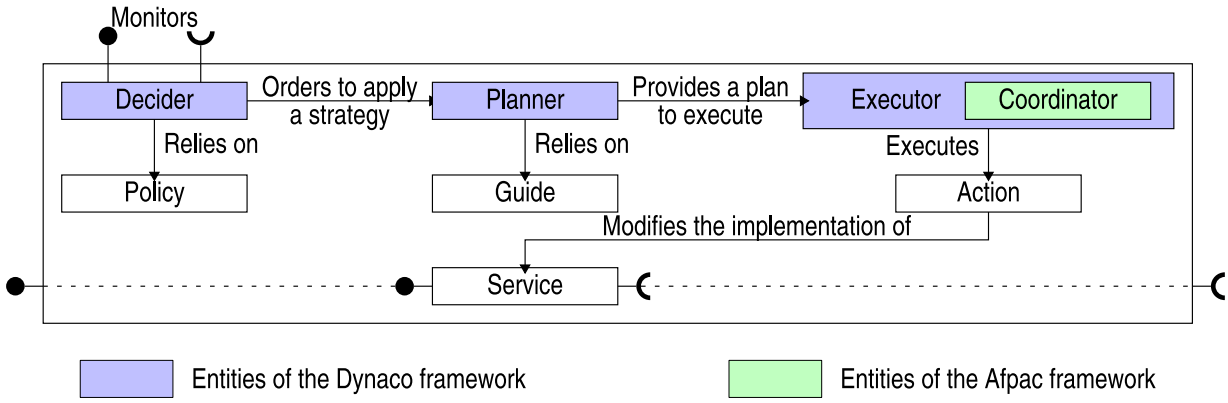


FIG. 2.1. *Model of a parallel self-adaptable component*

component itself). Figure 2.1 shows the architecture of a parallel self-adaptable component. A short description of the boxes follows.

The **monitor**s provide the information service about the environment of the component. They should offer two interfaces: a subscription-based event source and a query interface. Monitors are not restricted to collect information about the outside world of the component: the component itself and its activity can be relevantly monitored (for example for estimating its current service rate). Furthermore, nothing prevents the component from being one of its own monitors.

The **service** of the component is the functionality it provides. The service can be implemented by several algorithms that might be used to solve a single problem. Each of these algorithms uses differently the resources; thus, there is one preferred algorithm for each configuration of the resources allocated to the component.

The **action**s are the ways to modify the component. They modify the way the service is implemented, replacing the algorithm with another one, adjusting its parameters or anything else. Typically, actions use reflective programming techniques to affect the component.

A **strategy** is an indication of the way the component should behave: it states the algorithm that should be executed and the values that its parameters should adopt. The **decider** makes the decisions about adaptability. It decides when the component should adapt itself and which strategy should be adopted. In order to decide, it relies on the policy and on information provided by monitors. The **policy** describes component-specific information required to make the decisions. For example, the policy can be an explicit set of rules or the collection of the performance models of the algorithms implemented by the component.

A **plan** is a program whose instructions are invocations of actions and control instructions. Thus, plans are programs that modify the component. The **planner** establishes an adaptation plan that makes the component adopt a given strategy. To do so, it relies on the guide. The **guide** gives component-specific information required to build plans. It can consist in predefined plans; or in specifications of how actions can be composed (their preconditions and postconditions for example).

The **executor** is the virtual machine that interprets plans. In addition to forwarding invocations to actions, it is responsible for concerns such as atomicity of plan execution. The executor embeds a coordinator for synchronizing the execution of the plan with the processes that execute the service. To do so, the **coordinator** chooses a point (an instantaneous statement that annotates a special state) within the execution path of the service in order to insert the action.

In the context of parallel components, the service can be implemented either sequentially (exactly one execution thread) or in parallel (several communicating execution threads). In the latter case, the coordinator must implement a parallel algorithm.

Our parallel adaptable components are split in 3 parts. The decider, planner and executor form the DYNACO framework (in blue): they provide the general functionalities of dynamic adaptation. The coordinator is part of the AFPAC framework (in green) that is specific to parallel components. Finally, service, actions, policy and guide are specific to the component: they are not included in any framework.

2.2. Contracts between components. Because the quality of the services used by a component is of greater importance to the adaptation, we attach to each port link a contract that describes the effective quality

of the provided service. Moreover, for the sake of uniformity, our model abstracts resources as "provide ports" of some component called the environment. The execution environment of a component is thus completely described by the contracts attached to its "use ports". Consequently, any change in the execution environment is reflected by a change in those contracts, triggering the adaptation.

Contracts are dynamically negotiated and renegotiated: a component negotiates sufficient quality of service with its subcontractors in order to respect the contracts with its clients; it negotiates with its clients the quality of the provided services according to the ones contracted with its subcontractors. In that way, any renegotiation of one contract automatically propagates to the entire assembly of components (the application) to adjust the quality of all the provided services of all components. Consequently, it propagates the trigger of the adaptation to the components that require it.

2.3. Performance model. The goal of dynamic adaptation is to help the component to give the best performance. Thus, the model assumes that adaptation occurs when the component runs a non-optimal service given its allocated resources. This means that the completion time is worse than it could be. In a general way, the conceptual performance model of a single-threaded component that adapts itself is shown by figure 2.2. The x-axis is the execution flow of the component with adaptation points labelled from 1 to 5; the curve gives the expected completion time if the component adapts itself at the corresponding point in its execution flow. The lower the expected completion time is for a given adaptation point, the better that adaptation point is.

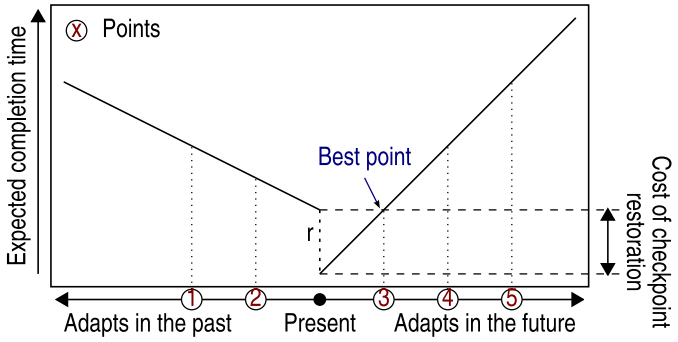


FIG. 2.2. *Conceptual performance model of an adapting component*

The adaptation can occur in either the future or the past of the current state. In the case of adapting in the past, it requires the restoration of a checkpoint (that has a cost r on the curve). At the extremes, adapting in the future is adapting once the execution has ended (or equivalently not adapting at all); while adapting in the past consists in restarting from the begining (purely static adaptation). It appears that whatever the search direction, the best point will always be the nearest one from the current state. In figure 2.2, the best adaptation point is the one labelled 3.

In the case of a parallel component, several curves are superimposed. Because the threads of the component are not necessarily synchronized, the curves are slightly shifted (the present time is not at the same distance of adaptation points for all threads). The global completion time is thus the maximum of the ones for all threads. In this case, the best point can be in the future for some threads and in the past for some others.

In this paper, we arbitrarily chose to consider only the search direction towards the future of the execution. In the case of a parallel component, we look for a global point that belongs to the future of all threads.

2.4. Consistency model. In the case of a parallel service, the coordinator must enforce the consistency of the adaptation. We defined one consistency model, which we called the "same state" consistency model. In this model, the adaptation is said to be consistent if and only if all threads execute the reaction from the same state (this logical synchronization does not require an effective synchronization of the execution threads).

This model only makes sense in the case of SPMD codes (parallel codes for which all threads share the same control flow graph). It has been further discussed in [5].

2.5. Scenario of dynamic adaptation. From time to time, possibly due to some external event, the decider determines whether the component should adapt itself according to the policy. It thus orders the planner to find a suitable plan for achieving the decided strategy. This plan is sent to the executor for execution. When the executor needs to execute an action in the context of the parallel execution of the service, it orders the

coordinator to choose an adaptation point. This point is the next one in the future of the execution path. When the threads of the service reach that adaptation point, their control flow is hooked in order to insert the execution of the requested action.

In this paper, we will concentrate on the coordinator. Section 4 describes how the point can be computed; section 5 depicts how an algorithm can be built. In the remaining text of the document, we will call "candidate points" the points at which a reaction can be executed. The one that has been chosen by the coordinator will be called the "adaptation point".

3. Definitions. This section recalls few definitions about partial orders that are useful in understanding the remaining text of this paper.

DEFINITION 3.1 (poset). *A poset (or partially-ordered set) is a pair (P, R) where P is a set and R is a binary relation over P that is (1) reflexive, (2) antisymmetric and (3) transitive.*

$$\begin{array}{ll} (1) & \forall x \cdot (x \in P \Rightarrow xRx) \\ (2) & \forall x, y \cdot \big(x, y \in P^2 \Rightarrow ((xRy \wedge yRx) \Rightarrow x = y)\big) \\ (3) & \forall x, y, z \cdot \big(x, y, z \in P^3 \Rightarrow ((xRy \wedge yRz) \Rightarrow xRz)\big) \end{array}$$

DEFINITION 3.2 (supremum). *Given a poset (P, R), the supremum $\sup(S)$ of any subset $S \subset P$ of P is the least upper bound u of S in P such that: (1) u succeeds all elements of S and (2) u precedes any element v of P succeeding all elements of S*

$$\begin{array}{ll} (1) & \forall x \cdot (x \in S \Rightarrow xRu) \\ (2) & \forall v \cdot (v \in P \Rightarrow ((\forall x \cdot (x \in S \Rightarrow xRv)) \Rightarrow uRv)) \end{array}$$

DEFINITION 3.3 (infimum). *Symmetrically to the supremum, given a poset (P, R), the infimum $\inf(S)$ of any subset $S \subset P$ of P is the greater lower bound l of S in P such that:*

$$\forall x \cdot (x \in S \Rightarrow lRx) \wedge \forall v \cdot (v \in P \Rightarrow (\forall x \cdot (x \in S \Rightarrow vRx)) \Rightarrow vRl)$$

The infimum of S in the poset (P, R) is the supremum of S in $\big(P, R^{-1}\big)$.

DEFINITION 3.4 (lattice). *A lattice $\mathcal{L}$ is a poset (P, R) such that for any pair $x, y \in P^2$, both the supremum $\sup(\{x, y\})$ and the infimum $\inf(\{x, y\})$ exist. This defines two internal composition laws:*

$$\begin{array}{ll} \sup(\{x, y\}) = x \vee y & \textit{(join)} \\ \inf(\{x, y\}) = x \wedge y & \textit{(meet)} \end{array}$$

4. Computing the future. The coordinator needs to find a global point such that it represents the same state for all threads. As it restricts consideration to points in the future of execution paths this point should be the first one in the future for all threads, given the assumption on the performance model depicted in § 2.3.

4.1. Local prediction of the next point. For each thread, the coordinator needs to find the next candidate point in the future of the execution path. This point would be the one at which the coordinator is going to execute the reaction in the case of a single-threaded component.

4.1.1. General schema. An execution path can be seen as a path in the control flow graph that models the code. If loops are unrolled (for example by tagging nodes with indices in iteration spaces), the control flow graph defines a "precedes" partial order between the points of a code. Given this relation, each thread of the parallel component is able to locally predict its next point at least in trivial cases, when no conditional instruction is encountered (in this case, nodes in the control flow graph have at most one successor).

4.1.2. Uncertain predictions. In the general case, predicting the future of an execution path is undecidable due to conditional instructions: their behavior is unpredictable since it depends on runtime computations. This occurs for both conditions and loops, which appear as nodes having several successors in the control flow graph.

Three strategies may be used when such a node is encountered during the computation of the adaptation point.

- **Postpone.** The computation of the adaptation point can be postponed until the effective behavior of the instruction is known. Once the unpredictable instruction has been executed, the computation of the adaptation point resumes with a new start point (the one that is being entered).
- **Skip.** The computation of the point can move forward in the control flow graph until all control subflows merge. In this case, the prediction jumps over loops and conditions.
- **Force.** The behavior of the conditional instruction can be guessed when additional static information is available. With this strategy, one branch of the conditional instruction is assumed accordingly to some application-dependent knowledge; then some application-level mechanism enforces that the execution path respects this assumption.

Two examples that follow illustrate the usage of the "force" strategy. Firstly, it is possible for some loops to insert unexpected empty iterations; in this case, it can be guessed that the conditional instruction will execute the branch that stays in the loop; this behavior can be enforced by making one more empty iteration on-demand. Secondly, if the code includes assertions to detect error cases, it could be reasonably assumed that those assertions hold; in this case, it can be guessed that the corresponding conditional instruction will execute the branch leading to error-free cases. As mispredictions exactly match detected runtime errors, error handlers invoked upon assertion failures can be used to handle effects of mispredictions.

The above examples exhibit two interpretations of the "force" strategy. In the first example, the coordinator changes the execution behavior of the component in order to fit its requirements. The coordinator uses the static knowledge about the component to ensure that this change does not have any semantic impact on produced results. In the second example, the coordinator let the component execute its normal behavior and simply exploits static knowledge to make its prediction.

4.2. Characterization of the chosen global point. The point that should be chosen is the supremum u, if it exists, of the set S of the next local points for each thread with respect to the "precedes" partial order (noted $\preceq$). Indeed, the property $\forall x \cdot (x \in S \Rightarrow x \preceq u)$ guarantees that u is in the future of all threads; the property $\forall v \cdot (v \in P \Rightarrow ((\forall x \cdot (x \in S \Rightarrow x \preceq v)) \Rightarrow u \preceq v))$ means that u is the first point amongst those in the future of all threads.

Even if loops are unrolled, the control flow graph is not a lattice as it may contain patterns such as the one of figure 4.1(a). In this example, the supremum $\sup(\{P1, P2\})$ does not exist. This pattern captures a subset of conditional instructions. The three strategies that has been identified for locally predicting the next point can be used to work-around non-existence of the supremum.

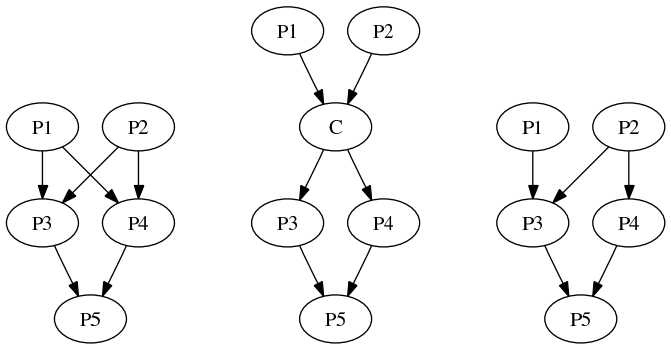
- **Postpone.** Postponing the computation of the adaptation point until the unpredictable instruction has been executed can be seen as inserting a special node that models that instruction in the control flow graph. For example, in figure 4.1(b), a node C has been inserted to represent the conditional instruction. When such a node is chosen, a new round for finding a satisfying point is started once the corresponding instruction has been executed.
- **Skip.** Skiping the conditional block is equivalent to compute the supremum within the subset defined by:

$$\{a | \forall y \cdot ((\forall x \cdot (x \in S \Rightarrow x \preceq y)) \Rightarrow (a \preceq y) \vee (y \preceq a))\}$$

  This set contains the points that are related to all successors of all points in S. Those points are guaranteed to be traversed by any execution thread starting from any point in S. The definition of that set ensures that upper bounds of S in that set are totally ordered by $\preceq$. Consequently, the supremum of S in that set ordered by $\preceq$ exists if at least one upper bound exists. In the given example, it restricts to compute the supremum $\sup(\{P1, P2\})$ in $\{P1, P2, P5\}$; this supremum exists.
- **Force.** Forcing one branch of a conditional instruction removes all out edges but one of the corresponding node. This is sufficient to make the supremum exist. For example, on figure 4.1(c), the conditional instruction that follows $P1$ has been forced such that it always leads to $P3$.

Each conditional instruction may use a distinct strategy. In such a case, several strategies are combined.

4.3. Comparison of the strategies. These three strategies have their own drawbacks. The "postpone" one shortens the time between choosing a point and reaching the chosen point. It may result in an increase of the risk that the execution thread tries to get through the locally chosen point before it has been globally either confirmed or evicted. The "skip" one chooses an adaptation point further in the future of the execution path (and possibly falls back at the end of the code). The "force" one modifies the code of the execution thread.



(a) Pattern which prevents the supremum from existing

(b) Inserting nodes for conditionals

(c) Removing branches of conditionals

FIG. 4.1. *Example of control flow graph with non-existent supremum*

5. Building an algorithm. In order to solve the problem of finding a point in the case of a parallel service, an algorithm has been designed.

5.1. Identification of the candidate points. Having a good identification system for the points is a key issue. Indeed, as § 4.2 has shown, the case of parallel self-adaptable components requires the computation of the supremum of a set of points. This requires to be able to compare easily candidate points with respect to the "precedes" partial order. However, with naive point identification, deciding whether a candidate point precedes, succeeds, or is not related to another one requires the computation of the transitive closure of the control flow graph with unrolled loops. The same apply for the join composition law. This is why a smarter identification system for the points is needed.

5.1.1. Description of the identification system. We think that the good representation to use is a tree view of the hierarchical task graph. Figure 5.1 gives an example algorithm that has been annotated with candidate points; figure 5.2 shows the control flow graph between annotated points; figure 5.3 shows the corresponding tree view. The edges of the tree are labelled as follows:

- out edges of loop nodes are labelled with the value of the indice within the iteration space of the loop;
- out edges of condition nodes are labelled with either the symbol "then" (the condition is true) or the symbol "else" (the condition is false);
- other edges (out edges of block nodes) are labelled with the execution order number in the control flow.

```
Algorithm gcd(a, b):
    loop until ((a mod b) = 0)
        if (a < b) then
                // candidate point P1
                tmp ← a
                a ← b
                b ← tmp
            else
                // candidate point P2
                a ← (a mod b)
        end if
    end loop
    // candidate point P3
    return (b)
```

FIG. 5.1. *An examplary algorithm*

Considering the tree view of the hierarchical task graph, one can see that the nodes of the control flow graph are the leaves of the tree. Each candidate point is identified by the sequence composed of the labels of

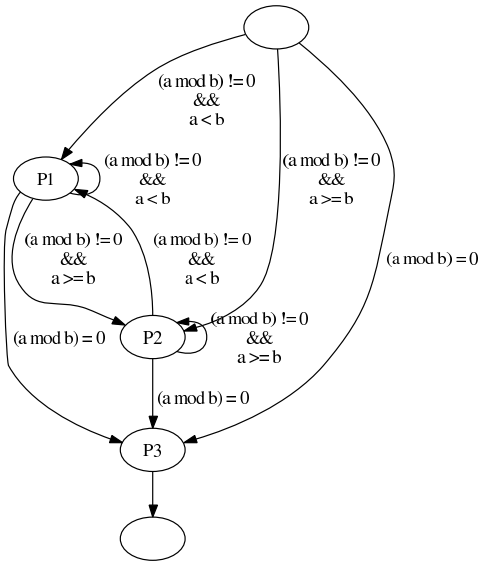


FIG. 5.2. *Control flow graph of the algorithm 5.1*

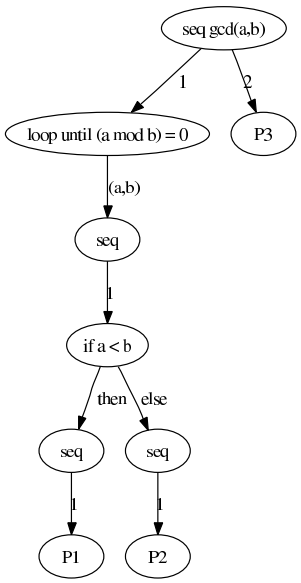


FIG. 5.3. *Tree view of the hierarchical task graph of the algorithm 5.1*

the edges traversed by the path from the root to the leaf corresponding to it. For example, the node "$P3$" is identified by $\langle 2 \rangle$; the node "$P1$" is identified by the sequence $\langle 1, (a,b), 1, \text{else}, 1 \rangle$.

5.1.2. Order on points. The set of edge labels in the tree is $\mathcal{E} = \mathbb{N} \cup \mathcal{I} \cup \{then, else\}$, where $\mathbb{N}$ is the set of execution order numbers (natural numbers) and $\mathcal{I}$ the iteration space partially ordered by an application-specific relation $\preceq_{\mathcal{I}}$. We define the following partial order over the set $\mathcal{E}$ of edge labels of the tree:

$$\forall x, y \cdot \left(x, y \in \mathcal{E}^2 \Rightarrow \left(x \preceq_{\mathcal{E}} y \Leftrightarrow \left(\left(x, y \in \mathbb{N}^2 \wedge x \leq y\right) \vee \left(x, y \in \mathcal{I}^2 \wedge x \preceq_{\mathcal{I}} y\right)\right)\right)\right)$$

Notably, the two constants "then" and "else" are set not comparable to other values and values in iteration spaces (such as (a, b) in the example) are partially ordered by an application-specific relation.

The edges of the control flow graph (that represent the "precedes" partial order between points) are (graphically) transverse to those of the tree. Consequently, the lexicographical partial order $\preceq^n_{\mathcal{E}}$ on the sequences is equivalent to the partial order on points defined by the control flow graph. Thus, with this identification system, deciding whether a point precedes, succeeds, or is not related to another one can be computed as a direct lexicographical comparison (not requiring the computation of the transitive closure of the control flow graph anymore). Moreover, this ordering directly takes into account the loop indices in iteration spaces, whereas the control flow graph does not.

5.1.3. Discussion. This identification system makes sense only in the case of a parallel component in which all threads are allowed to have different dynamic behaviors (different behavior of conditionnal instructions and loop indices desynchronization). Otherwise, a simple counting identification system is sufficient. Furthermore, in the case of a non-parallel component, no identification system is required.

In the above description of the identification system, in the tree view of the hierarchical task graph, leaves are candidate points and internal nodes are "sequence", "loop" and "if" control structures. This restricts the expressivity for writing programs. Function calls can be represented as leaves that connect to the root the target function tree, as long as the function is bound statically. Thanks to that connection, the sequence identifying a point includes the complete call stack used to reach that point. Problems arise with runtime function binding upon calls (for example calls through pointers) and with "goto" like instructions. In those cases, the proposed order relation does not respect the control flow graph. As programs can be rewritten in order to respect those restrictions, further discussion about control structures is beyond the scope of this paper.

5.2. Computation of the local adaptation point. Because we restricted consideration to candidate points in the future of the execution path, the local adaptation point is the next point in the path. Computing

this point simply consists in following the edges of the control flow graph. This corresponds to a depth-first traversal of the tree from the left to the right. This traversal begins at a start point that is at least the one being executed at the time of the computation.

5.3. Instrumentation of the code. In order to predict the next candidate point in the future of the execution path, the coordinator must be able to locate the actual execution progress in the space of the candidate points. As it must be able to build the sequence identifying the candidate point that is being executed by the execution thread, the coordinator requires an instrumentation of the code.

The instrumentation of the code consists in inserting some pieces of code at each node of the tree view of the hierarchical task graph as presented at the preceding section. The figure 5.4 shows an example of an instrumented algorithm.

```
Algorithm instrumented_gcd (a, b):
    enter_function (gcd (a, b))
    enter_loop (a, b)
    loop until (leave_loop ((a mod b) = 0))
        iteration_loop (a, b)
        if (enter_condition (a < b)) then
                candidate_point (P1)
                tmp ← a
                a ← b
                b ← tmp
            else
                candidate_point (P2)
                a ← (a mod b)
        end if
        leave_condition ()
    end loop
    candidate_point (P3)
    return (leave_function (b))
```

FIG. 5.4. *Instrumented version of the algorithm 5.1*

The process of inserting the instrumentation statements in the source code of the component can be largely automated using aspect-oriented programming (AOP [13]) techniques. We have proposed in [18] a static aspect weaver whose join points are the control structures. This weaver has been successfully used to insert the instrumentation statements. It relies on candidate points, which are placed by hand by the developer, to detect which control structures need to be instrumented (those that contain at least one candidate point).

5.4. Protocol for computing the supremum. Many trivial protocols might be designed to compute this supremum (for example, centralized). Many already exist to compute a maximum (in particular for leader election); they can easily be modified to compute a supremum instead of a maximum. However, we think that a specific protocol suits better this case.

5.4.1. Motivation for designing a specific protocol. The full set of the local adaptation points is not always necessary to compute the supremum. The idea is to superimpose the agreement protocol and the local computations of the next candidate point in order to exempt some of the execution threads from computing their local adaptation point. In particular, it is interesting to exempt those whose prediction is postponed due to conditional instructions. Indeed, it allows the computation of the local adaptation point to resume sooner with a start point that is further in the execution path. Moreover, superimposing the protocol and the computation evicts the earliest adaptation points sooner than if the whole set of local adaptation points must be computed. This avoids situations in which the execution thread tries to execute through a locally chosen point that has not been either confirmed or evicted yet.

5.4.2. Informal description of the protocol. The key idea of the protocol is to let the threads negotiate. Each thread can propose to the others its view of what the common adaptation point is, namely the point it has chosen locally. If it has not chosen one (because of unpredictable instructions such as conditions), it can

give clues to the other threads. These clues can be for example the root node of the subtree representing the conditional block. When a thread receives a proposition or a clue, it may take different actions depending on how it compares to its own proposition.

- If its own proposition is better (that is to say in the future of the received one), it rejects the received proposition.
- If the received one is better, it is adopted.
- If the two propositions are not comparable, the thread retracts its proposition and computes a new one that takes into account the two previous ones. The computation starts at the point at which the control subflows merge (the supremum of the two points).
- If both propositions are the same, the thread agrees with the sending thread.

The protocol also progresses when an execution thread provides new information. For example when the functional thread executes a conditional instruction, it may retract its clue and propose some new information.

The protocol ends when all threads agree with each other's and when the agreement is on a true proposition (not just a clue). This means that they all have adopted the same point.

5.4.3. Anatomy of the protocol. We have designed a specific protocol to solve this problem. This protocol is based on a unidirectional ring communication scheme. Each thread confronts its proposition with the one of its predecessor and sends it only to its successor in the ring. The end of the protocol, namely the reach of the agreement, is detected using a Dijkstra [6]-like termination algorithm. Our protocol distinguishes strong propositions (values that can be chosen by the agreement) from weak propositions (values that forbid the agreement to conclude, previously called clues). The protocol allows threads to retract their proposition. It strictly defines the criteria of the consistency of retraction.

Our protocol is pessimistic. If an execution thread tries to get through a point thought to be the chosen one, the algorithm refuses to speculate whether it should allow the execution thread to continue or execute the reaction. It rather suspends the execution thread until it gets certainty. As a result, because the "postpone" strategy eases such situations (as described in § 4.3), this strategy tends to increase the risk of suspending the execution thread.

6. Experiments. The experiments presented here aim at comparing the strategies described at § 4.1.2 with regard to functional code suspension and delay before the execution of the reaction. The goal is to exhibit and validate the expected behaviors described in § 4.3. This characterization would help developers to choose the right strategy depending on their code.

Three experiments have been made: synthetic loop code, synthetic condition code and NAS Parallel Benchmark 3.1 [14] FFT code. Only the results of the latter will be presented and discussed.

6.1. Experimental protocol. The experimental protocol consists in trigerring adaptation many times at random intervals (with uniform distribution to avoid implicit synchronization between the functional code and the adaptation trigger) while the code is being executed by a 4 processors cluster. Experiments result in two data series: one indicates for each adaptation the time elapsed between the trigger and the effective execution of the reaction; the other gives the time during which the functional thread has been suspended while choosing the point at which to adapt. For each data series, one curve is drawn that gives an approximate measure of the density of samples for each observed value.

6.2. Observations. Figures 6.1 and 6.2 show the results with the FFT code. This experiment aims at comparing the "postpone" and "skip" strategies for conditional instructions within a main loop. The main loop has the "force" strategy so that the synchronization and communication statements induced by our algorithm are exclusively related to the inner instructions. This prevents distortions of the observations.

Figure 6.1 shows that the "postpone" strategy tends to select an adaptation point that arrives sooner in the execution path than the "skip" strategy. Conversly, figure 6.2 shows that the "postpone" strategy tends to suspend the functional code for a longer time than the "skip" strategy. Indeed, the "postpone" strategy leads to higher density of samples up to about 0.5 s of functional code suspension. However, this observation has to be balanced as it appears that both strategies do not suspend the functional code in most of the samples.

6.3. Discussion. These results and those we obtained with synthetic codes confirm the behaviors we expected in section 4.3. Namely, the choice between the "postpone" and the "skip" strategies, is a trade-off between the precision of the prediction and the risk to uselessly suspend the functional code.

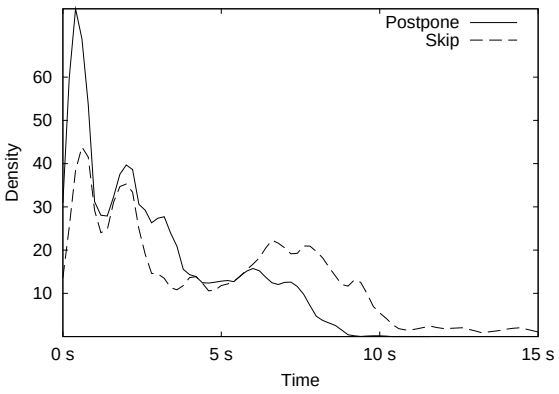


FIG. 6.1. *Density curve for adaptation delay*

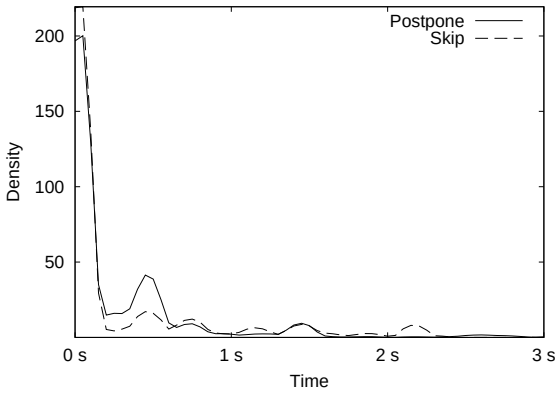


FIG. 6.2. *Density curve for suspension time*

The observations are explained by the fact that the "postpone" strategy tends to delay the agreement in order to choose a more precise point (closer to the current state), in comparision to the "skip" strategy. It makes the "postpone" strategy increase the risk for other threads to reach that point before the agreement is made, and thus to suspend their execution. On the other side, the "skip" strategy tends to choose a point that is further in the future of the execution path; it results in a lower risk for other threads to reach that point before the agreement is made and thus to suspend their execution.

Consequently, if the time between a conditional instruction and the following point is large enough to balance the delay of the agreement, the "postpone" strategy should be preferred as it results in a sooner adaptation. Otherwise, the "skip" strategy should be used in order to avoid the suspension of the execution thread.

7. Related works and domains.

7.1. Other projects on dynamic adaptation of parallel codes. Most of the projects that faced the problem of dynamic adaptation such as DART [16] do not consider parallelism. Other projects such as SaNS [8] and GrADSolve [17] build adaptive components for Grid. Those projects target adaptation only at the time of component invocation. This supposes a fine-grain encapsulation in components; otherwise, if methods are too long, adaptation is defeated because it cannot occur during execution. Within the context of Commercial-Off-The-Shelf (COTS) component concept, sufficiently fine-grained encapsulation for the dynamic adaptation to make sense only at invocation level does not appear valuable enough; whereas sufficiently coarse-grained encapsulation for valuable COTS components are not able to really dynamically adapt themselves if they do so only at each invocation. This is why we think that components should be able to adapt themselves not only at method calls, but also during the execution of methods.

The Grid.It project [2] also faces the problem of dynamically adapting parallel components. The followed approach is based on structured parallelism provided by high-level languages. The compiler generates code for virtual processes that are mapped to processing elements at run-time. An adaptation consists in dynamically modifying this mapping. This adaptation can be done within language constructs transparently to the developper. In this case, the consistency enforcement mechanism can rely on assumptions on the overall structure of the program as it is generated by a compiler.

PCL [1] focuses on the reflective programming tools for the dynamic adaptation of parallel applications. This project considers dynamic adaptation as modifications on a static task graph modeling the code and provides at runtime primitive operators on that graph. It gives a full framework for reflective programming that is useful to implement reactions. The problem of the coordination of the reaction and the execution threads of the functional code has been studied with PCL in [9]. Their approach to the problem of the coordination is similar to ours: the adaptation is scheduled to be executed at an adaptation point in the future of the execution path. However, the problem of dealing with unpredictable conditional instructions is not addressed. Furthermore, the consistency model is different from ours. In PCL, each reaction is triggered at any point of the specified region. For example, on figure 7.1, tasks B and D are not distinguished. According to the consistency model, in a two-threads component, if one is executing A and the other C, a "region in" reaction targeting region $RGN1$ is scheduled at the $A \rightarrow B$ edge for the first thread and at the $C \rightarrow D$ edge for the second one. We think that it is a better consistency model to

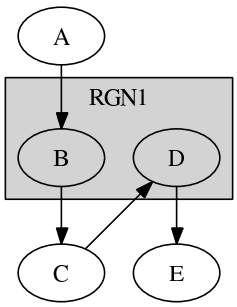


FIG. 7.1. *Example of a region in a PCL static task graph*

schedule the reaction at the $C \rightarrow D$ edge for both threads (our "same state" consistency). One solution might be to put B and D in two different regions; but in this case, PCL fails specifying that the reaction can be scheduled either before B or before D, but at the same edge for all threads. Furthermore, PCL is unable to handle the case of thread desynchronization in loops (one thread doing more iterations than another does) for regions being in such a loop. The simple region counter is not sufficient to solve this problem.

7.2. Resource management in Grid infrastructures. The dynamic characteristic of Grid architecture appears well accepted, as reflected by the presence of monitoring services such as Globus MDS. On the other side, job management usually considers that the set of resources allocated to one job remains constant for the whole lifetime of the job. Approaches such as AppLeS [4] and GridWay [12] show that even Grid applications can benefit from non-constant schedules. With those two projects, when the scheduler or the application detects that a better schedule can be found, the application checkpoints itself, then the job is cancelled and resubmitted with new resource constraints. Our approach is slightly different. We assume that the resource management system is able to modify job schedules without having to cancel and resubmit jobs. Thanks to this assumption, checkpointing and restarting the application is not required, potentially leading to lower cost for the adaptation in particular when the intersection before and after the rescheduling is not empty.

7.3. Computation steering. The problem of steering a parallel component is very close to our problem of adapting this component. Indeed, in both cases, we need to insert dynamically code at some global state. The EPSN project [10] aims at building a platform for online steering and visualization of numerical simulations. In [11], the authors describe the infrastructure they built to do the steering. Their structured dates is similar to our candidate point identification system. Both systems are based on similar representations of the code. Moreover, steering like dynamic adaptation tries to find the next candidate point in the future of the execution path. Thus, computation steering and dynamic adaptation may benefit from fusionning in a single framework.

7.4. Fault tolerance. Fault tolerance might be seen as a particular case of dynamic adaptation. Indeed, it is the adaptation to the "crash" of some of the allocated resources. However, things are not that simple. Fault tolerance encompasses two different problems: the recovery of the fault and the adaptation to the new situation. Only the second one really belongs to the problem of dynamic adaptation. Many strategies can be used to recover from faults. The closest one from dynamic adaptation is probably checkpointing in the sense it requires some global points. Whereas dynamic adaptation can look for a point in the future of the execution path, checkpointing requires a point in the past. Nevertheless, mechanisms required by dynamic adaptation may be useful to implement recovery strategies based on checkpointing. The instrumentation of the code can be used as hooks at which checkpoints can be taken. Indeed, it implicitly statically coordinates the checkpoints of all threads, avoiding the problem of computing a global consistent state. Reciprocally, dynamic adaptation can take advantage of the techniques that have been developped to checkpoint and restart computations in the area of fault-tolerance.

8. Conclusion. In this paper, we have briefly described the overall model we introduced to build self-adaptable parallel components. This paper essentially focuses on the coordinator functional box of the AFPAC framework. This coordinator is responsible for scheduling the reaction in the execution path of the several execution threads.

The protocol we designed follows a pessimistic approach with regard to the fact that the execution thread may go through a confirmed point. In the future of our work, we will study how optimistic approaches can be designed. For example, when the functional code reaches a point suspected to be chosen but not yet confirmed, it may continue its execution instead of waiting for the point to be either confirmed or evicted. If at the end that point is confirmed, the situation may be repaired by rolling-back the functional code.

Moreover, our coordinator searches adaptation points exclusively in the future of the execution path. However, this is an arbitrary choice that has no other justification than simplifying algorithms. In our upcoming researches, we will study how algorithms can be generalized in order to allow the coordinator to look for adaptation points in in both directions (future and past). Thanks to this, chosen adaptation points may be closer to the optimum.

Finally, our "same state" consistency model has to be extended to the case of non-SPMD components: we have already thought of another consistency model defined by a customizable correspondence relation between the candidate points of the threads.

As a longer term research, we are working on completing our architecture in order to build a complete platform. In particular, we are studying how generic engines can be used within our Dynaco framework. We are also investigating protocols for adaptation within assemblies of components. Endly, we are studying how a resource manager could take advantage of applications able to adapt themselves.

Acknowledgement. Experiments described in this paper have been done with the Grid 5000 French testbed `http://www.grid5000.fr/`

REFERENCES

[1] V. Adve, V. V. Lam, and B. Ensink, *Language and compiler support for adaptive distributed applications*, in ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001), Snowbird, Utah, June 2001.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo, *Components for high performance grid programming in the grid.it project*, in Workshop on Component Models andd Systems for Grid Applications, June 2004.

[3] F. André, J. Buisson, and J.-L. Pazat, *Dynamic adaptation of parallel codes: toward self-adaptable components for the grid*, in Workshop on Component Models and Systems for Grid Applications (Held in conjunction with ICS'04), June 2004.

[4] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, *Adaptive computing on the grid using apples*, IEEE Transactions on Parallel and Distributed Systems, 14 (2003), pp. 369–382.

[5] J. Buisson, F. André, and J.-L. Pazat, *Dynamic adaptation for grid computing*, in Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers), P. Sloot, A. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, eds., vol. 3470 of Lecture Notes in Computer Science, Amsterdam, Feb. 2005, Springer-Verlag, pp. 538–547.

[6] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, *Derivation of a termination detection algorithm for distributed computations*. retranscription de [7], 1982.

[7] ———, *Derivation of a termination detection algorithm for distributed computations*, Information Processing Letters, 16 (1983), pp. 217–219.

[8] J. Dongarra and V. Eijkhout, *Self-adapting numerical software for next generation application*, Aug. 2002.

[9] B. Ensink and V. Adve, *Coordinating adaptations in distributed systems*, in 24th International Conference on Distributed Computing Systems, Mar. 2004, pp. 446–455.

[10] *Epsn project*.

[11] A. Esnard, M. Dussere, and O. Coulaud, *A time-coherent model for the steering of parallel simulations*, in Europar 2004, Sept. 2004.

[12] E. Huedo, R. S. Montero, and I. M. Llorente, *The gridway framework for adaptive scheduling and execution on grids*, Scalable Computing: Practice and Experience, 6 (2005), pp. 1–8.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*, in Proceedins of European Conference on Object-Oriented Programming, M. Akşit and S. Matsuoka, eds., vol. 1241, Berlin, Heidelberg and New-York, 1997, Springer-Verlag, pp. 220–242.

[14] NAS, *Parallel benchmark*. `http://www.nas.nasa.gov/Software/NPB/`

[15] C. Pérez, T. Priol, and A. Ribes, *A parallel corba component model for numerical code coupling*, in Proc. 3rd International Workshop on Grid Computing, M. Parashar, ed., no. 2536 in Lect. Notes in Comp. Science, Baltimore, Maryland, USA, Nov. 2002, Springer-Verlag, pp. 88–99. Held in conjunction with SuperComputing 2002 (SC '02).

[16] P.-G. Raverdy, H. L. V. Gong, and R. Lea, *DART : a reflective middleware for adaptive applications*, in OOPSLA'98 Workshop #13 : Reflective programming in C++ and Java, Oct. 1998.

[17] S. Vadhiyar and J. Dongarra, *GrADSolve: RPC for high performance computing on the grid*, in Euro-Par 2003: Parallel Processing, H. Kosch, L. Bszrmnyi, and H. Hellwagner, eds., vol. 2790 of Lecture Notes in Computer Science, Springer-Verlag, Aug. 2003, pp. 394–403.

[18] G. Vaysse, F. André, and J. Buisson, *Using aspects for integrating a middleware for dynamic adaptation*, in The First Workshop on Aspect-Oriented Middleware Development (AOMD'05), ACM Press, Nov. 2005.

Edited by: M. Tudruj, R. Olejnik.
Received: February 24, 2006.
Accepted: July 30, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 75–86. http://www.scpe.org
ISSN 1895-1767

WEBCOM-G AND MPICH-G2 JOBS*

PADRAIG J. O'DOWD, ADARSH PATIL AND JOHN P. MORRISON†

Abstract. This paper discusses using WebCom-G to handle the management & scheduling of MPICH-G2 (MPI) jobs. Users can submit their MPI applications to a WebCom-G portal via a web interface. WebCom-G will then select the machines to execute the application on, depending on the machines available to it and the number of machines requested by the user. WebCom-G automatically & dynamically constructs a RSL script with the selected machines and schedules the job for execution on these machines. Once the MPI application has finished executing, results are stored on the portal server, where the user can collect them. A main advantage of this system is fault survival, if any of the machines fail during the execution of a job, WebCom-G can automatically handle such failures. Following a machine failure, WebCom-G can create a new RSL script with the failed machines removed, incorporate new machines (if they are available) to replace the failed ones and re-launch the job without any intervention from the user. The probability of failures in a Grid environment is high, so fault survival becomes an important issue.

Key words. WebCom-G, Globus, MPICH-G2, MPI, Grid Portals, Scheduling and Fault Survival

1. Introduction. The Globus Toolkit [1] has become the de-facto software for building grid computing environments and MPICH-G2 [2] uses Globus to provide a grid-enabled implementation of the MPI v1.1 standard. Being able to run unmodified legacy MPI code in this manner, strengthens the usability of grid computing for end users. However, there are some disadvantages/limitations to running MPICH-G2 jobs with RSL scripts (e.g., little or no support for job recovery after failure) and these will be discussed in detail in Section 3. This work investigates using WebCom-G to address some of these issues, such as automating the deployment, execution & fault survival of MPICH-G2 jobs [3, 4]. Other work [5] has investigated the use of WebCom-G for handling issues like fault survival with MPICH and running MPI jobs in a Beowulf cluster environment.

The remainder of this paper is organised as follows: Globus and its Execution Platform is discussed in Section 2. MPICH-G2 and how it uses Globus is described in Section 3. In Section 4, the WebCom-G Grid Operating System is presented, including an in-depth look at the WebCom-G components used in this work. In Section 5, the control of MPICH-G2 by WebCom-G is discussed and how WebCom-G can ensure the fault survival of MPICH-G2 jobs. In Section 6, some sample executions and results are presented to verify that the system operates correctly. Finally, Section 7 presents conclusions of this work.

2. Globus and its Execution Platform. Globus [1] provides the basic software infrastructure to build and maintain Grids. As dynamic networked resources are widely spread across the world, information services play a vital role in providing grid software infrastructures, discovering and monitoring resources for planning, developing and adopting applications. The onus is on the Information services to support resource & service discovery and subsequently use these resources and invoke services. Thus the Information services is a crucial part of any Grid.

An organisation running Globus hosts their resources in the Grid Information Service (GIS), running on a Gatekeeper machine. The information provided by the GIS may vary over time in an organisation. The information provider for a computational resource might provide static information (such as the number of nodes, amount of memory, operating system version number) and dynamic information such as the resources uncovered by the GIS, machine loads, storage and network information. Machines running Globus may use a simple scheduler or more advanced schedulers provided by Condor, LSF, PBS and Sun Grid Engine.

Typically users submit jobs to Globus (2.4) by means of a Resource Specification Language (RSL) script executing on the Gatekeeper, provided they have been successfully authenticated by the Grid Security Infrastructure(GSI). The RSL script specifies the application to run and the physical node(s) that the application should be executed on and any other required information. The Gatekeeper contacts a job manager service which in turn decides where the application is to be executed. For distributed services, the job manager negotiates with the Dynamically Updated Request Online Co-allocator (DUROC) to find the location of the requested service. DUROC makes the decision of where the application is to be executed by communicating with each machines' lower level Grid Resource Allocation Manager (GRAM). This information is communicated back to the job manager and it schedules the execution of the application according to its own policies. If no job manager

*The support of Science Foundation Ireland and Cosmogrid is gratefully acknowledged.
†Computer Science Dept., University College Cork, Ireland. ({p.odowd, adarsh, j.morrison}@cs.ucc.ie).

is specified, then the default service is used. This is usually the "fork" command, which returns immediately. A typical grid configuration is shown in Fig. 2.1.

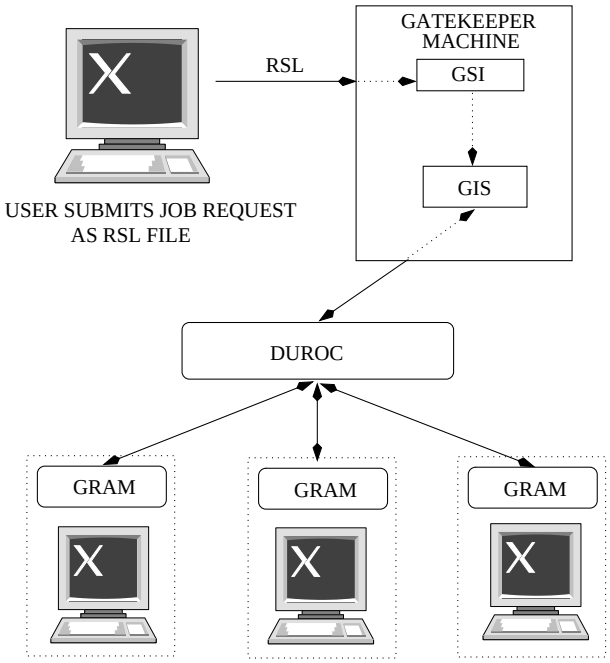


FIG. 2.1. *Globus 2.4 Execution model. A user generates an RSL script and submits it to the Gatekeeper. The Gatekeeper, in conjunction with DUROC and GRAM facilitate the distributed execution of requested services on the underlying nodes*

There are some disadvantages to using RSL scripts. Most notably, in a distributed execution if any node fails, the whole job fails and will have to be re-submitted by the user at a later time. There is no diagnostic information available to determine the cause of failure. Only resources known at execution time may be employed. There is no mechanism to facilitate job-resource dependencies. The resource must be available before the job is run, otherwise it fails. There is no in-built check-pointing support, although some can be programmatically included. This may not be feasible due to the particular grid configuration used. Also, RSL is only suited to tightly coupled nodes, with permanent availability. If any of the required nodes are off-line, the job will fail.

3. MPICH-G2. MPICH-G2 [2] is a grid-enabled implementation of the MPI v1.1 standard. It uses services from the Globus Toolkit to handle authentication, authorisation, executable staging, process creation, process monitoring, process control, communication, redirecting of standard input (& output) and remote file access. As a result a user can run MPI programs across multiple computers at different sites using the same commands that would be used on a parallel computer or cluster. MPICH-G2 allows users to couple multiple machines, potentially of different architectures, to run MPI applications. It automatically converts data in messages sent between machines of different architectures and supports multi-protocol communication by automatically selecting TCP for inter-machine messaging and (where available) vendor-supplied MPI for intra-machine messaging. According to [2], performance studies have shown that overheads relative to native implementations of basic communication functions are negligible.

As shown in Fig. 3.1, MPICH-G2 uses a range of Globus Toolkit services to address the various complex situations that arise in heterogeneous Grid environments. MPICH-G2 was created by creating a 'globus2' device for MPICH [6]. MPICH supports portability through its layered architecture. At the top is the MPI layer as defined by the MPI standards. Directly underneath this layer is the MPICH layer, which implements the MPI interface. Most of the code in an MPI implementation is independent of the underlying network communication system. This code, which includes error checking and various manipulations of the opaque objects, is implemented at the MPICH layer. All other functionality is passed to lower layers by means of the Abstract Device Interface (ADI). The ADI is a simpler interface than MPI proper and focuses on moving data between the MPI layer and the network subsystem. Those wishing to port MPI to a particular platform need

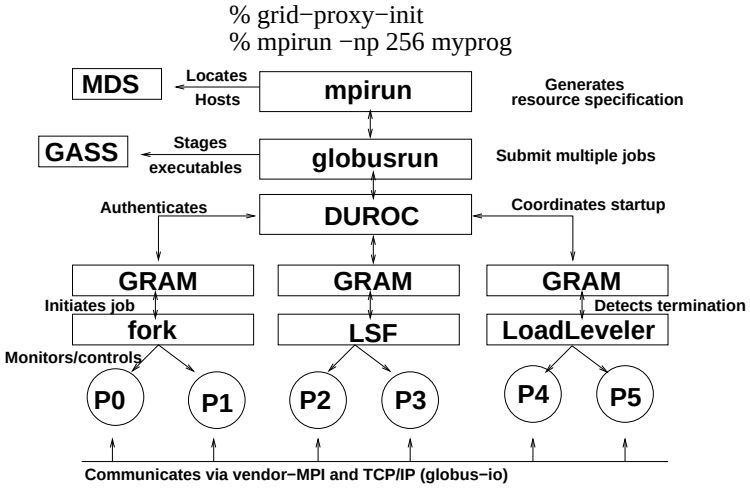


FIG. 3.1. *Overview of the startup of MPICH-G2 and the use of various Globus 2.4 Toolkit components to hide and manage heterogeneity. (This diagram was taken from [2])*

only define the routines in the ADI in order to obtain a full implementation. This is how MPICH-G2 works, by creating a special ADI device that uses the Globus Toolkit.

4. WebCom-G. WebCom-G [7, 8] is a Grid middleware that seeks to act as a "Grid Operating System". Unlike other Grid middlewares, WebCom-G (through it's use of the CG Model [9]) plans to hide the underlying complexity of a Grid infrastructure from application developers and end users. With WebCom-G, applications can be easily built as simple CG workflows and submitted to to a WebCom-G Grid for execution. WebCom-G also provides interoperability with existing middlewares like Globus, DCOM, Corba and EJB. CG workflows can be made to target services created with different middlewares. This section presents an overview of WebCom-G and the advantages it provides to application developers and end users.

The WebCom-G Grid Operating System is based on a pluggable module architecture and constructed around a WebCom kernel [10, 11, 12, 13]. It offers many features suited to Grid Computing. Before the WebCom-G project began, much research had already been carried out in the development of the WebCom meta-computer and the WebCom-G project was able to leverage this work. WebCom-G can be seen as a "Grid-Enabled" implementation of WebCom, where new features have been added to WebCom to allow it operate in Grid Environments.

Architecture of WebCom-G—A Modular Base Design. WebCom-G is an abstract machine architecture [7, 8, 10] consisting of a number of modules and a central component called the Backplane (see Fig. 4.1).

The Backplane component forms the basis of a WebCom-G machine and is used to load required modules. A basic WebCom-G system consists of the backplane and a set of five core modules:

1. Processing Module
2. Communications Module
3. Fault Tolerance Module
4. Load Balancing Module
5. Security Manager Module

Though users are free to implement there own Processing Modules, in this paper, the Processing Module refers to a Condensed Graph Engine [9]. A detailed description of all these modules is beyond the scope of this paper, a more thorough discussion of them found in the referenced papers.

To allow easy adaption to specific applications, the Backplane component allows the dynamic loading of different modules. The pluggable nature of each WebCom-G module provides great flexibility in the development of new modules and adapting WebCom-G to new application areas. The Backplane acts as a boot-strapper that co-ordinates the activities and communication between modules via a well defined interface. Inter-module communication is carried out by a WebCom-G messaging system through the Backplane. The Backplane inspects messages and determines whether they should be routed to local modules or to other WebCom-G machines, via the current communication manager. This mechanism provides great flexibility especially as "the

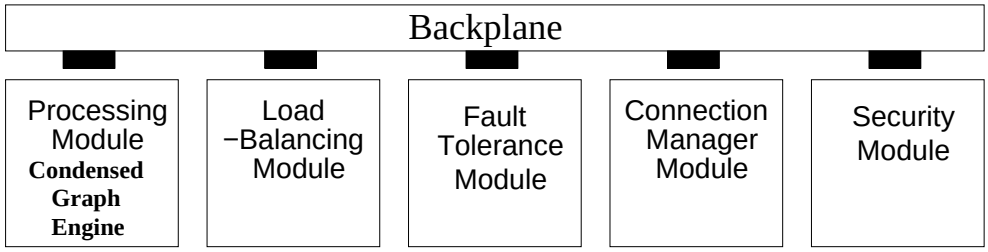


FIG. 4.1. *A minimal WebCom-G installation consists of a Backplane component, a Communication Manager Module and a number of module stubs for Processing, Fault Tolerance, Load Balancing and Security*

mechanism can be applied transparently across the network: transforming the metacomputer into a collection of networked modules". This allows an arbitrary module to request information from local modules or modules installed on different WebCom-G machines. For example, when making load balancing decisions, a server's Load Balancing Module can request information from the Load Balancing Module of each of its clients.

To facilitate the development of new applications and the integration of existing applications into WebCom-G, a number of programming and development tools were developed to allow the easy creation of Condensed Graphs. These tools include

1. APIs for languages like Java and C++.
2. An XML Schema for expressing Condensed Graphs.
3. Visual Tools that allow users to create Condensed Graph applications graphically e.g., WebCom-G IDE
4. High Level Language compilers that compile existing applications (written in languages like Java) directly into Condensed Graphs. These compilers can extract parallelism from sequentially written applications and have the advantage of not requiring application developers to know anything about the Condensed Graph Model of Computing.

4.1. WebCom-G IDE. The WebCom-G Integrated Development Environment provides a graphical method for creating Condensed Graphs. From within the IDE a user can create, load, save and execute Condensed Graph applications. A palette of nodes is supplied; these can be dragged onto the canvas, and linked together to form the graph. Fig. 4.2 shows an example graph to calculate the factorial of a number, which was developed with the WebCom-G IDE.

The IDE has the advantage of allowing the rapid development of Condensed Graphs compared to using the CG construction APIs or writing a graph manually in XML. When a Condensed Graph is created in the IDE and is being output, it is converted to XML which can be stored on disk (& subsequently reloaded by the IDE) or submitted directly to a running WebCom-G machine for execution from the IDE.

The palette on the IDE by default contains a set of core nodes for the construction of Condensed Graphs, but users can also create their own nodes which can be stored in a *Node Database* and loaded by the IDE and displayed on the palette. Also another very important feature of the IDE is that it can load services from the Interrogator Database and display these services on the palette as well. The *Interrogator Database* will be discussed in Section 4.2; it's basically a database of all the available services running on all the machines making up the current WebCom-G Grid. At specified intervals, WebCom-G machines can run their Interrogators, which register all available services on these machines with the Interrogator Database.

With all these services displayed on the palette, users can easily create distributed applications from the IDE that incorporate many different technologies, which then can be submitted to a WebCom-G Grid for execution, where the WebCom-G properties like load-balancing, fault tolerance and security can be taken advantage of.

4.2. Default WebCom-G LB Module & Interrogators. The Load Balancing module of a WebCom-G machine decides where instructions (tasks) are to be executed. (In WebCom-G, tasks are referred to as instructions—but note the term instructions can mean instructions of various grain sizes. So in WebCom-G, a large sequential program is referred to as an instruction.) Different Load Balancing modules use different strategies when scheduling instructions for execution. They can take into account issues like security restrictions (by consulting with a Security Manager module), specialized resource locations and access reliability. In addition to these type of considerations, different Load Balancing modules can be employed to implement specific load balancing algorithms.

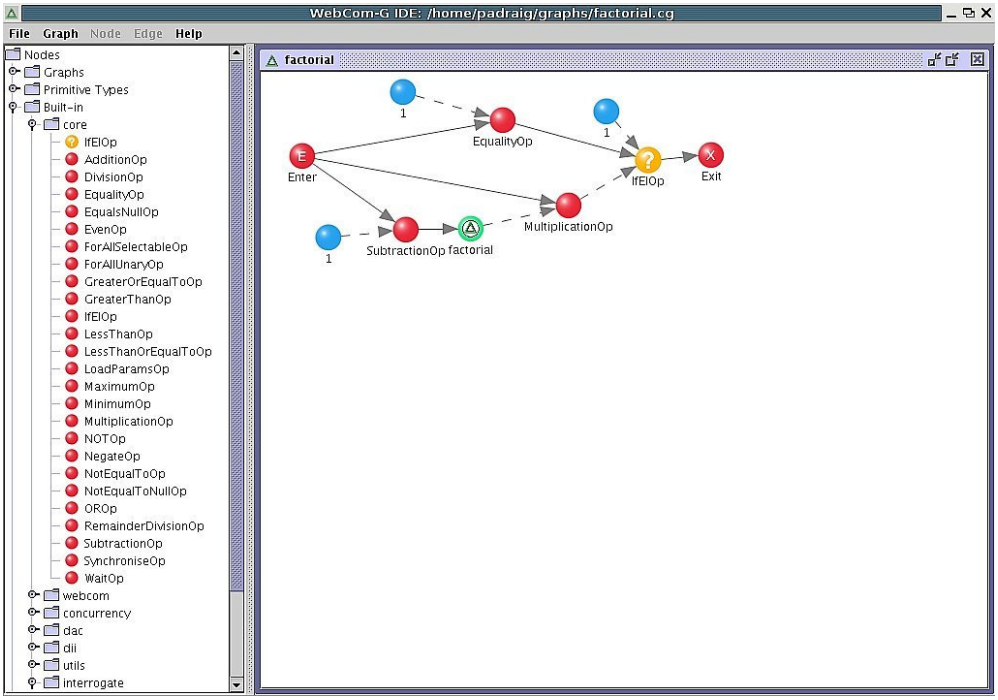


FIG. 4.2. *An example graph to calculate the factorial of a number, which was constructed in the WebCom-G IDE*

Though WebCom-G Load Balancers can take on many forms, and users are free to implement their own load balancing strategies, one very important Load Balancer module is the Default WebCom-G LB module. The default WebCom-G Load Balancer (which employs its own targeting mechanism), allows service invocations to be targeted at specific machines that provide those particular services. Some of the main components of this Load balancer are the Interrogators.

1. **Services:** When the term *"service"* is used in this paper it can cover a broad range of software types, some of these are listed below:
   (a) Machine dependent users applications and licensed software i. e., Matlab or just standard user applications installed on a machine.
   (b) CORBA, EJB and DCOM services.
   (c) Web Services.
   (d) Globus Web Services and other Grid services.
   The above list only mentions some of the possibilities and users can easily create their own execution plug-ins, that allow new service types to be handled.
2. **Interrogators:** An *Interrogator* is a component that can interrogate a WebCom-G machine to see if and what services the machine is running. Different Interrogators have to be created for different service types .i.e an CORBA Interrogator for CORBA, a EJB Interrogator for EJB etc. Once an Interrogator gathers a list of services present on a particular machine, it can register these services with an Interrogator Database. This Interrogator Database will contain a list of all the services running on all the machines making up the current WebCom-G Grid. This Interrogator Database can then be consulted by WebCom-G Load Balancer modules to see where certain services can be executed. Also the Interrogator Database can be used by the WebCom-G IDE, see Section 4.1.

When WebCom-G is running on a machine, it can run it's own interrogators to see what services the machine is running and then register these services with the Interrogator Database. Any WebCom-G machine in general can use this Interrogator Database for scheduling issues e.g., to see what machines are running a certain service. A top-level WebCom-G machine can request all it's client WebCom-G machines (and their clients, if they have any) to run their interrogators by simply executing a particular Condensed Graph application. This Condensed Graph is recursive, in that when it executes on a machine it invokes the interrogators on that machine and passes an instance of its self onto all the clients of the machine for execution. So every

client WebCom-G machine connected to the tree, registers all they're available services with the Interrogator Database.

When the default WebCom-G Load Balancer is asked to schedule a service (or standard instruction) that is not in the Interrogator Database, it can do two possible things depending on the rules that were set by the user—it can either schedule the service/instruction for execution on a remote client (using a round-robin scheduling policy) or execute the instruction locally.

4.3. WebCom-G Portal Server. As well as using the IDE and web services to submit jobs, a WebCom-G (Web) Portal Server can also be used to access a WebCom-G Grid (e.g., `http://portal.webcom-g.org`). In order to avoid a single point of failure, a number of Portals per Grid can be maintained. The WebCom-G Portal Server allows users to access a WebCom-G Grid, once a Web Browser is installed on their machine. When a user is granted access to a WebCom-G Portal, they can perform the following activates:

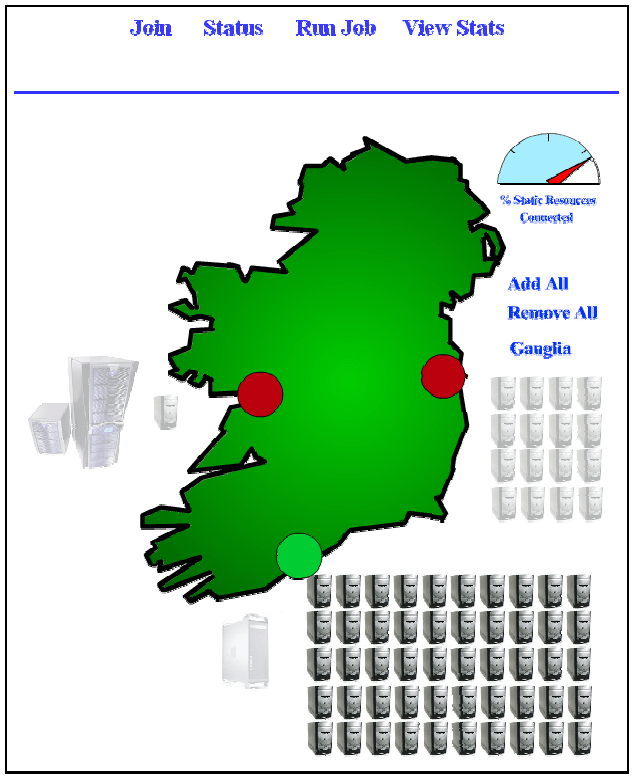


FIG. 4.3. *Graphical view of the resources connected to the WebCom-G Portal*

1. Donate the idle compute cycles of their PC to the WebCom-G Grid
2. Submit jobs to the WebCom-G Grid for execution—jobs can range from simple sequential programs to complicated parallel/distributed Condensed Graph applications. Results from executed jobs can be stored on the server for collection by a user at a later time.
3. View the statistics of the resources in the current WebCom-G Grid (If security access allows). Statistics that can be viewed range from CPU load, memory usage, software packages installed on machines, current status of executing jobs etc. Various external Information Gathering Systems can be used by WebCom-G to make scheduling decisions (e.g., the use of Ganglia [14]) and a user can view this information on the Portal Server—see Fig. 4.4.

Adding proper security mechanisms to WebCom-G (& WebCom-G Grids in general) to guarantee the privacy of users' data etc, is still the focus of on going work and despite a lot done, a lot of work remains. Normally the WebCom-G daemon running on the WebCom-G Portal forms the head of the Grid (the root of the tree), though this isn't necessary always the case, parts of a WebCom-G Grid can form a peer-to-peer topology e.g., its possible to have a number of WebCom-G Portals connected to each other as peers and some client resources only visible to certain Portals, but if the need arises portals can donate some of their resources

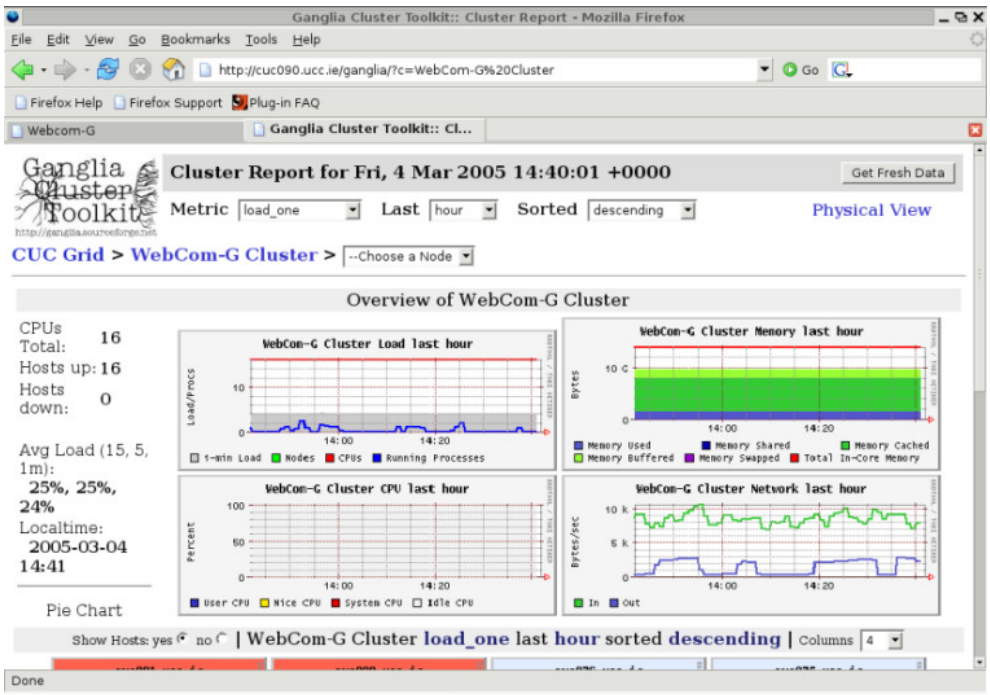


FIG. 4.4. *Information gathered by Ganglia on the resources attached to a WebCom-G Grid*

to other portals (if security permissions allow).

Fig. 4.4 shows the information gathered by Ganglia [14] from a number of resources connected to a WebCom-G Grid. This information can be used by WebCom-G to make scheduling decisions and can also be displayed to a user, who wants to see the current state of Grid resources.

5. Automating the Deployment, Execution & Fault Survival of MPICH-G2 jobs with WebCom-G. Authorised users can visit a WebCom-G Portal (see Section 4.3) and submit their applications, which are written in the form of Condensed Graphs. WebCom-G then schedules the execution of these Condensed Graphs across the available resources. As well as the end results, various statistics about each job and the current state of the underlying resources (see Fig. 5.1) are also maintained by the Portal.

The above notion was extended for allowing portal users to execute MPI (MPICH-G2) jobs on the WebCom-G Portal (and the WebCom-G Grid). Users can visit the WebCom-G Portal, upload their MPI code and specify the number of machines they require. WebCom-G firsts decides the machines on which the MPI application will execute. The MPI code is then compiled and executed on these machines. If any errors occur during the compilation, they are returned to the user and the process aborted. Finally the application is scheduled for execution on the underlying resources and the following tasks are performed:

- A Resource Specification Language (RSL) script is built on the fly, depending on machine availability, path & package availability.
- The application is run using the RSL file.
- In the case of failure, the application is rescheduled with a new RSL file, dynamically created by WebCom-G from a new interrogation of the underlying resources. WebCom-G then re-launches the job with the new RSL file. No user intervention is required at this point.
- Finally results are sent to the Portal, for collection by the user.

5.1. Overview of the MPICH-G2 CG Application. This section presents an overview of the MPICH-G2 Condensed Graph application that was developed, in order to use WebCom-G to manage MPICH-G2 jobs (i.e. handling issues like fault survival etc).

Fig. 5.2 shows the MPICH-G2 Condensed Graph application that was created with the IDE. The following presumptions were made about the graph based on the Globus test-bed that was used in its development. All machines taking part used a shared file-system, and before the graph is executed the source will have been

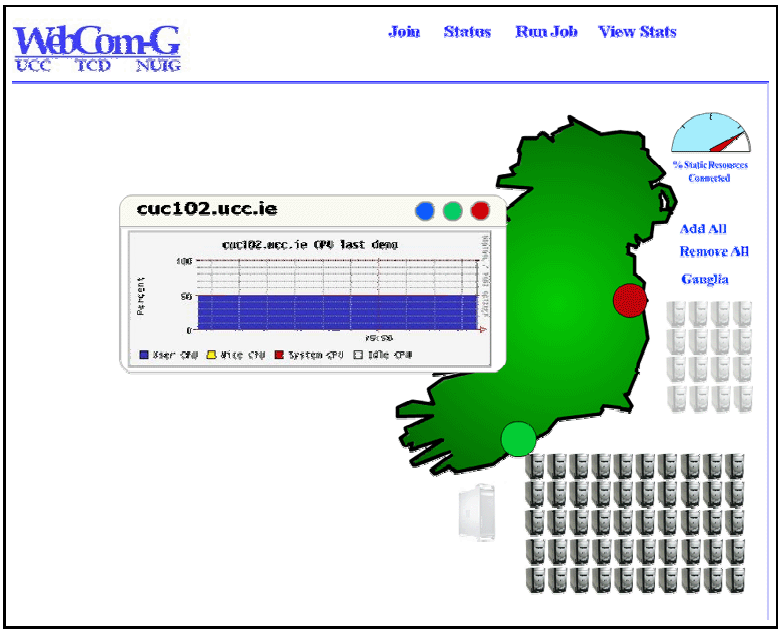


FIG. 5.1. *Zooming in on the current state of a particular machine/resource in a WebCom-G Grid*

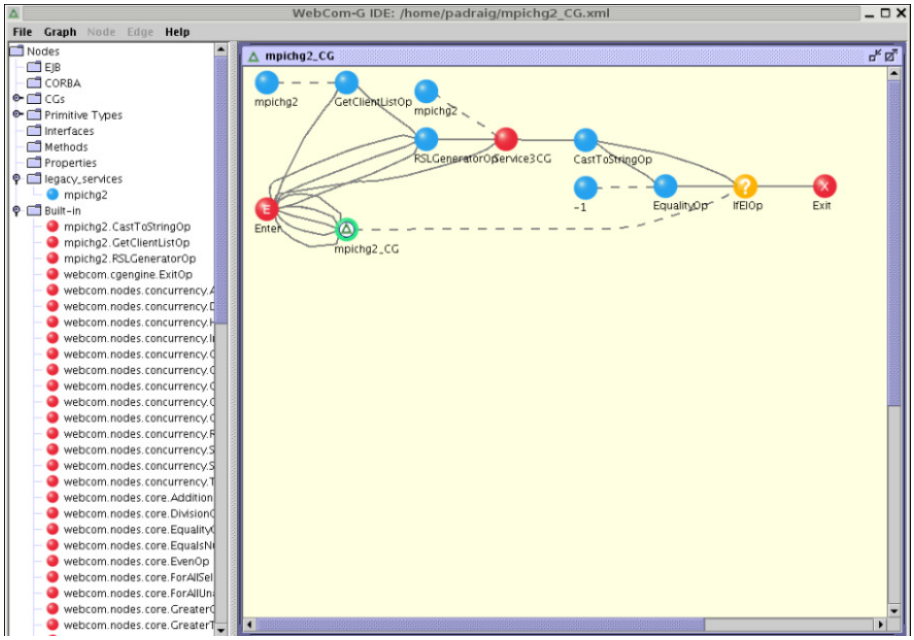


FIG. 5.2. *The MPICH-G2 Condensed Graph Application, developed in the WebCom-G IDE*

compiled (and will have to have compiled without errors, otherwise the process is aborted and the user is notified.) Though the graph can easily be changed to account for machines that don't use a shared file system. (Whether there is a shared file-system present or not, this has no affect on testing whether the fault survival mechanism of WebCom-G works.)

The graph takes five inputs:

- Number of machines required.
- MPI source code.
- Directory to run the MPICH-G2 job from.
- Arguments to pass to the MPICH-G2 job.

- Timeout: Users (if they want) are allowed to specify a timeout for their MPICH-G2 job, if the execution time of a job exceeds the timeout, the job is thought to have failed and is rescheduled.

Overview of the main nodes of the graph (see Fig. 5.2) and how it executes:

- *GetClientListOp*—This node takes two parameters, the name of a service, which in the case of this graph is mpichg2 and the number of machines required. The node when executed, outputs a list of machine names (no greater then the number of required machines). This node uses the current list of available client machines connected to the (WebCom-G) Portal and the Interrogator Database to see which of these clients can execute MPICH-G2 jobs and then creates a list of machines that can be used in the execution of the MPICH-G2 job.
- *RSLGeneratorOp*—This node generates an RSL file which can be used to execute the MPICH-G2 job. It takes the list of machine names generated by the GetClientListOp node, the name of the executable. the directory to run the MPICH-G2 job from and the arguments to pass to the job. From these, this node generates the RSL Script.
- *Service3CG*—This node executes the MPICH-G2 job with the generated RSL Script and the timeout if specified by the user. This node then monitors the execution of the job and in the event of a job failure/crashing, it detects the job failure and reports it. This node can detect failures of MPICH-G2 jobs in the following ways:
  - *mpirun returns and data has been written to the "stderr"*: There are a few bugs in the current release of MPICH-G2 (for details, readers are referred to the MPICH-G2 web site at `www.hpclab.niu.edu/mpi/`) that affect the detection of failures from the mpirun command. One of these bugs is that "the exit code passed to MPI_Abort does not get propagated back to mpirun". Another is that sometimes "when calling MPI_Abort, "stdout/stderr" are not always flushed unless the user explicitly flushes (fflush) both prior to calling MPI_Abort, and even then, the data is sent to stdout/stderr of the other processes". These bugs are due to be fixed in future releases of MPICH-G2. So as the Service3CG can not get the exit code passed back by MPI_Abort to mpirun, it just monitors the "stderr", if any data is written to "stderr" the node considers the job to have failed. When users are submitting MPI jobs to the WebCom-G Portal, they are expected to be aware of these issues. When future releases of MPICH-G2 are released, these problems can be removed.
  - *A WebCom-G client (that is in the RSL script) fails*: Users (if they want) can specify that a job is considered to have failed if a WebCom-G machine which was included in the RSL script fails during the execution of the MPICH-G2 job. So when a WebCom-G client fails, the Service3CG forces the job to terminate and reports the job as failed. One potential problem with this type of fault detection is that, it could be possible that the WebCom-G client that failed might have finished actively taking part in the job and so even despite its failure, there is no reason to reschedule the job.
  - *Timeout*: Users (if they want) are allowed to specify a timeout for their MPICH-G2 job, if the execution time of a job exceeds the timeout, the job is thought to have failed and is rescheduled. This timeout is only supplied as an option and is not ideally suited to a Grid environment, as resources in the grid are heterogeneous in nature. So executing an MPICH-G2 job on different machines (of different performance levels), obviously can cause the execution time of the job to vary greatly between runs. It's presumed, if a user specifies a timeout that they are thinking of a worst case scenario.
- *IfElOp*—When this node fires, if the MPICH-G2 job failed it will evaluate to true, in which case the *MPICHG2_CG* node fires, this node is really just a recursive call to the graph itself and causes it to execute again. (When the graph is re-executed any machines that have failed with not be in the list generated by the *GetClientListOp* node. Users can set in a properties files to have a re-integration of the underlying resources again in case any new resources have been added since the previous execution, though any new resources that join the Portal, usually perform interrogation when they join.) At the moment, the job will only be re-executed five times, after that a message is returned to the user informing them that the job failed five times in a row. It is then up to the user, whether they want to re-submit the job straight away or wait a certain amount of time. Otherwise the job will have executed correctly and the graph exits and returns the results to the Portal for collection by the user.

6. Sample Executions and Results. A simple application was developed to demonstrate the system in action (using WebCom-G to manage MPICH-G2 jobs)—the application simply counts all the prime numbers up to a certain number (1 million) and returns the result. The number partition (1 to 1 million) is divided

evenly among the available machines—obviously with this type of application, its execution time decreases as more machines are added. No complex algorithms were used just a simple brute force check, as the goal was to highlight WebCom-G's management of MPICH-G2 jobs not the efficiency of the MPI program.

Our test-bed consisted of six Debian machines that had Globus, MPICH-G2 and WebCom-G installed. So when the Portal was scheduling MPICH-G2 jobs it could use these six machines.

The following executions were performed and their results recorded:

- Simulation 1: Execute the application on five machines (no failures).
- Simulation 2: Execute the application on four machines (no failures).
- Simulation 3: Execute the application on five machines & during it's execution one controlled machine failure occurred, causing WebCom-G to reschedule the job
- Simulation 4: Execute the application on five machines & during it's execution two controlled machines failures occurred, causing WebCom-G to reschedule the job

Note: As there are only six machines in the test-bed, if a user wants five machines but two fail then obviously the application can only be re-run with four. This is because our test-bed had just six machines, it's presumed that in a true Grid environment, more machines would be available then required by the user and if one of their allocated machines fails, a new one can be obtained in its place.

The *'Total Execution Time'* of the application includes the time from when 'mpirun' was called to when it returns. The *'Individual Execution Time'* includes the time each machine spent executing its own prime count on the number partition that was assigned to it. In the case of a controlled machine failure this time indicates the time from when it started until it failed. (Time is shown in seconds.) As the number space (1 to 1 million) is divided into number segments, the segments with smaller numbers with execute much faster then the segments with larger numbers. For example in the following experiments Machine 1 gets smaller numbers then Machine 5, so Machine 1 checks its assigned number space alot faster then Machine 5. Also its worth noting that in Simulation 3 and 4, Machine 1 finished executing before the failures, but when the application was re-launched Machine 1 re-did the work it had already completed - this is because the current system guarantees only fault survival. If the job fails before fully finishing, it is completely re-launched.

Simulation 1 : Execute the application on five machines (no failures).

- Total execution time: 232
- Individual execution time for each machine:
  - Machine 1: 28.3
  - Machine 2: 78.2
  - Machine 3: 123.5
  - Machine 4: 171.2
  - Machine 5: 207.9

Simulation 2 : Execute the application on four machines (no failures).

- Total execution time: 281
- Individual execution time for each machine:
  - Machine 1: 43.4
  - Machine 2: 120.5
  - Machine 3: 188.6
  - Machine 4: 262.5

Simulation 3 : Execute the application on five machines & during it's execution one controlled machine failure occurred, causing WebCom-G to reschedule the job.

- Total execution time: 355
- Individual execution time for each machine -run 1:
  - Machine 1: 28.4
  - Machine 2: 78.6
  - Machine 3: 100
  - Machine 4: 100
  - Machine 5: (failed at) 100
- Individual execution time for each machine -run 2:
  - Machine 1: 28.2
  - Machine 2: 78.5

  - Machine 3: 123.3
  - Machine 4: 171.2
  - Machine 5: (new machine) 207.3

Simulation 4 : Execute the application on five machines & during it's execution two controlled machines failures occurred, causing WebCom-G to reschedule the job.

- Total execution time: 411
- Individual execution time for each machine -run 1:
  - Machine 1: 28.3
  - Machine 2: 78.1
  - Machine 3: 100
  - Machine 4: (failed at) 100
  - Machine 5: (failed at) 100
- Individual execution time for each machine -run 2:
  - Machine 1: 43.4
  - Machine 2: 118.3
  - Machine 3: 188.3
  - Machine 4: (new machine) 256.4

(With the two machines failures, only four machines left)

Summary. Fig. 6.1 shows the total execution times of the different simulations. Due to the simplicity of the application these results are, as would be expected:

- Simulation 1 takes 232 seconds to execute with five machines.
- Simulation 2 takes 281 seconds to execute with four machines, obviously taking longer to execute then Simulation 1.

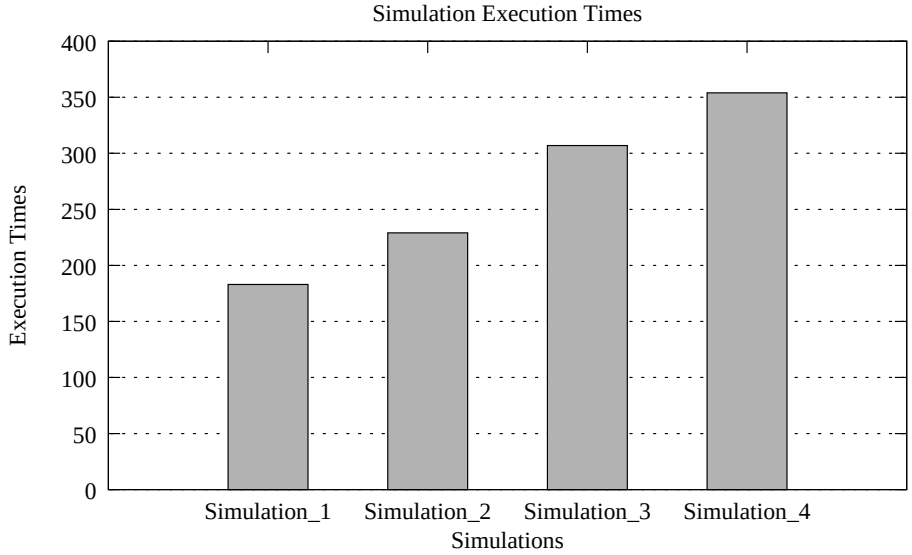


FIG. 6.1. *Total execution times of the different simulations*

- Simulation 3 takes 355 seconds to execute, because during its first execution at time 100, machine five fails, causing the whole job to fail and be rescheduled. During its second execution the application completes fully. So the total execution time for this simulation is roughly around: 100 (time in the first execution when the job failed) + 232 (the total execution time from Simulation 1) + 23 (the overhead of running the graph in WebCom-G and the start-up of an MPICH-G2 job—the time 100 refers only to the time the program spent executing the prime count code, it does not include the startup time for the MPICH-G2 job).
- Simulation 4 takes 411 seconds to execute, because during its first execution at time 100, two machines (four and five) fail, causing the whole job to fail and be rescheduled. During its second execution

the application completes fully (but with only four machines, as there were only six machines in our test-bed and two had failed). So the total execution time for this simulation is roughly around: 100 (time in the first execution will the job failed) + 281 (the total execution time from Simulation 2) + 25 (the overhead of running the graph in WebCom-G and the start-up of an MPICH-G2 job—the time 100 refers only to the time the program spent executing the prime count code, it does not include the startup time for the MPICH-G2 job).

7. Conclusions. In this paper, it was discussed how WebCom-G (& the CG Model) can be used to automate the deployment, execution & fault survival of MPICH-G2 jobs. The main purpose of this research has been to focus on fault survival issues related to MPICH-G2 jobs and allow users to execute their MPI jobs in grid environments, through the use of simple easy to use web interfaces. If a job submitted to the grid fails, the whole job will have to be re-submitted. However, the benefits of using WebCom-G for scheduling these jobs are immediately perceptible. If a job fails WebCom-G's fault survival will ensure that the job is automatically rescheduled. This may be to the same grid, or a different grid with the required resources. This dynamic scheduling of grid operations is possible by delaying the creation of the RSL script until just before it is required. Once the RSL script has been created, the physical configuration is determined.

REFERENCES

[1] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[2] Nicholas T. Karohis. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. citeseer.ist.psu.edu/554400.html.

[3] Padraig J. O'Dowd, Adarsh Patil, and John P. Morrison. Managing MPICH-G2 Jobs with WebCom-G. In *Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC)*, Lille, France, July 2005.

[4] Adarsh Patil, Padraig J. O'Dowd, and John P. Morrison. Automating the Deployment, Execution & Fault Survival of MPICH-G2 jobs with WebCom-G. In *Proceedings of the 2005 Cluster Computing and Grid (CCGrid) Symposium*, Cardiff, UK, May 2005.

[5] Brian Clayton, Therese Enright, and John P. Morrison. Running MPI Jobs with WebCom. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, June 2005.

[6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[7] John P. Morrison, Brian Clayton, David A. Power, and Adarsh Patil. WebCom-G: Grid Enabled Metacomputing. *The Journal of Neural, Parallel and Scientific Computation. Special Issue on Grid Computing.*, 2004(12)(2):419–438, April 2004.

[8] David A. Power, Adarsh Patil, Sunil John, and John P. Morrison. WebCom-G. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2003)*, Las Vegas, Nevada, June 2003.

[9] John P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[10] David A. Power. *A Framework for: Heterogeneous Metacomputing, Load Balancing and Programming in WebCom*. PhD thesis, University College Cork, 2004.

[11] James J. Kennedy. *Design and Implementation of an N-Tier Metacomputer with Decentralised Fault Tolerance*. PhD thesis, University College Cork, 2004.

[12] David A. Power. WebCom: A Metacomputer Incorporating Dynamic Client Promotion and Redirection. Master Thesis, University College Cork, 2000.

[13] John P. Morrison, James J. Kennedy, and David A. Power. WebCom: A Volunteer-Based Metacomputer. In *The Journal of Supercomputing, Vol. 18(1)*, pages 47–61, 2001.

[14] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. In *Parallel Computing, Vol. 30, Issue 7*, July 2004.

Edited by: M. Tudruj, R. Olejnik.
Received: February 24, 2006.
Accepted: July 30, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 87–94. http://www.scpe.org
ISSN 1895-1767

PION: A PROBLEM SOLVING ENVIRONMENT FOR PARALLEL MULTIVARIATE INTEGRATION

SHUJUN LI* , ELISE DE DONCKER* , AND KARLIS KAUGARS*

Abstract. ParInt is a package for parallel multivariate numerical integration. This paper describes the design and implementation of a problem solving environment based on ParInt and Web technology. We call it ParInt ONline (Pion). It facilitates both common end-users and experts to solve computationally intensive numerical integration in parallel. No parallel programming experience or any knowledge of Unix/Linux operating systems is needed for the users. When the user submits an integration problem to Pion via a Web browser, the problem solving environment will compile the integrand function and link it dynamically with the ParInt package so that the execution can be done in parallel on high performance computing servers. The system was designed to be a globally accessible integration platform that operates as a black box, taking user data and producing the results.

Key words. problem solving environment (PSE), parallel integration, scientific visualization

1. Introduction. A high performance computing system consists of the hardware, the runtime systems, the programming languages, the problems, and the users. The objective is to solve the problems. Too much is involved for most users to buy the hardware, set up the systems, learn a special programming language, and write programs to solve their problems. Usually, it is unnecessary for them to even know the details of the algorithms. The solution to bridge the gap between the user and the high performance computing system is the so-called Problem Solving Environment (PSE) [19, 14, 12, 4]. A PSE provides a complete environment for solving large computationally intensive problems. Attention is paid to the needs of both end-users and experts, who are in a position to steer the solution of the application, and require suitable problem solving techniques based on information gathered during the problem solving process.

The characteristics of a PSE are:

Problem domain. PSEs offer an integrated environment for solving specialized problems, including means for composing, compiling, and running applications, while handling security issues.

Parallel resources. PSEs provide easy access to distributed computing resources, and state of the art problem solving power to the end user.

Black box configuration. PSEs attempt to remove hardware and software complexity, in order to eliminate the need for the end user to be an expert in the solution method.

Grid distribution. PSEs can be implemented on top of grid services.

Visualization. PSEs should address visualization that enhances computational steering, analysis and interpretation of the resulting data.

The aim of this work is to incorporate the state of the art integration techniques provided by ParInt into a globally accessible integration service. This effort is best understood in the context of a complete problem solving environment [19, 14]. We will call ours Pion, which stands for Parallel Integration ONline.

Many PSEs are designed to solve problems in specific domains. Ecce [13] is a PSE, consisting of a suite of applications or components for constructing molecules, assigning basis sets, selecting input options, browsing resource availability, launching jobs, visualizing results, and creating and managing projects and calculations.

PDELab [12] is a PSE for modeling physical objects described by partial differential equations. It provides a graphical interface to define a problem and select a solution method, then provides powerful computational resources to solve the problem and visualize the results.

VizCraft [11] is problem-solving environment used by aircraft designers during conceptual design. It integrates simulation code that evaluates a design with visualization for analyzing the design individually or in contrast with other designs.

WBCSim [11] is a prototype PSE that is intended to aid in the research of wood-based composite materials, by allowing access to legacy FORTRAN programs.

The following systems target more general applications. The paper [8] presents a model for a World Wide Web computational server, which uses Maple as its underlying engine, and communicates with its clients using OpenMath, MathML, and VRML. The authors mainly focus on the client/server communications.

*Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008 USA ({sli,elise,kkaugars}@cs.wmich.edu).

Cactus [1] is an open source PSE intended to provide a unified modular and parallel computational framework for engineers and physicists.

NETCARE [5], powered by PUNCH [6, 5] and Condor [16], is a Web-accessible distributed infrastructure of tools and computing resources for the computer architecture community. PUNCH allows users to access and run command-line tools via standard World Wide Web browsers. Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management for batch jobs. With NETCARE, students, educators, and researchers can share, test, evaluate, and use these tools on problems in electronic design automation.

However, it is impractical to use any of the current general purpose Web-enabling tools for our application. The idea of Web-based program execution is simple. Via the graphical user interface on the Web page, input parameters for an application are passed to the middleware which interacts with the Web server. The middleware starts the application and sends the results back to the user via the Web server. However, the PSE of ParInt must be designed and implemented specifically to handle the following situations.

The ParInt end user enters an integrand function which will be compiled into a library. The library is loaded dynamically by ParInt. This is a dedicated procedure that is not handled via general purpose Web-enabling mechanisms.

Secondly, some ParInt parameters are interdependent. For example, an n-dimensional problem needs $2n$ numbers for its hyper-rectangular lower and upper limits. When an argument indicates that the integration region is a simplex, $n(n+1)$ numbers are required to define the simplex region. In our approach, the JavaScript checks parameters and handles inputs accordingly.

Finally, ParInt uses a heap (optionally, a double ended heap or deap) to keep intermediate regions, which may consume a large amount of memory. Therefore, a customized resource management strategy is preferred.

To design a PSE for numerical integration, we have to keep the following factors in mind. The execution time of a job is often unpredictable, which means that both the system administrator and the user should be able to control the execution of the job. The resources may have to be restricted by the system administrator. For example, the system administrator should be able to limit the number of processes running on a node.

Based on current technologies, the performance, user interface, and accessibility lead to trade-offs. It may be easy to meet one or two of these, but building a system that meets all of them requires extra effort. For example, one can write a command-line parallel program for users who have access to a parallel system. One can even provide a decent graphical interface. But if the system is powerful and it needs to be made accessible to remote users, there are still a lot of considerations. On the other hand, if the performance issue can be neglected, so can the term PSE, because many available desktop applications are somewhat user-friendly. Most PSEs address the issue of performance. Some have a Unix/Linux desktop interface. Some are implemented in Java so that users can access the PSE using a Web browser. We will discuss the pros and cons of using Java Applets as the major GUI below.

We aim at building a problem solving environment for numerical integration, so that the users can focus on the problems themselves, instead of focusing on coding, computational techniques or the algorithms used to solve the problems. Unless the integrand functions are predefined in the program, solving an integration problem numerically involves entering the integrand function in some way and compiling it, so that it will work with other parts of the program.

Commercial products such as Maple, Mathematica and MatLab support symbolic and numerical integration. However, the users have to pay for the products and learn to use them. They may not be efficient enough to solve complicated problems. In particular, numerical integration is not well-supported. Non-commercial functions, packages or toolkits are also available. A common problem of these programs is that they are not user-friendly. If we want to provide a user-friendly solution over the Web, there are a number of possible ways to proceed.

Web browsers are becoming popular as interfaces to remote high performance computers. Some PSEs provide this kind of interface. Based on the current technology, there are three major graphical user interface technologies for accessing remote computing resources, namely Java Applets, Java Web Start, and through a Web browser without using Java.

Some PSEs use Java Applets in their graphical user interface to display images or data. We also have an Applet version of the interface to ParInt [9]. In [9], multi-thread Java server communicates with the Java Applet and controls the job execution. We found that it is inconvenient to manage the system and user data with our server. A Java Applet can be a good visualization component of the environment. However, if the Web-based PSE is designed to be globally accessible, it is much easier to use a middleware layer coded in PHP

(a widely-used, Open Source, general-purpose scripting language that is especially suited for Web development) or JSP (JavaServer Pages), with database support to handle most interactions and data.

Java Web Start is an innovative technology for deploying applications based on the Java 2 platform, which enables the user to launch full-featured applications via a browser. It provides the ease of deployment and the use of HTML, as well as the power and flexibility of a full-fledged application. It frees the developer from concerns as to how the client is launched. It may be a good candidate for result visualization. However, if we use it to support the client deployment, we still have to implement our own server that takes care of everything, including connections.

Furthermore, it is increasingly difficult to implement a PSE when the application logic becomes complex. The users need to install Java plug-ins, and it is difficult for some users to install plug-ins successfully on some platforms. With these considerations in mind, we used a simple and extensible method based on JavaScript and DOM (Document Object Model) [18] to build the GUI.

2. ParInt Overview. ParInt is a software package that numerically solves integration problems in parallel. Multivariate functions are integrated over hyper-rectangular or simplex regions, using cubature, Monte-Carlo (MC) or Quasi-Monte Carlo (QMC) rules. General product regions can be handled using transformations [17]. For a function $f(\mathbf{x})$, the objective is to calculate an approximation Q to the integral

$$I = \int_{\mathcal{D}} f(\mathbf{x}) \, \mathrm{d}\mathbf{x},$$

over a given n-dimensional domain $\mathcal{D}$, and an error estimate E_a, subject to

$$|I - Q| \leq E_a \leq \max\{\varepsilon_a, \varepsilon_r |Q|\},$$

where ε_a, ε_r are user-specified absolute and relative error tolerances, respectively. The integrand may be a vector function $\mathbf{f}(\mathbf{x})$, as long as the component functions are sufficiently similar (to allow their integral approximation using the same strategy).

ParInt will evaluate the integrand function a number of times to get the result. The number of function evaluations is used as a measure of the amount of effort spent in the computation. The user can set an upper limit on the number of integrand evaluations to ensure that the computation will stop. It is quite possible that the result of an integration may not be achieved within the given error tolerances, due to the nature of the integrand function, the accuracy requirement, the computation time, the effect of possible round-off error in the computation, or the limits on machine precision. If the function evaluation limit is reached, then the required accuracy has probably not been achieved. Various algorithm parameters can also be specified by the user, for optimization and speed-up of the computation.

3. The Architecture. The ParInt code can be invoked through function calls in an application. It can also be compiled into a ParInt console executable. The user can perform integration via a GUI, which runs locally or remotely. The overall architecture is shown in Figure 3.1.

ParVis [7, 3] is an interactive graphical tool for analyzing region subdivision trees and mesh structures for adaptive partitioning algorithms. It currently provides post-mortem event viewing. In future work, it will be integrated into the Web-based interface environment to enable computational steering.

This paper focuses mainly on the left half of Figure 3.1, with the remote interface to the parallel computing resources. The figure depicts a typical three-tier architecture with database support. However, this environment differs from the common Web applications in that it has to deal with the special issues arising from high performance computing.

The remote interface is a Web browser. JavaScript and DOM [18] are intended for the user's specification of the problem and its parameters, and the display of the results and error estimates. The middle layer provides user management, integrand function management, job management, and result management. In the bottom layer, the ParInt Plug-in Compiler (PPC) [20] (which uses gcc) compiles the code of an integrand function to an object file, which is then loaded dynamically by ParInt. The back end contains the parallel ParInt integration engine and a database for user data.

4. User Management. Many Web applications need to keep user data for various reasons. The simplest way to handle this is by making a directory for each user. Creating individual directories has the advantages of less access time and easy deployment. This method is suited in small applications. MetaΨ [2] uses LDAP

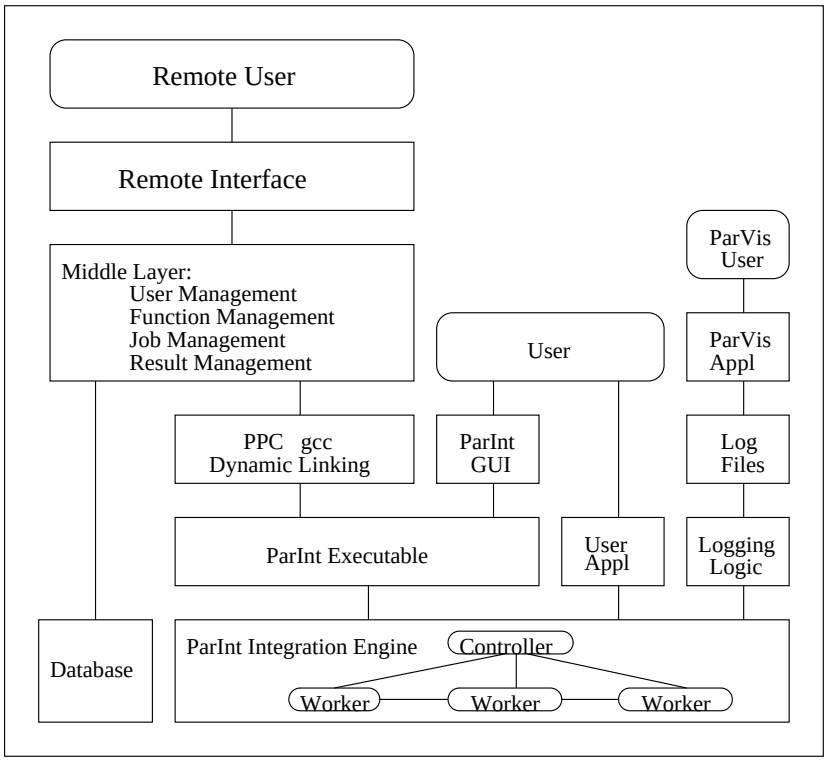


FIG. 3.1. *The Pion architecture.*

(Lightweight Directory Access Protocol) and a Web server as the middleware to locate computing resources and to keep user profiles. For Pion, we found that using a database is a better choice. In addition to the normal fields such as the user name and the email address, the user table currently has two additional fields, *maxRunTime* and *maxJobs*. These two values determine the user's computing resource access levels. Depending on the workload trends of the system, more fields may be added to classify the users. In addition to managing user data, a Web-based computational system must have a way to control the computing resources. This issue will be discussed in the job control section below.

5. Function Management. There are two stages in setting up a numerical integration problem, namely defining the problem and specifying the numerical algorithm's parameters. On the "New Function" page of the Pion interface (cf. Figure 6.1), the user enters the dimension of the integrand function, the absolute and relative error tolerances, the type of integration region, the region itself and the integrand function. The user has the option to upload the function definition from the local disk. When the function is submitted to the server, all the data is saved to a function table in the database. The function is then compiled to generate a .ppl file, which is linked dynamically with the PARINT executable at runtime. If there are compilation errors, another page will show them with highlighted line numbers. The user must then correct the errors and submit the function source code again.

For a multivariate integral of dimension n, $2n$ and $n(n+1)$ text boxes are required for specifying hyper-rectangular and simplex integration regions, respectively. At times, the user may want to change the dimensions or switch to another region type. To solve this problem, we use JavaScript to manipulate the DOM [18] object of the Web page. When the change of dimensions or region type is confirmed by another event, the JavaScript code will compute the new numbers of rows and columns, and refresh the text box table.

All integrals owned by a user are listed in a table. The user can select an entry to edit, clone or delete. The table also indicates which rules/techniques are available to solve the integration problem at hand. For example, Pion provides adaptive, Quasi-Monte Carlo, and Monte Carlo algorithms for hyper-rectangular integration regions. An adaptive cubature is implemented for simplex regions.

For low dimensional problems, the adaptive algorithm is preferred, unless the integrand behavior is erratic.

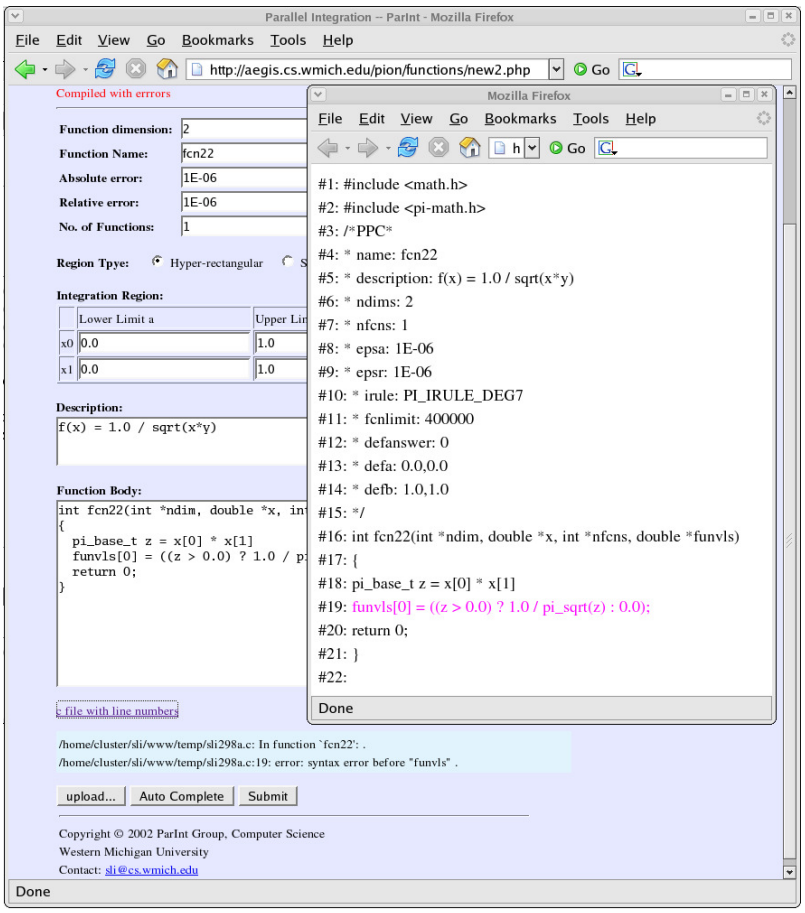


FIG. 6.1. *Integrand function compilation. Error messages are shown when a submitted integrand function has errors.*

For erratic integrands and/or high dimensions, Monte Carlo integration is advised. Quasi-Monte Carlo integration is justified for some high dimensional problems, under certain smoothness conditions on the integrands.

The user also specifies a function evaluation limit, a timeout limit, and a number of processes for the parallel execution. Advanced algorithm parameters are intended for users who have some idea of the algorithm implementation. These parameters can be specified by clicking the "Advanced Options" button.

6. PPC Compiler. Before an integrand function written in C is compiled, it is combined with the other integration parameters to generate an intermediate file. The intermediate file is compiled by a pre-processor called PPC (PARINT Plug-in Compiler) to generate a temporary C source file [20]. This temporary file is then passed to the C compiler to be compiled into a .ppl file. The C compiler reports errors in terms of the intermediate file, not the function entered by the user in the "Function Body" box. The middleware of Pion adjusts the error line numbers to reflect problems with the original file and highlights them for the end user as shown in Figure 6.1. In the event of no errors, the user will be informed with a short message of a successful compilation.

It is clear that allowing the user to write a function and run it on the system is potentially a security hole. Therefore some system calls are not allowed in the .ppl file. They are: accept(), bind(), fopen(), getmsg(), msgget(), open(), pause(), poll(), putmsg(), select(), semop(), wait(), alarm(), brk(), chdir(), dlclose(), dlerror(), dlopen(), dlsym(), exec(), fork(), popen(), pthread_create(), sbrk(), sethostent(), setgid(), setuid(), signal(), system(), thr_create(), and umask(). The restriction with respect to certain function calls is important in an environment where the user's real identity may be unknown.

7. Job Control. Load balancing is always an issue in distributed computing. When a user submits a job to Pion, the middleware will call the MPI [10] mpirun command which typically uses the first p (a user specified

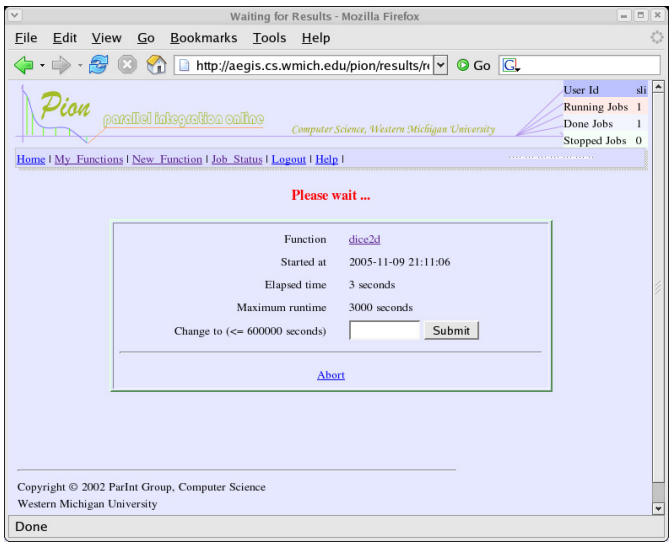


FIG. 7.1. *This page will display the results; it also gives the option to increase the execution time limit, or to abort the execution.*

number) processors defined in a machine file to execute the job in parallel. In a multi-user environment, it likely that the first several processors are overloaded, while the last ones are idle. Pion has a module to check the workloads of all machines and select suitable processors for the mpirun command. The "System Info" pages show the status of the system in detail for both users and administrators.

A globally accessible computing system should have a mechanism to optimize system utilization. Unlike an online shopping application where the transaction time range is well-known, the time for a computational job is in general unpredictable. It can be a millisecond or a month, depending on the size of the problem. A user might submit a job that runs forever and never returns any result. As mentioned before, there is a field in the user table to specify a maximum execution time (*maxRunTime*) for the user's jobs. The value can be increased by the system administrator upon request. Another value is the maximum number of jobs (*maxJobs*) that a user can start concurrently in the system. This value is 1 by default, but can also be increased upon request. Our policy is to allocate just enough resources for a user. The user interface is designed to encourage the users to manage their jobs interactively. The same logic is applied to the input box for the number of processors. Its default value is always one, while most other boxes display the last values entered.

Killing a job is not an easy task in a Web-based computing system. Questions must be answered such as: who can kill the job (the user, the system, or both); how the job is killed; when the job should be killed; how the user is informed if the system has killed a job.

A user can refer to a status table on the Web page, which shows the numbers of running jobs, done jobs, and stopped jobs (killed by the user or the system). Let us first cover how the user executes a job, then discuss the implementation in Pion. After entering the integrand function and its problem parameters, the user: 1) specifies execution parameters including the number of processors and a timeout limit (should be less than *maxRunTime*); 2) gets the result back if the execution time is less than about one second, or 3) waits for the result while the screen is being refreshed; 4) increases the timeout limit (but not exceeding *maxRunTime*) if necessary; 5) aborts the job execution, or 6) closes the browser and comes back later to check the job status; 7) saves or discards the result. Figure 7.1 shows the page for the user to manipulate a job.

The database has a transaction table with fields including: *startTime*, *timeout*, *endTime*, and *isChecked*. When the user submits a job, the timeout value and a timestamp are stored to the timeout field and *startTime* field, respectively. The system checks the transaction table periodically, and kills all jobs that are timed out. After a job has stopped running, a timestamp is written to the *endTime* field. After the user has viewed the result, the *isChecked* field is changed to true, indicating that this transactional record will not be manipulated again unless for statistics purposes. The data in the transaction table can be used to generate a histogram indicating the activities of the system. For example we can show the workload of the day or of the past week. The implementation of job management depends greatly on the scripting language of the middleware.

8. Result Data Management. Result management is simple but critical to problem solving environments on high performance computer systems. Simple Web applications rarely save the results for the user, because it is easy to repeat the computations. Pion, however, keeps the results even though the user closes the browser and ends the session. After the output has been generated, it is saved to a file. Although it is trivial to support e-mail notification, we decided not to implement this feature, because the user will come back and check the run status regularly if the output or the execution time is important.

Unlike other PSEs, Pion does not allow the users to access the file systems of the server directly, nor does it provide a virtual hierarchical file system. From the user's point of view, file manipulation is unnecessary. The results of an integration problem are associated with the problem and can be accessed via the links in a table.

9. Performance Evaluation. Pion was designed and implemented to be a globally accessible Web-based problem solving environment for high performance computation. Much attention has been paid to the security and load balancing of the system. It can be accessed at the following URL, http://aegis.cs.wmich.edu/pion/. Currently our Beowulf Linux cluster consists of 64 nodes, including 32 AMD Thunderbird nodes at 1.2 GHz and 32 AMD Athlon nodes at 800MHz. They are connected by both Fast Ethernet and Myrinet. For users who want to run ParInt on their own hardware resources, the ParInt package is available for download [15]. It can be compiled to parallel executables for a cluster, or sequential executables for a standalone Unix/Linux machine.

The administrative overhead of Pion is insignificant. The compilation time of an integrand function and the functions it calls depends on the size of the source code. The overhead of the Pion middleware for a job execution is about 0.1 second plus the extra waiting time described below. When a job is submitted, the middleware will wait 0.3 seconds for the result. If the result is produced within 0.3 seconds, it is returned to the user. If the execution time is longer, a window as shown in Figure 7.1 confirms that the job is being executed. The page is refreshed to pull the result in an interval of two seconds at first. The interval is incremented by 2 seconds for subsequent refreshes until it reaches 10 seconds.

Myrinet is the default communication device. The system administrator can switch to Ethernet if needed. If there are not enough working Myrinet nodes, Ethernet is selected automatically for a job that requires a larger number of processors.

10. Conclusions and Future Work. Pion provides a state-of-the-art globally accessible problem solving environment, so that users with little computer background can also access high performance computing resources to solve their numerical integration problems.

In PUNCH [5], different applications are supported by using templates to generate HTML code for the user interface. There are actually six different executables on the Pion servers to support different integration algorithms. They handle adaptive, QMC and MC methods using double or long double precision. PHP classes are used to encapsulate the parameters for each method. Similar classes can be implemented to support applications other than ParInt.

The visualization tool ParVis [7] is used on Linux platforms to analyze the ParInt results. Future work includes integrating a visualization tool with the browser so that the end-users can analyze their results online, and to allow for computational steering. It is further necessary to have ParInt available on multiple clusters. XML technology can be used to exchange integration data between clusters. XML format input and output is an extension currently in progress.

Acknowledgments. This work was supported in part by NSF grants CISE ACR-0000442, EIA-0130857, ACI-0203776, and Western Michigan University.

The authors thank the members of the ParInt project, including Laurentiu Cucos and Paul Castone, for helpful discussions.

REFERENCES

[1] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf, *The Cactus Code: A problem solving environment for the grid*, in Proc. High Performance Distributed Computing 2000, 2000, pp. 253–260.

[2] R. Baraglia, R. Ferrini, and D. Laforenza, *MetaΨ: A web-based metacomputing environment to build a computational chemistry problem solving environment*, in 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, 2002.

[3] E. de Doncker, R. Zanny, and K. Karlis, *Integrand and performance analysis with* ParInt *and* ParVis. Unpublished.

[4] E. Gallopoulos, E. Houstis, and J. R. Rice, *Computer as thinker/doer: Problem-solving environments for computational science*, IEEE Computational Science & Engineering, 1 (1994), pp. 11–23.
[5] N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes, *Punch—Web portal for running tools*, IEEE Micro, 20 (2000), pp. 38–47.
[6] N. H. Kapadia and J. A. B. Fortes, *PUNCH: An architecture for Web-enabled wide-area network-computing*, Cluster Computing, 2 (1999), pp. 153–164.
[7] K. Kaugars, E. de Doncker, and R. Zanny, ParVis*: Visualizing distributed dynamic partitioning algorithms*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00), 2000, pp. 1215–1221.
[8] H. Le and C. Howlett, *Client-server communication standards for mathematical computation*, in International Symposium on Symbolic and Algebraic Computation, 1999, pp. 299–306.
[9] J. Li and E. de Doncker, *Dynamic visualization of computations on the internet*, in Lecture Notes in Computer Science, vol. 1593, Springer-Verlag, 1999, pp. 360–369.
[10] Message Passing Interface Forum. `http://www.mpi-forum.org/`
[11] N. Ramakrishnan, L. T. Watson, D. G. Kafura, C. J. Ribbens, and C. A. Shaffer, *Programming environments for multidisciplinary grid communities*, 2001.
[12] J. R. Rice and R. F. Boisvert, *From scientific software libraries to problem-solving environments*, IEEE Computational Science & Engineering, 3 (1996), pp. 44–53.
[13] K. Schuchardt, J. Myers, and E. Stephan, *Open data management solutions for problem solving environments: application of distributed authoring and versioning to the extensible computational chemistry environment*, in 10th IEEE International Symposium on High Performance Distributed Computing, 2001, pp. 228–238.
[14] M. S. Shields, O. Rana, D. W. Walker, M. Li, and D. Golby, *A Java/CORBA-based visual program composition environment for PSEs*, Concurrency—Practice and Experience, 12 (2000), pp. 687–704.
[15] ParInt Group. `http://www.cs.wmich.edu/parint`, ParInt web site.
[16] D. Thain and M. Livny, *Building reliable clients and servers*, in The Grid: Blueprint for a New Computing Infrastructure, I. Foster and C. Kesselman, eds., Morgan Kaufmann, 2003.
[17] J. Van Voorst, A. Raju, E. de Doncker, and K. Kaugars, *Transformation interface - ParInt*, in IASTED 15th International Conference on Parallel and Distributed Computing and Systems, 2003, pp. 702–706.
[18] W3C, *Document Object Model*. `http://www.w3.org/DOM/`
[19] D. W. Walker, M. Li, O. F. Rana, M. S. Shields, and Y. Huang, *The software architecture of a distributed problem-solving environment*, Concurrency: Practice and Experience, 12 (2000), pp. 1455–1480.
[20] R. Zanny, E. de Doncker, K. Kaugars, and L. Cucos, ParInt*1.2 User's Manual*, 2003. `http://www.cs.wmich.edu/parint/`

Edited by: I. Gladewll.
Received: July 11, 2003.
Accepted: May 04, 2005.

Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 95–108. http://www.scpe.org
ISSN 1895-1767

THE GREAT PLAINS NETWORK (GPN) MIDDLEWARE TEST BED

AMY W. APON*, GREGORY E. MONACO†, AND GORDON K. SPRINGER‡

Abstract. GPN (Great Plains Network) is a consortium of public universities in seven mid-western states. GPN goals include regional strategic planning and the development of a collaboration environment, middleware services and a regional grid for sharing computational, storage and data resources. A major challenge is to arrive at a common authentication and authorization service, based on the set of heterogeneous identity providers at each institution.

GPN has built a prototype middleware test bed that includes Shibboleth and other NMI-EDIT middleware components. The test bed includes several prototype end-user applications, and is being used to further our research into fine-grained access control for virtual organizations. The GPN prototype applications and namespace form a basis for the design and deployment of a robust and scalable attribute management architecture.

Key words. Middleware, Shibboleth, NMI-EDIT, Signet, Grouper, Grid computing

1. Introduction and Background. The Great Plains Network (GPN) [1] is a regional consortium of public universities in the states of Arkansas, Kansas, Missouri, Nebraska, Oklahoma, North Dakota, and South Dakota and regional higher education state networks in these states. Member representatives have recognized the strategic importance of sharing resources collaboratively and shared the goals of national efforts, such as the NMI (NSF Middleware Initiative), NMI-EDIT (NSF Middleware Initiative - Enterprise Desktop and Integration Technologies) [2] and Shibboleth [3], devoted to facilitating inter-institutional collaboration. A long-term goal of GPN is to build a regional middleware infrastructure to share existing grid computing resources (computation, storage, data, and applications) across the region and to provide a platform for the development of new tools and technologies.

In June, 2004, GPN was selected to be one of four projects funded by the Extending the Reach (ETR) program. Participating institutions include the University of Arkansas, the University of Missouri, the University of Oklahoma, South Dakota State University, the University of Kansas, the University of Nebraska-Lincoln, the Peter Kiewit Institute, North Dakota State University, the University of South Dakota, and the Great Plains Network Consortium. Community members include volunteers from both academic computing and research groups. The objectives of the ETR project are: 1) strategic planning on a regional level; 2) the development of a knowledge base, including the test bed installation and testing of middleware environments; and 3) educational outreach to participating institutions. One result of this project has been the development and deployment of a unique middleware test bed across GPN institutions that incorporates several NMI-EDIT software components.

1.1. The Challenge. While national networking and networking-related initiatives such as the National Science Foundations Middleware Initiative (NMI) present new opportunities to improve network capacity, security and reliability for research activities, these initiatives also present challenges to the GPN consortium institutions. The GPN consortium includes campuses that are spread across the central United States, there is no hierarchical organization or a single central authority, and there are several heterogeneous approaches to the implementation of core middleware services across campuses. Unlike campus or state organizations in which policy can be established administratively, GPN is best able to influence regional decision making by calling attention to best practices. At the inception of this project, the proposed participants were at varying stages of planning and implementation for core middleware services. Several campuses were implementing either Lightweight Directory Access Protocol (LDAP) or Microsoft Active Directory systems, and one campus had developed its own identification and authorization system. In addition to technical accomplishments, this middleware project has been an example of successful voluntary collaboration among several institutions with a goal of human capacity-building, training and consulting, and shared resources across campuses and state networks [4, 5]

The remainder of this paper describes the GPN core middleware test bed components, primarily including Shibboleth, prototype resources that have been developed to support the GPN federation, the development of federation infrastructure, synergism with regional research groups and the broader middleware and grid computing community, and conclusions.

*University of Arkansas (aapon@uark.edu)
†GPN and Kansas State University (greg@greatplains.net)
‡University of Missouri-Columbia (springer@cecs.missouri.edu)

2. Shibboleth Middleware Component. Encouraged by the funding of the Extending the Reach Proposal in June, 2004, the GPN began developing a middleware test bed that included Shibboleth, integration with campus identity management systems, the development of prototype resources, and a namespace and attribute architecture to support resource sharing among members of the GPN institutions.

NMI-EDIT's Shibboleth is a project and software package from Internet2/MACE (Internet2's Middleware Architecture Committee for Education). It is a protocol and architecture for sharing attributes among trusted institutions, and is designed to authorize a user to a remote web-based resource through the use of the login and attribute information that is maintained at the home institution of the user. Shibboleth is an ongoing project. The current work by Shibboleth developers includes extensions to non-web-based resources and integration with other middleware tools for grid computing and attribute management [6].

Shibboleth protects resources in the same way that a user ID and password can protect resources. However, Shibboleth protection is based on group membership, or attributes, rather than on the identification of a particular user. Access to resources based on group membership is a common access control mechanism for many types of applications, including defined groups for file access in traditional Unix systems and integrated database applications or student information systems. Shibboleth adds to this capability the distributed management of the user attributes at home institutions, rather than the management of these through a central repository.

An example can illustrate the versatility and a usage scenario of Shibboleth. For example, suppose that the University of Missouri-Columbia (MU) would like to share access to its supercomputing cluster and terascale storage facility to authorized users at the University of Arkansas (Uark). Suppose that the potential users at Uark are members of the biomedical informatics research group and the resource is to be made available for a particular research project. With Shibboleth, users at Uark would login using the LDAP directory service at the University of Arkansas using their institutional login ID and password. The user attribute, member of the biomedical informatics research group would be retrieved from the Uark directory service and passed to the resource at MU. Through the Shibboleth infrastructure, the MU computing resource would acquire the attribute (e.g., group membership) that it needs from Uark. It would test the value of the attribute (e.g., biomedical informatics research group) to verify that the user has been allowed to access the resource, and then the user would have access. In this process the users identity information only needs to be maintained at the home institution. A centralized directory server for the whole GPN biomedical informatics virtual organization is not needed. Only the attributes, such as group membership and institution affiliation, that are required to access the resource need to be passed to the MU resource. Additionally, the Uark user has the option not to allow those attributes to be sent to MU. The administrator of the MU resource may configure the system to grant or deny access based on the access control policy that has been established.

Shibboleth allows the privacy of the user to be maintained if the administrator of the resource allows this. The user could desire this, for example, if the researcher is performing access on a large data resource for AIDS and the researcher does not want to reveal his or her personal identity as an AIDS researcher. Only the group membership information is sent, and the user's privacy is protected.

The architecture can allow members of a group to share a single account on a resource. The architecture also allows for a user identity to be passed to a resource that requires the use of a specific account for each user. For example, access to a computational resource may only be allowed to certain users that have already had accounts established with administrator approval. In this case a user's identity would be passed through the Shibboleth protocol and would be matched to the corresponding account on the resource.

Shibboleth consists of two primary software components, an Identity Provider and a Service Provider. The Identity Provider component of Shibboleth is integrated with the identity management system of a user's home institution. It authenticates a user using local authentication mechanisms, and then allows user attribute information that is maintained in an institutional identity directory to be sent to a requesting remote resource. The Shibboleth Service Provider software component protects access to a resource by remote users, and allows access only by users that meet the attribute requirements for using the resource.

The Shibboleth package relies on several underlying software servers and protocols for passing user authentication and attribute information between the Identity Provider and the Service Provider. Figure 2.1 illustrates the flow of the Shibboleth protocol. As shown in Figure 2.1, a user first requests a service, via a web browser, from a Service Provider (SP). The user may want to access a database or to submit a job to a computational resource, for example. The first phase of the Shibboleth protocol is the authentication phase. The SP redirects the request to a Shibboleth "Where Are You From" (WAYF) server, which in turn prompts the user to provide the name of a home institution. This request can be satisfied from a drop-down menu from which the user

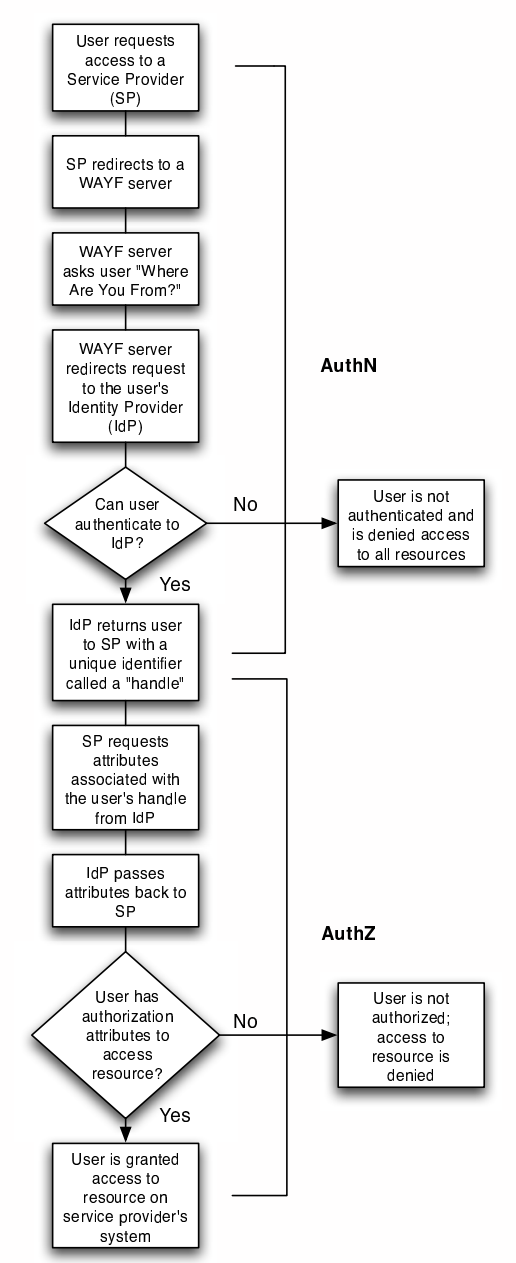


FIG. 2.1. *Shibboleth Protocol*

makes a selection, for example, or software short-cuts can be coded into the system that provide the name of the home institution for certain services. After the home institution of the user has been identified the user's request is redirected to the Identity Provider (IdP) of the home institution. The IdP prompts the user to login using the user's home account and password. After this step, the user is authenticated and the Shibboleth system returns a unique "handle" that is used in the second phase of the protocol.

The second phase of the Shibboleth protocol determines what resources that the user is allowed to access. A distinctive feature of Shibboleth is the separation of the authentication and authorization steps. The Service Provider (SP) requests the necessary attributes from the home institution that are associated with the handle that has been provided in the authentication phase. The user can choose to release or not release attributes to particular requesting services. The IdP passes the attributes back to the SP. As a final step, the SP determines

if the attributes that have been presented are sufficient to allow access to the resource that has been requested. The user is granted or denied access based on the attributes that have released from the home institution.

Shibboleth relies strongly on a trust relationship between the home institution of the user (the Identity Provider) and the institution that provides the resource (the Service Provider). The provider components must both own a public key certificate that is signed by a common Certificate Authority. In addition, the providers must also both be configured to accept requests from the other institution. For resource access to be granted to the user, the Identity Provider must accept requests from the Service Provider to release needed user attribute information. The Service Provider must be configured to allow access by users from the Identity Provider institutions that hold appropriate attributes,

The Shibboleth software package is layered on top of standard software environments. The Identity Provider is a Java web application that runs in the Tomcat servlet container over the Apache web server. The Service Provider is somewhat more complex, but uses standard C/C++ and XML based software components.

The Shibboleth Identity Provider is designed to be integrated with local campus directory and identity management systems. This task has been complicated within GPN by the heterogeneous approach to identity management on the various campuses. In general, campuses that have an LDAP-based identity management system and who were successful at building a preliminary installation of a Shibboleth Identity Provider have been successful during the first year of the project. This success includes integrating the Identity Provider with their local campus system.

3. GPN Prototype Resources. Several prototype resources have been established that allow testing of the middleware test bed. These include a biomedical application, a GPN data repository, and a web-based interface to a computational cluster. Each of these resources has one or more associated attributes (or entitlements) that a user must have to be allowed access. Currently the entitlements are configured using the eduPersonEntitlement field of the EduPerson schema [7], as discussed in Section 4.

3.1. Biomedical Application. In order to demonstrate the use of entitlements within the defined framework, a working research application in animal genomics was converted from its original user interface. The original interface used a Java applet to authenticate and authorize access to the research data by members of the research team. Using the Java applet along with a security database identifying users to be granted secure access to the data, members of the research team could access the data via a standard web browser. No access to the web site or the data is allowed except to the researchers. The user interface was custom designed and implemented for the protection of this data for the research team.

The application was converted to use the Shibboleth protocol for authentication and authorization by replacing the existing Java applet front end. A Shibboleth service web page was created and a link was added to the original application's main web page. This link makes a call to a customized Shiblogin command that internally logs the requesting GPN Shibboleth authenticated user into the protected genomics web site. This is accomplished without the user having to provide a separate login id and password to the application.

Shibboleth login pages for the biomedical application are shown in Figure 3.1. In this figure the user, springer, has selected the home institution, University of Missouri-Columbia, and has entered his password for his home user account. After selecting "OK", the Shibboleth protocol will authenticate user "springer" with the Identity Provider at the University of Missouri-Columbia. The Shibboleth protocol will also obtain this user's attributes (i. e., entitlements) from the Attribute Authority component of the Identity Provider at the University of Missouri-Columbia.

The Shiblogin command receives the Shibboleth generated credentials and entitlements that are verified in the login process. The software ensures that the credentials provided are from an active Shibboleth session and that the required entitlements authorizes the user to access the genomic data. No changes to the original application were required. An additional userid and password was added to the application's security database to grant access to the application by all GPN virtual organization members with the BioSci entitlement.

Within the genomic application, all of the Shibboleth credentials as well as the application's Shiblogin information are available for the application to use as needed. It is possible to provide even more fine-grained control of access. For example, attributes could be defined so that GPN users cannot change any data, but have read-only access to all of the research data.

3.2. GPN Repository. The GPN Repository is a data storage facility that permits members of the collaborating institutions to store and retrieve documents, data and other materials that are to be shared among all of the members. The GPN repository became available to users during the fall of 2004. As with the

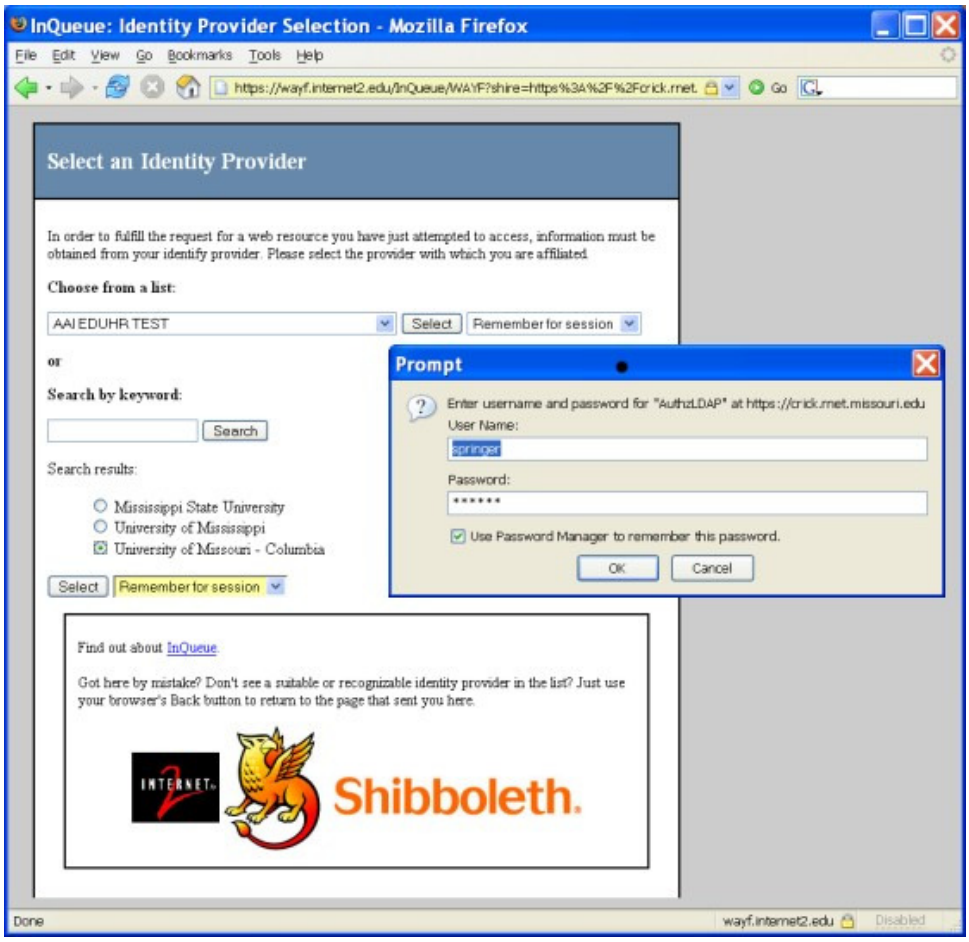


FIG. 3.1. *Shibboleth Login for the User "Springer"*

Biomedical Application, access to the GPN repository is protected by Shibboleth and user authentication to a GPN institution. Currently, access is provided to the GPN repository to any user who is a member of a GPN institution and for whom the Repository entitlement is returned by the campus Identity Provider.

Figure 3.2 shows the front page of the GPN Repository after user "springer" has logged in. It should be noted that the user only needs to log in one time using the Shibboleth protocol, and after that is provided access to several different resources, including the biomedical applications, the GPN repository, and the WebMPI application. Access is granted to the user "springer" for the applications for which his user account has valid attributes. The web page has been configured to allow the user to move easily between the repository and other applications, as shown in the figure. Also note that the network communication is secured via SSL (i. e., the lock on the status line is closed).

The GPN repository is motivated by the need to share large documents within the GPN. Frequently, and many times inappropriately, email (with attachments) is used to share materials among various groups of people. Some of these materials may be quite large and cause many problems such as overflowing mailboxes, causing data to be duplicated in a variety of locations unnecessarily, and can easily become outdated for volatile materials. In general, email is quite inefficient in utilizing resources among collaborators. The GPN repository attempts to overcome some of these inefficiencies by having the materials collected, organized and accessible in one (or more) locations with access restricted to the members of the GPN federation and without a lot of administrative overhead.

One of the uses of the GPN Repository has been to publish copies of the talks that were made at the annual GPN meeting in June, 2005, in Kansas City. The presenters provided copies of their talks to be made available to the GPN members. One presentation was in excess of 58MB and could not be sent to everyone as an email

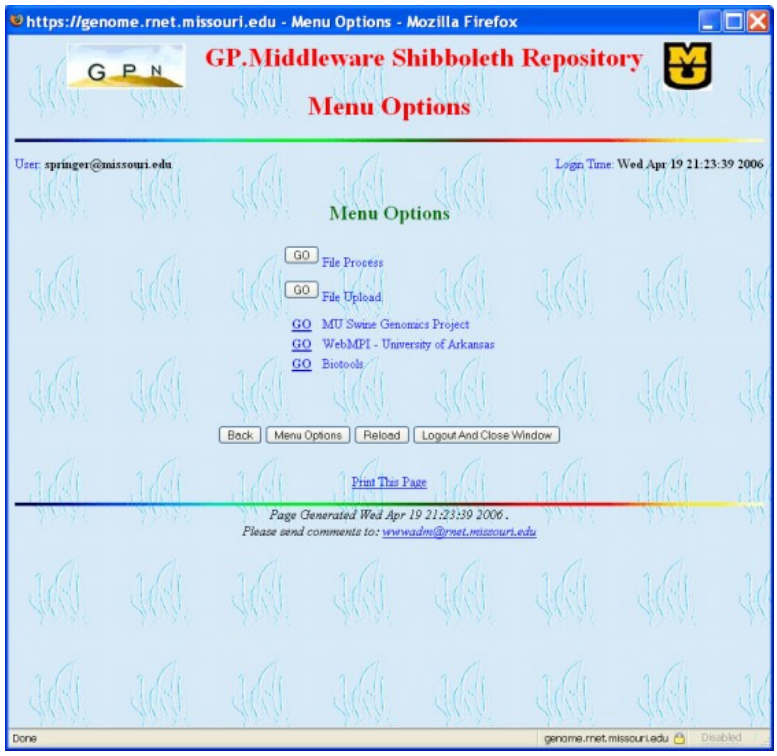


FIG. 3.2. *GPN Repository Page Showing User "springer" Logged In*

attachment. By placing one copy of the presentation in the repository, all GPN members have access to the presentation whenever they want, it is accessible only to the GPN members and no administrative overhead is necessary to provide this protection such as would be required to publish the document on a protected web site.

While the repository presently is used to house documents, presentations and other GPN materials, in the longer term the repository can also be used to provide high performance access to data used by application programs in Grid computing environments among the VO group servers at several institutions.

An ongoing and complementary project is the development of a web-based Subversion [8] document repository system that is also protected by Shibboleth. Subversion allows document check out and check in, and will allow the protection of document subdirectories based on user attributes.

3.3. WebMPI. WebMPI is a Shibboleth application that enables remote users to access a Linux cluster using a web browser for the purpose of running parallel applications. A typical user may be a student in a university course who is studying parallel programming using the MPI (Message Passing Interface) API. As with other Shibboleth applications, users contact the WebMPI interface, and then are redirected through the Shibboleth protocol to authenticate through the authentication mechanism of their home institutions. User attributes are passed to the WebMPI interface, which then determines if the user is authorized to use the cluster or not.

The current implementation of WebMPI is a prototype. The interface consists of a collection of HTML pages and CGI scripts that perform the user's commands on the underlying cluster resource. The interface consists of five main pages. The first page is an introduction page that provides some brief help information on MPI programming. The second page is a file upload page that allows users to browse to a directory on a local computer, select an MPI source file, and upload the file to the WebMPI cluster. The file upload page is shown in Figure 3.3. The third page is a compile page that allows the user to select a file and compile it using either the C or Fortran compiler with MPI libraries. The fourth page is an execute page that allows the user to select a compiled MPI program and to execute it with additional user parameters. The final page is a results that page allows the user to view the output of the MPI application. The current system is convenient for student programs and other types of small demonstration programs. However, it is limited for large production scientific applications. For example, it does not currently have the capability to manage multiple large input or output files.

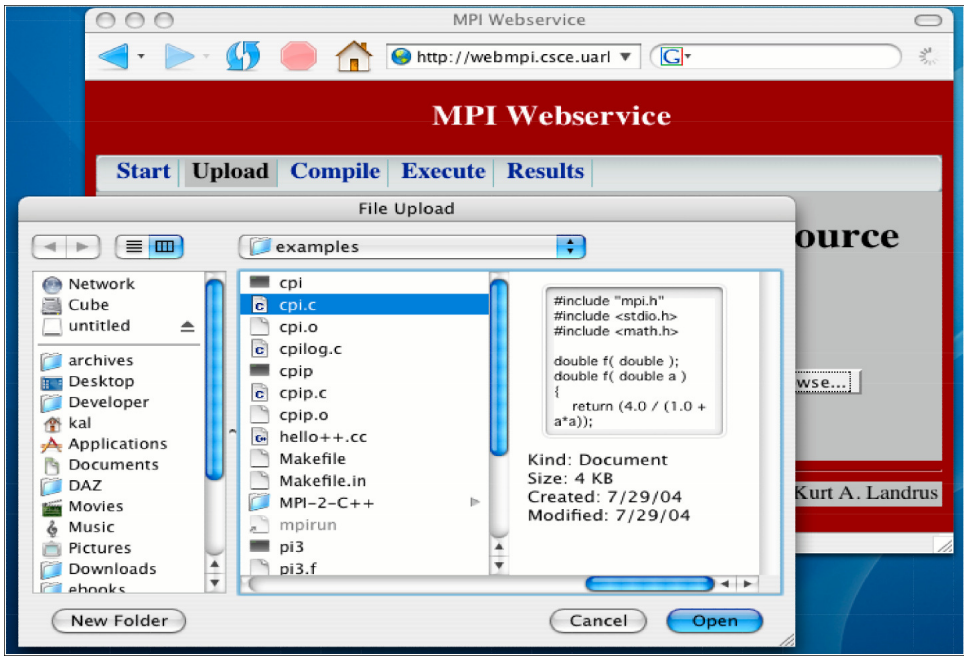


FIG. 3.3. *WebMPI File Upload Page*

One of the issues encountered in the development of WebMPI that does not come up in the same way for resources such as the GPN Repository is how to handle processing on the underlying cluster. In a normal MPI application, a user logs in to the head node of the Linux cluster using a local account and password. Any new files are created in the user's home directory and all processes that are created run under the user's account. The underlying Linux operating system protects the computer system from a stray process that creates too many files, or that creates files that are too large, or that consumes too much CPU time. In contrast, one of the features of Shibboleth is that it allows the protection of user privacy. While it is possible to pass a user name or account within the Shibboleth architecture when it is desired to do so for a particular application, Shibboleth works by not revealing the user's name or account in general. Access is typically based on group membership, or the possession of a particular attribute. With WebMPI this is a challenge in two ways. First, there needs to be a way to map an incoming user to a subdirectory on the underlying cluster, and secondly, there needs to be a way to map an incoming user to an account in which processes may execute. This is solved in a typical grid application that uses Globus Toolkit [9] by creating an account on the machine and then mapping the user's credentials (i. e., the Distinguished Name) to that account through the grid-mapfile. A Distinguished Name is not passed to the Service Provider by the Shibboleth protocol in general.

One of the attributes that can be maintained by the Identity Management system at the home institution is the PersonID that is a part of the eduPerson schema. The PersonID is an opague identifier that is guaranteed to be unique for a person. WebMPI maintains subdirectory integrity by mapping the PersonID to a subdirectory on the underlying cluster. By using a mapping in this way, a user may authenticate a second time and be mapped to the same user space representing the same subdirectory. For example, a user is able to start a long-running MPI application, and then return at a later time to view results. Note that for a user to be authorized to use WebMPI, the Identity Provider at the home institution of the user must release the PersonID attribute to the WebMPI resource.

User-level MPI processes are handled in WebMPI by executing all MPI applications in a single account named the webmpi account. In the interface, a user requests the execution of an MPI program. Since WebMPI is a web application, the suEXEC feature of the Apache web server can be used to execute the MPI program in the webmpi account on behalf of the user. Results are maintained in the appropriate subdirectory. This technique is secure in that only users who authenticate through the Shibboleth interface are allowed to execute programs; however, it does not allow the logging of usage of individual users. Other solutions to the process mapping problem are being explored, including integration with GridShib [10].

4. GPN Federation Infrastructure. The Shibboleth middleware component and the prototype applications are supported by several infrastructure components and design approaches in the GPN federation. The supporting infrastructure includes federation server support, structured management of attributes, the greatplains.net MACE namespace, and management of entitlements.

4.1. Federation Server Support. Early in the project, GPN institutions joined InQueue [11], a test federation organized by Internet2 for institutions who are learning how to use Shibboleth and the federated trust model. A federation organization supports the GPN federation process by providing two different types of servers and services. First, GPN members of InQueue initially configured their Identity Provider with a public key certificate from the Bossie Certificate Authority recommended by InQueue since this was easy and at no cost. The Bossie CA is not secure or suitable for production use, but does provide a simple and fast way to build a working prototype Shibboleth system. As GPN members have moved toward production use of their Identity Provider, the Bossie certificates have been replaced by certificates from Verisign and other production-quality CA's. InCommon [12] is a formal federation that has been developed to support a production CA and production use of the Shibboleth architecture. In general, in order for the identity and resource providers in GPN to continue to trust each other, they have to be configured with the public key of each CA that is in use and is trusted in a production setting by the GPN members.

A second server that a formal federation organization supports the GPN consortium is the WAYF, or a "Where Are You From?" server. In the Shibboleth protocol, when a user first contacts a Resource Provider through a web page, the Resource Provider must determine the home institution of the user for that person to be authenticated. The Shibboleth protocol redirects a request to the WAYF named by the Resource Provider. The WAYF, in turn, maintains a list of potential Identity Providers, allows the user to select one, and then redirects the user to the authentication process of the Identity Provider selected by the user.

The WAYF in InQueue currently supports dozens of participating organizations, which is distracting when trying to do testing with GPN resources. In addition, the InQueue WAYF does not meet the needs of GPN for the testing and configuration of the middleware test bed. As attributes have been tested, for example, two Identity Providers have been maintained at the University of Arkansas. The first remains integrated with the campus Identity Management system and is visible to all campus users, but the second is configured in a test mode to only allow access by certain test user accounts. GPN has built a test federation by implementing a test WAYF that lists the test Identity Provider. In turn, the resources also being tested are configured to redirect users to the test WAYF. This lightweight test environment provides a mechanism for introducing new resources and new types of attribute access without interfering with production use of Shibboleth on some campuses. As the middleware infrastructure of GPN grows the components in the lightweight test environment will need to be replaced by production versions of these components.

4.2. Structured Management of Attributes. As the resources in the GPN federation have become more complex, the need has arisen for a structured way of managing the attributes that are required for access to these resources. The EDUCAUSE/Internet2 eduPerson Task Force has defined a data structure (object class) that defines attributes that are useful for individuals in higher education. Among the attributes are an individual's institutional affiliation, and their relationship to the institution, such as faculty, student or staff. At a very high level, this provides a coarse-grained means for distinguishing groups within an institution, such as all faculty members or all students enrolled in a particular class in a particular semester. At this level, decisions about authorizations to utilize various services at an institution can be readily made.

The eduPerson Task Force identified the syntax and semantics of these attributes. The development of this object class is now managed by the MACE-Directory Working Group, which is encouraging wide-spread adoption of the attributes among institutions of higher education. While these attributes are common across institutions and as a proposed standard the defined object class is or can be quite useful to enable a wide variety of applications and services both within an institution and external to it.

The GPN group, upon reviewing the attribute fields, has a need for more fine-grained controls over authorizing access to specific, shared, collaborative resources and services that span several institutions among the Great Plains states. The values for attributes for most fields in the eduPerson object class are single-valued and/or predefined (e.g., faculty, staff, student, or member).

To accomplish the goals of the GPN middleware project, an attribute that can take on various values and, in fact, be multi-valued is desired. One field that has the required properties is the eduPersonEntitlement field. Thus, in these first steps of the middleware project the GPN group has used the eduPersonEntitlement field to

define the necessary values that facilitate the need for fine-grained decision-making when authorizing access to inter-institutional, collaborative resources. Use of the entitlement attribute in combination with other eduPerson attributes sets the stage for fine-grained control of authorizing access to specific resources and services.

In the current implementation the entitlements are defined as LDAP attributes using the eduPersonEntitlement definition. The entitlement attribute permits multiple values. The entitlement attribute is a semicolon separated string containing MACE registered values that are asserted and verified during the authorization process to grant or deny access to an entitlement dependent resource.

Shibboleth processes these attributes at both the Identity Provider (IdP) and the Service Provider (SP). The simplest way to release the attributes by the IdP is to use the sample attribute release policy which releases specified attributes to all requesters. However, it is possible to release attributes to some, but not all institutions. The SP must define what attributes it will accept, and from whom. In other words, an attribute acceptance policy might allow all GPN institutions to assert eduPersonEntitlement, but no others. Shibboleth by default disallows any institution from asserting attributes scoped to another institution. Also, an IdP may choose to only release certain values of an attribute to a particular SP. For example, a GPN SP might only be able to see GPN-related eduPersonEntitlement values, and not those related to other organizations.

In the Apache configuration for the SP, the required attributes are defined, and if the "ShibRequireAll" directive is specified, all attributes must be present. Otherwise, the default Apache behavior is to authorize access if any one of the attributes matches. These can be matched exactly, or a regular expression may be used to match a range of acceptable values. Note that if regular expressions are used, extreme caution should be paid to ensuring that unwanted matches do not mistakenly get accepted.

The GPN access to resources can be further mediated by a portal application that determines final access using the combination of entitlements that are asserted. Apache will permit users asserting any GPN MACE registered entitlement. This is achieved using a regular expression. Once the user is at the portal, they may be granted access to one or more resources (or none, as appropriate) depending on what entitlements are asserted. There are currently four resources at two different institutions that require the use of eduPersonEntitlements: the biomedical data and application service, biomedical computing resources, and the GPN Repository at the University of Missouri, and the WebMPI computational resource at the University of Arkansas.

4.3. Registration of the MACE Greatplains.net Namespace. The Great Plains Network (GPN) has registered the name urn:mace:greatplains.net with the Internet2 Middleware Architecture Committee for Education (MACE). This name is the top level of a hierarchical namespace controlled by the GPN for use in its collaboration efforts. In the case of GPN, this namespace includes specific entitlement values that are used to provide fine-grained access control to GPN defined resources.

As part of the MACE namespace registration process, a URL is provided to access online documentation for the registered namespace. In the case of the greatplains.net MACE namespace, the registered URL is: http://www.greatplains.net/mace-gpn. This web site defines the namespace and the entitlement attributes defined for use by the GPN group, as shown in Figure 4.1. The documentation for MACE as well as the corresponding IETF RFC (3613) that MACE is based is provided as links from this web page.

The GPN middleware project is using the MACE namespace to define and specify the eduPersonEntitlement information used to identify resources and authorization information needed to authorize access to specific resources and services in use among the GPN collaborating institutions. Individuals who have authenticated through Shibboleth (at their home institution) and who have eduPersonEntitlements that match the greatplains.net defined entitlements are granted access to the defined resources or services. Individuals without such entitlements are denied access to the corresponding resources and services even though they have been authenticated via Shibboleth from one of the collaborating institutions.

The MACE registered namespace for greatplains.net is critically important to this project since it provides a persistent URN naming convention under control of the GPN collaborating institutions. This mechanism creates a virtual organization consisting of a wide variety of individuals from a collection of institutions that does not fall into the normal classifications for individuals in a typical identity management system within or across multiple institutions. For example, the virtual organization can permit selected students, faculty and staff at various institutions to access services without granting access to all such groups of individuals from all of the institutions. In short, the defined entitlements provide a way to utilize middleware standards, while providing an extensible means for accommodating specific and unique needs of groups of individuals that can be easily tailored for fine-grained discriminating decisions implemented in application programs and systems.

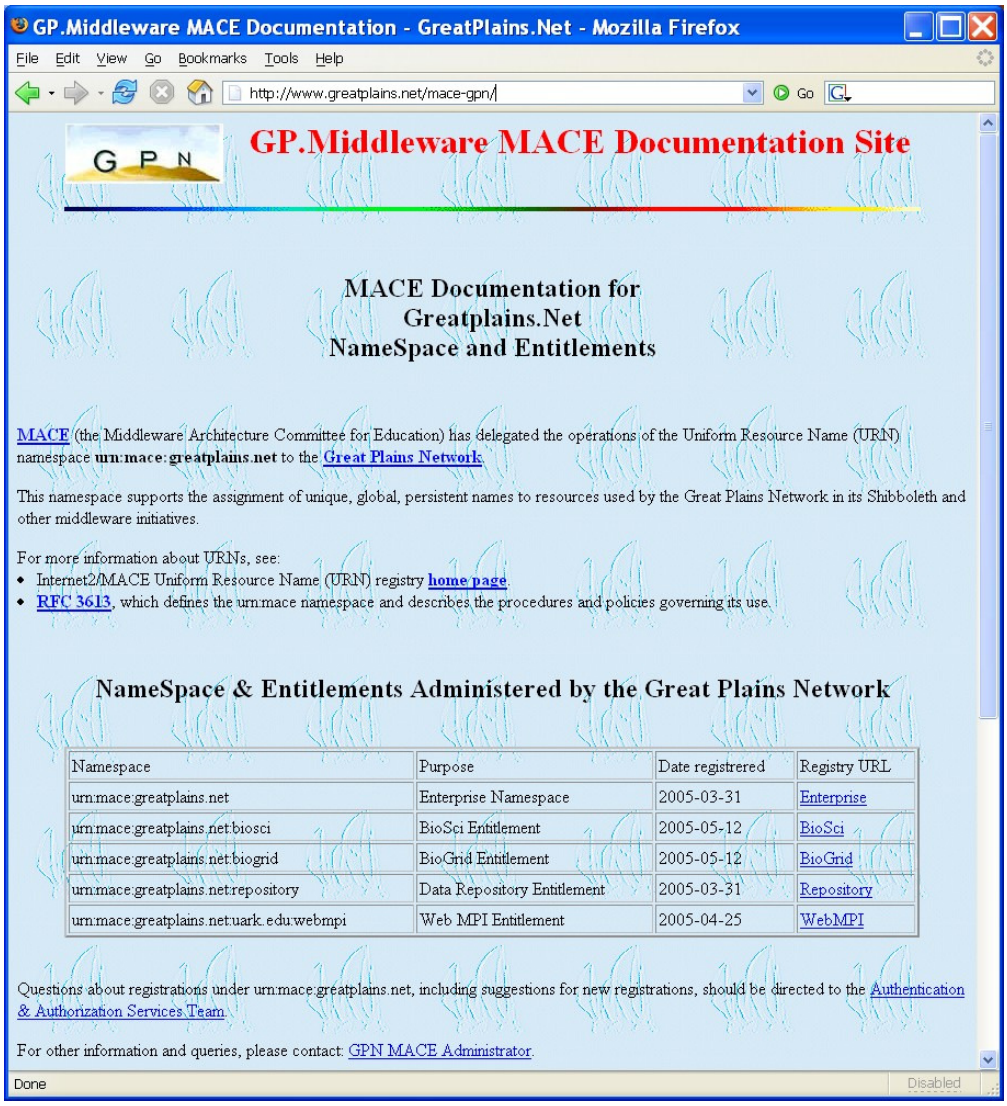


FIG. 4.1. *The GP.Middleware MACE Documentation Web Site*

4.4. Building an Attribute Architecture. The GPN middleware team is defining an attribute architecture to support the Great Plains Network Consortium in its efforts to create a regional collaboration environment among its members. This architecture is predicated on the eduPersonEntitlement attributes defined in the GPN registered MACE NameSpace. The MACE NameSpace: urn:mace:greatplains.net is central to facilitating the development of applications and services to be shared among individuals located at consortium member institutions.

Shibboleth is used for authentication into the virtual organization (i. e., greatplains.net) environment. The urn:mace:greatplains.net NameSpace is used for required authorizations to access the collaboration resources defined within the NameSpace via Shibboleth/EduPerson released entitlements. It is important to note that the enforcement of the entitlement rules for access is enforced in a split responsibility fashion. The first part of the entitlement syntax (urn:mace:greatplains.net) is enforced by the Shibboleth target Apache web server. That is, if the prefix of the entitlement for the GPN MACE NameSpace does not appear in the entitlement list at authentication time, access to the Shibboleth target is denied. The last part of the NameSpace entitlement (everything following the final colon in each entitlement in the list of entitlements asserted during the authentication) is used to enforce the specific authorizations an entity has within the defined entitlement NameSpace.

The presently defined entitlements consist of two groups; the entitlements requested for use by the University of Missouri-Columbia and the entitlements requested for use by the University of Arkansas. Conceptually, these entitlements are hierarchical and interoperable. And, globally these entitlements permit fine-grained control over selected resources or capabilities offered to users located at several institutions located in the Great Plains region.

The architecture and definitions presently active are accessible at:
`http://www.greatplains.net/mace-gpn`

The entitlements defined include:

- `urn:mace:greatplains.net:biosci` The BioSci attribute allows authenticated identities to access and utilize resources to support research activities in the biological and life sciences among the member institutions.
- `urn:mace:greatplains.net:biogrid` The BioGrid attribute allows authenticated identities to access and utilize region-wide grid computing resources to support research activities requiring grid computing access.
- `urn:mace:greatplains.net:repository` The Repository attribute allows authenticated identities to access and utilize a region-wide data repository for sharing documents, files and data among the participating members.
- `urn:mace:greatplains.net:uark.edu:webmpi` The WebMPI attribute allows authenticated identities to access and utilize the grid computing cluster at the University of Arkansas to develop, test, and run MPI-based parallel programs.

The three entitlements: BioSci, BioGrid, and Repository have a hierarchical relationship. BioSci is the superior (top level) entitlement that incorporates the BioGrid and Repository entitlements. That is, any authenticated entity that incorporates the BioSci entitlement automatically is presumed to also have the BioGrid and Repository entitlements as well. However, an authenticated entity with the BioGrid entitlement does not incorporate the BioSci entitlement nor the Repository entitlement. The same is true for the Repository entitlement. Namely, having the Repository attribute does not incorporate either the BioSci or the BioGrid attribute for authorization purposes.

The WebMPI entitlement has an interoperable relationship with the BioSci entitlement. Namely, at the University of Arkansas, an authenticated entity with the WebMPI attribute is granted access to the grid computing resource for developing MPI programs on a cluster machine. However, another authenticated entity with the BioSci attribute will be authorized for the WebMPI attribute when required or needed. Since our resources are in prototype phase this interoperation capability is not fully functional at this time.

Details of the meaning of the specific entitlements that have been defined can be found by accessing the appropriate links in the attribute table found at: `http://www.greatplains.net/mace-gpn`. These definitions are in active development and are subject to change at any time. The URN attributes are persistent, but the details of the authorizations they carry may change.

At this time, the BioSci, BioGrid, and Repository MACE entitlement attributes are associated with the Shibboleth Target: `https://crick.rnet.missouri.edu/GPN`. The WebMPI MACE entitlement is associated with the Shibboleth Target: `http://webmpi.csce.uark.edu`.

5. Synergism with Other Research Communities. As the GPN continues to build virtual organizations several opportunities have arisen. Synergistic activities among researchers in GPN provide motivation and opportunity to build on the middleware test bed, and our efforts have allowed us to build collaborative research relationships with members of the larger middleware community.

5.1. Regional Collaboration. A group of researchers within the GPN group of institutions has begun discussion and planning to promote biomedical application development and collaboration using the middleware infrastructure being developed within the GPN. This effort is in its infancy, but is due in part to the expanding efforts of the GPN to encourage researchers to become active in projects that enable applications of specific interest to the researchers.

Using the middleware infrastructure, the researchers can concentrate on the deploying of applications and application development of biomedical software for research use. The middleware infrastructure also enables the group to utilize inter-institutional resources, such as Grid technology, to further the individual research efforts. In fact, the GPN MACE entitlements (BioSci and BioGrid), were set up to support the efforts of this group and other groups in the future.

The biomedical sciences group is planning a pilot implementation to be used to prepare for submissions of research proposals to external funding agencies. This planning process includes selecting appropriate applications suited for this activity, develop a prototype pilot implementation and collect data to be used in proposal writing.

A second group of researchers in the area of environmental sciences, water resource modeling and watershed research is also keenly interested in using the developing middleware infrastructure in support of research. The environmental science group has a history of collaboration that precedes the GPN project. The group has met regularly and participated in several proposals to date.

Within the GPN environmental science research community several broad-based water modeling applications have been developed. These applications are used across the region to predict the effects of changes due to weather events, farming practice, and urbanization. Some of the applications use data that is collected across a broad geographic region and shared among the researchers in the area. The needs of this virtual organization to share data, applications, and research results in an authorized manner are helping to drive the development of GPN middleware.

5.2. Integration with Grid Computing Middleware. Grid computing has been one of the driving forces for the GPN middleware group. Once the ability to use Shibboleth for authentication across institutions was accomplished, the ability to access shared resources and applications has become an important part of the group's efforts. While Shibboleth provides an architecture and protocol for authentication and authorization, many other capabilities are required for resource sharing in the region, including service discovery, service monitoring, data management, workflow management, and so on.

Globus Toolkit is the default middleware for providing grid capabilities. Since the formal completion of the ETR project in early 2006 our efforts have shifted somewhat to unifying resources using Globus so that we will have a grid that incorporates these capabilities. We currently have a set of cluster resources at four different institutions that have been installed with Globus Toolkit.

Globus is based on Grid Security Infrastructure (GSI) [13], which relies on X.509 certificates for the authentication and authorization of each user and each resource in the grid. A user certificate contains, among other information, a globally-unique Distinguished Name. The Distinguished Name is used directly in Access Control Lists to determine if a user is allowed to access a particular resource. One or more common Certificate Authorities must sign, directly or indirectly through a proxy certificate, the certificates held by the user and the resource.

GridShib is a software product that allows a middleware infrastructure built using the Globus Toolkit to incorporate the attribute-based authorization protocol of Shibboleth [10]. The software package includes two plug-ins, one that interfaces to Globus Toolkit version 4, and one that interfaces to Shibboleth. The primary purpose of the Globus plug-in is to request attributes about a requesting user from the Shibboleth Attribute Authority (a software component at the Identity Provider site). The Globus plug-in parses the attributes, caches them, makes an access control decision and then allows or disallows access to the resource. The Shibboleth plug-in allows the Attribute Authority to map a Distinguished Name from a GSI X.509 certificate to a local user identifier, and to query the attribute database based on the local user identifier.

GridShib assumes that a user (a grid client) and the grid service both possess an X.509 credential. That is, a user logs in using a Globus login process to acquire a certificate or proxy certificate. GridShib also assumes that the user has an account with a home institution a Shibboleth Identity Provider. The Identity Provider and the grid Service Provider each have a globally unique providerID, and the Service Provider and Identity Provider use a common metadata format and exchange metadata out-of-band.

A beta release of GridShib software became available in fall, 2005. The GPN virtual organization is in the process of incorporating these components into the existing Shibboleth and Globus infrastructure.

5.3. Challenges of Managing Entitlements in Virtual Organizations. Defining entitlements and utilizing them in applications to provide fine-grained discrimination for authorization is challenging. A truly significant aspect of this challenge involves the management of the entitlements in identity management providers (IdP) across a virtual organization. A large number of policy and technical decisions must be developed to allow entitlement data to be incorporated into separate IdPs governed by different business and policy models.

In the current test bed the eduPersonEntitlement field is used to store user attributes in local campus LDAP directories, and the Shibboleth Attributed Authority retrieves these attributes for authorization to a server. The eduPersonEntitlement field was readily available at the time of the development of the GPN middleware test

bed, but this approach of storing virtual organization attributes in local campus LDAP directory services is not a long-term scalable solution. How to management entitlements and user attributes is a major topic of discussion in NMI, Shibboleth, Grid, and other efforts. This problem is also the focus of a recent NSF-sponsored workshop on Campus Research Computing Cyberinfrastructure.

In cooperation with the GPN, NMI-EDIT and Internet2/MACE Signet groups, we are looking at the issues and means for virtual organizations to authorize, request and manage authorization data that must be incorporated into separate idPs to be effective. For example, currently the GPN entitlement values must be contained in an identity provider's identity management database for individuals who are authenticated at their home institution and are members of the virtual organization. The virtual organization should be the ones to authorize entitlements in a given idP, but idPs are reluctant to empower anyone but their local identity management team to update anything in their database. Similarly, a single individual may have several identities/roles in an organization and it must be determined which identity is to be granted access in an external virtual organization. The current thinking is that the entitlements must be maintained in a database that is separate from the campus identity management database but must somehow be coordinated with it. These issues are of significant interest to a wide range of institutions, government agencies, and funded research projects. We will be working with the NMI effort to incorporate new technologies as better approaches are developed and to develop or deploy the prototype that will be used in the test bed.

Signet is a developing core middleware component that assists in the management of privileges [14]. Signet provides: 1) a standard user interface for privilege administrators; 2) a consistent, simplified policy definition via roles and including integration with core campus organizational data; 3) improved visibility, understandability and auditability of privilege information; and 4) standard interfaces to other infrastructure services.

Grouper is an open source toolkit for managing groups, and is designed to work complementary with Signet and other middleware components [15]. Grouper is designed to manage multiple sources of group information, such as what may be found among the several GPN institutions and sets of user groups who desire to access the various GPN and other resources. In addition to GridShib, we anticipate that Signet and Grouper will become core components of the developing GPN middleware test bed.

6. Conclusions. The development and the deployment of middleware infrastructure in the GPN is an on-going project. The accomplishments can be described both in terms of the technological accomplishments described in this paper as well as the progress in inter-institutional collaboration. Several lessons have been learned during the task of building a middleware infrastructure in such a heterogeneous and widely-distributed environment, including:

1. Collaboration among individuals is essential. We have supported collaboration with weekly teleconferences and a mailing list. Since the tools are sometimes difficult to install and use it is essential that individuals have community support for asking technical questions. Individuals are much more comfortable contacting colleagues that they already know for help in installing and configuring a new tool than they are in contacting an unknown person. A high level of collaboration allows the project to move along much more quickly.
2. A middleware project among cooperating institutions that lack a central hierarchical administration will be limited by the level of support of the institutions in the region. Specifically, if a university infrastructure resource, such as a campus LDAP server or campus firewall configuration, is required to be a part of the architecture, then staff employees must have time allocated during the week that is specifically devoted to the middleware project. In the case of faculty, the tenure reward system must recognize the contribution to the middleware project as evidence of productive research and teaching. The benefits of middleware infrastructure are long-term, and so an institution must take a long-term view of its contribution to the project and not expect that the individuals efforts will immediately benefit the institution.
3. Communication and constant education about the on-going changes in the middleware community are essential. For example, the release of the GridShib software component for unifying Shibboleth and Globus security models, the development of Signet and Grouper, and the current discussions on Campus Research Computing Cyberinfrastructure are having an impact on the test bed. Some members of the GPN middleware team are participating in a NMI-EDIT Signet and Grouper Early Adopters Workshop and will bring the ideas from those organizations to the larger middleware group.

The GPN ETR project has accomplished the original goal of the deployment of a middleware test bed on a small number of campuses. These prototypes and test components are a building block for further work. The GPN middleware group is in an excellent position to collaborate on new grid and middleware computing projects and have an opportunity to test the evolving mechanisms in a live test bed with multiple participating institutions operating as a virtual organization.

The next goals for GPN include the development of a robust and scalable architecture for attribute management and fine-grained access control, the development of applications and a community of users that can utilize the storage and computing resources that have been developed, and move towards a production quality set of tools in support of research and education in the region.

7. Acknowledgements. We would like to acknowledge the work and contributions of Kathryn Huxtable, Denis Hancock, Larry Sanders, Shafquat Bhuiyan, Arturo Guillen, Abdul Khambati, Kurt Landrus, James McCartney, Linh Ngo, and the support of our colleagues from the Great Plains Network Consortium, without whose help this project would not have been possible.

Funding for this project is provided in part by an Extending the Reach (ETR) grant from Educause on behalf of the NMI-EDIT Consortium of Internet2, Educause, and SURA, and with several statewide university systems and regional networks. See `http://archie.csce.uark.edu/gpn/` for more information about this funded project.

REFERENCES

[1] Great Plains Network Consortium (2006), `http://www.greatplains.net/`
[2] Enterprise and Desktop Integration Technologies (EDIT) Consortium (2006), `http://www.nmi-edit.org/index.cfm`
[3] Shibboleth Project, Internet2 Middleware (2006), `http://shibboleth.internet2.com`
[4] Amy Apon, Greg Monaco, Gordon Springer, and Kathryn Huxtable, *Great Plains Network: Building the Regional Middleware Infrastructure,* NMI-EDIT Case Study Series, January, 2006. `http://archie.csce.uark.edu/gpn/publications/GPN-Building-NMI-case-study-final.pdf`
[5] Gordon Springer, Shafquat Bhuiyan, and Arturo Guillen, *Great Plains Network: Integrating Shibboleth, Grid and Bioinformatics,* NMI-EDIT Case Study Series, January, 2006. `http://beagle.rnet.missouri.edu/GPN/Docs/MUCaseStudy.pdf`
[6] Tom Barton, Jim Basney, Tim Freeman, Tom Scavo, Frank Siebenlist, Von Welch, Rachana Ananthakrishnan, Bill Baker, Monte Goode, and Kate Keahey, *Identity Federation and Attribute-based Authorization through the Globus Toolkit, Shibboleth, Gridshib, and MyProxy,* In 5th Annual PKI R & D Workshop, April 2006. `http://grid.ncsa.uiuc.edu/papers/gridshib-pki06-final.pdf`
[7] EduPerson Object Class Specification, Internet2, April, 2006. `http://www.nmi-edit.org/eduPerson/draft-internet2-mace-dir-eduperson-latest.html`
[8] Subversion project (2006). `http://subversion.tigris.org/`
[9] Ian Foster and Carl Kesselman, *Globus: A Metacomputing Infrastructure Toolkit,* Intl J. Supercomputer Applications, 11(2):115-128, 1997.
[10] Welch, Von, Tom Barton, Kate Keahey, Frank Siebenlist, *Attributes, Anonymity, and Access: Shibboleth and Globus Integration to Facilitate Grid Collaboration,* Proceedings of the 4th Annual PKI R & D Workshop, 2005. `http://grid.ncsa.uiuc.edu/papers/gridshib-pki05-final.pdf`
[11] InQueue Home (2006), `http://inqueue.internet2.edu/`
[12] Incommon Federation (2006), `http://www.incommonfederation.org/`
[13] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke, *Security for grid services,* In Twelfth International Symposium on High Performance Distributed Computing (HPDC-12). IEEE Computer Society Press, 2003.
[14] Signet Home Page (2006). `http://middleware.internet2.edu/signet/`
[15] Grouper Working Group (2006). `http://middleware.internet2.edu/dir/groups/grouper/`

Edited by: Dana Petcu.
Received: February 08, 2006.
Accepted: May 25, 2006.

CALL FOR PAPERS

Scalable Computing: Practice and Experience Journal (`http://www.scpe.org`) is planning a Special Issue on "Parallel Evolutionary Algorithms".

Evolutionary Algorithms (EA) are computer-based solving systems, which use evolutionary computational models as a key element in their design and implementation. They have a conceptual base of simulating the evolution of individual structures via the Darwinian natural selection process. EA's has been widely accepted for solving several important practical applications in engineering, business, commerce etc. As we all know, the problems of the future will be more complicated in terms of complexity and data volume. Generally, evolutionary computation requires a massive computational effort to yield efficient and competitive solution to real-size engineering problems.

This special issue is focussed on all theoretical and practical aspects related to the ***parallelisation* of evolutionary computation.** The topics of interest include, but are not limited to:

- parallel genetic operators,
- parallel fitness evaluation,
- evolutionary multi-objective optimization in a parallel environment,
- multipopulation and coevolutionary approaches,
- synchronous and asynchronous parallel distributed evolutionary algorithms,
- distributed/parallel genetic programming,
- distributed and cellular evolutionary algorithms,
- hybrid distributed/parallel algorithms (evolutionary algorithms hybridized with other meta-heuristics),
- parallel evolutionary algorithms implementations,
- applications, including real world applications,
- and the others connected.

All papers will be peer reviewed by three independent referees.

The time schedule for this publication is as follows:

- submission deadline: September 15th, 2006
- authors notification: November 15th, 2006
- camera-ready submission: December 15th, 2006

For further details, please do not hesitate to contact the editors:
Ajith Abraham (`ajith.abraham@ieee.org`)
Paweł B. Myszkowski (`pmyszkowski@wsiz.wroc.pl`)
Shahram Rahimi (`rahimi@cs.siu.edu`)

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:

- programming environments,
- debugging tools,
- software libraries.

Performance:

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2$_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the PDCP WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.