SCALABLE COMPUTING
Practice and Experience

Special Issue: Practical Aspects of High-Level Parallel Programming

Editor: Frédéric Loulergue

Volume 6, Number 4, December 2005

ISSN 1895-1767

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

Scalable Computing: Practice and Experience

Volume 6, Number 4, December 2005

TABLE OF CONTENTS

Scalable Computing: Practice and Experience
Volume 6, Number 4, p. iii. http://www.scpe.org
ISSN 1895-1767

GUEST EDITOR'S INTRODUCTION

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

This special issue of presents recent work of researchers in these fields. These articles are extended and revised versions of papers presented at the first international workshop on Practical Aspects of High-Level Parallel Programming (PAPP), affiliated to the International Conference on Computational Science (ICCS 2004). 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, visualisations 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 for parallel and grid computing and computational science researchers who are potential users of these languages and tools.

One concern in the development of parallel programs is to predict the performances of the programs from the source code in order to be able to optimize the programs or to fit the resources needed by the programs to the resources offered by the architecture. In their paper, *Evaluating the performance of pipeline-structured parallel programs with skeletons and process algebra*, Anne Benoît et al., propose a framework to evaluate the performance of structured parallel programs with skeletons and process algebra. Frédéric Gava in *External Memory in Bulk-Synchronous Parallel ML* provides an extension of the Bulk Synchronous Parallel ML library by input/output operations on disks, together with an extension of the Bulk Synchronous Parallel model.

Another direction of research is to set constraints on the resources used by the programs. Stephen Gilmore et al. designed and developed the Camelot language which is a resource-bounded functional programming language which compiles to Java byte code to run on the Java Virtual Machine. Their paper *Extending resource-bounded functional programming languages with mutable state and concurrency* extends Camelot to include language support for Camelot-level threads and extends the existing Camelot resource-bounded type system to provide safety guarantees about the heap usage of Camelot threads. Franck Pommereau's previous work is about high-level Petri nets with a notion of time, called causal time, used for the specification and the verification of systems with time constraints. In his paper *Petri nets as Executable Specifications of High-Level Timed Parallel Systems* he presents a step forward the use of this formalism for execution purposes: an algorithm for the execution of a restricted class of high-level Petri nets with causal time.

High-level programming languages aim at easing the programming of systems. This should not hinder the predictability and the efficiency of programs. Joël Falcou and Jocelyn Sérot designed a high-level library C++ for the programming of the SIMD component of the Power PC processors, which is much simpler to use that lower level specific libraries but with a very good efficiency. Their EVE library is thus a very good practical choice for the programming of such hardware.

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: Rob Bisseling (Univ. of Utrecht, The Netherlands), Matthieu Exbrayat (Univ. of Orléans, France), Sergei Gorlatch (Univ. of Muenster, Germany), Clemens Grelck (Univ. of Luebeck, Germany), Kevin Hammond (Univ. of St. Andrews, UK), Zhenjiang Hu (Univ. of Tokyo, Japan), Quentin Miller (Miller Research Ltd., UK), Susanna Pelagatti (Univ. of Pisa, Italy), Alexander Tiskin (Univ. of Warwick, UK). I also thank the other referees for their efficient help: Martin Alt, Frédéric Gava and Sven-Bodo Scholz. Finally I thank all authors who submitted papers for their interest in the workshop, the quality and variety of 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 6, Number 4, pp. 1–16. http://www.scpe.org
ISSN 1895-1767

EVALUATING THE PERFORMANCE OF PIPELINE-STRUCTURED PARALLEL PROGRAMS WITH SKELETONS AND PROCESS ALGEBRA*

ANNE BENOIT†, MURRAY COLE , STEPHEN GILMORE , AND JANE HILLSTON

Abstract. We show in this paper how to evaluate the performance of pipeline-structured parallel programs with skeletons and process algebra. Since many applications follow some commonly used algorithmic skeletons, we identify such skeletons and model them with process algebra in order to get relevant information about the performance of the application, and to be able to take good scheduling decisions. This concept is illustrated through the case study of the pipeline skeleton, and a tool which generates automatically a set of models and solves them is presented. Some numerical results are provided, proving the efficacy of this approach.

Key words. Algorithmic skeletons, pipeline, high-level parallel programs, performance evaluation, process algebra, PEPA Workbench.

1. Introduction. One of the most promising technical innovations in present-day computing is the invention of grid technologies which harness the computational power of widely distributed collections of computers [8]. Designing an application for the Grid raises difficult issues of resource allocation and scheduling (roughly speaking, how to decide which computer does what, and when, and how they interact). These issues are made all the more complex by the inherent unpredictability of resource availability and performance. For example, a supercomputer may be required for a more important task, or the Internet connections required by the application may be particularly busy.

In this context of grid programming, a skeleton-based approach [5, 16, 7] recognizes that many real applications draw from a range of well-known solution paradigms and seeks to make it easy for an application developer to tailor such a paradigm to a specific problem. Powerful structuring concepts are presented to the application programmer as a library of pre-defined 'skeletons'. As with other high-level programming models the emphasis is on providing generic polymorphic routines which structure programs in clearly-delineated ways. Skeletal parallel programming supports reasoning about parallel programs in order to remove programming errors. It enhances modularity and configurability in order to aid modification, porting and maintenance activities. In the present work we focus on the Edinburgh Skeleton Library (eSkel) [6]. eSkel is an MPI-based library which has been designed for SMP and cluster computing and is now being considered for grid applications using grid-enabled versions of MPI such as MPICH-G2 [14].

The use of a particular skeleton carries with it considerable information about implied scheduling dependencies. By modelling these with stochastic process algebras such as Performance Evaluation Process Algebra [13], and thereby being able to include aspects of uncertainty which are inherent to grid computing, we believe that we will be able to underpin systems which can make better scheduling decisions than less sophisticated approaches. Most significantly, since this modelling process can be automated, and since grid technology provides facilities for dynamic monitoring of resource performance, our approach will support *adaptive* rescheduling of applications.

Stochastic process algebras were introduced in the early 1990s as a compositional formalism for performance modelling. Since then they have been successfully applied to the analysis of a wide range of systems. In general analysis is based on the generation of an underlying continuous time Markov chain (CTMC) and derivation of its steady state probability distribution. This vector records the likelihood of each potential state of the system, and can in turn be used to derive performance measures such as throughput, utilisation and response time. Several stochastic process algebras have appeared in the literature; we use Hillston's Performance Evaluation Process Algebra (PEPA) [13].

Some related projects obtain performance information from the Grid using benchmarking and monitoring techniques [4, 17]. In the ICENI project [9], performance models are used to improve the scheduling decisions, but these are just graphs which approximate data obtained experimentally. Moreover, there is no upper-level layer based on skeletons in any of these approaches.

*This work is part of the ENHANCE project, funded by the United Kingdom Engineering and Physical Sciences Research council grant number GR/S21717/01.

†School of Informatics, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK. enhancers@inf.ed.ac.uk, http://groups.inf.ed.ac.uk/enhance/

Other recent work considers the use of skeleton programs within grid nodes to improve the quality of cost information [1]. Each server provides a simple function capturing the cost of its implementation of each skeleton. In an application, each skeleton therefore runs only on one server, and the goal of scheduling is to select the most appropriate servers within the wider context of the application and supporting grid. In contrast, our approach considers single skeletons which span the Grid. Moreover, we use modelling techniques to estimate performance.

Our main contribution is based on the idea of using performance models to enhance the performance of grid applications. We propose to model skeletons in a generic way to obtain significant performance results which may be used to reschedule the application dynamically. To the best of our knowledge, this kind of work has not been done before. We show in this paper how we can obtain significant results on a first case study based on the pipeline skeleton. An earlier version of this paper is published in the proceedings of the workshop on Practical Aspects of High-level Parallel Programming (PAPP04), part of the International Conference on Computational Science (June 7-9, 2004, Kraków, Poland) [2]. In this extended version a presentation of PEPA is included; the model resolution and the tool AMoGeT are described more precisely; and more experimental results are exposed.

In the next section, we present the pipeline and a model of the skeleton. Then we explain how to solve the model with the PEPA Workbench in order to get relevant information (Section 3). In Section 4 we present a tool which automatically determines the best mapping to use for the application, by first generating a set of models, then solving them and comparing the results. Some numerical results on the pipeline application are provided in Section 5, and the feasibility of this approach is discussed in Section 6. Finally we give some conclusions.

2. The pipeline skeleton. Many parallel algorithms can be characterized and classified by their adherence to one or more of a number of generic algorithmic skeletons [16, 5, 7]. We focus in this paper on the concept of pipeline parallelism, which is of well-proven usefulness in several applications. We recall briefly the principle of the pipeline skeleton. Then we introduce the process algebra PEPA [13] and we explain how we can model the pipeline with PEPA. Finally, we show in Section 2.4 the state transition diagram of a three stage pipeline.

2.1. The principle of pipeline. In the simplest form of pipeline parallelism [6], a sequence of N_s *stages* process a sequence of *inputs* to produce a sequence of *outputs* (Fig. 2.1).

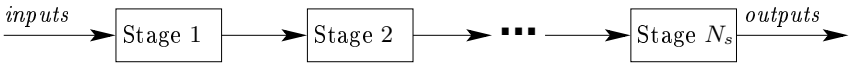


FIG. 2.1. *The pipeline application*

Each input passes through each stage in the same order, and the different inputs are processed one after another (a stage cannot process several inputs at the same time). Note that the internal activity of a stage may be parallel, but this is transparent to our model. In the remainder of the paper we use the term "processor" to denote the hardware responsible for executing such activity, irrespective of its internal design (sequential or parallel).

We consider this application class in the context of computational grids, and so we want to map it to our computing resources, which consist of a set of potentially heterogeneous processors interconnected by a heterogeneous network.

It is well known that a computing pipeline performs most effectively when the workload is well balanced across stages and there are a large enough number of inputs to amortize the costs of filling and draining. Our work directly addresses the first of these issues, by facilitating exploration of the stage-to-processor mapping space. The second issue remains the responsibility of the programmer: our approach assumes that running the application will take long enough for the system to reach an equilibrium behaviour. The models help us to study this steady state behaviour.

Considering the pipeline application in the eSkel library [6], we focus here on a pipeline variant which requires that each stage produces exactly one output for each input.

We now go on to present the PEPA language which we will use to model the pipeline application. The presentation below is necessarily brief and rather informal. For full details the reader is referred to [13]. The operational semantics can also be found in Appendix A.

2.2. Introduction to PEPA. The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. Timing information is associated with each activity. Thus, when enabled, an activity $a = (\alpha, r)$ will delay for a period sampled from the negative exponential distribution which has parameter r. If several activities are enabled concurrently, either in competition or independently, we assume that a *race condition* exists between them. The component combinators, together with their names and interpretations, are presented informally below.

Prefix: The basic mechanism for describing the behaviour of a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as S.

Choice: The choice combinator captures the possibility of competition between different possible activities. The component $P + Q$ represents a system which may behave either as P or as Q. The activities of both P and Q are enabled. The first activity to complete distinguishes one of them: the other is discarded. The system will behave as the derivative resulting from the evolution of the chosen component.

Constant: It is convenient to be able to assign names to patterns of behaviour associated with components. Constants are components whose meaning is given by a defining equation. For example, $P \stackrel{def}{=} (\alpha, r).P$ defines a component which performs activity α at rate r, forever.

Hiding: The possibility to abstract away some aspects of a component's behaviour is provided by the hiding operator, denoted P/L. Here, the set L of visible action types identifies those activities which are to be considered internal or private to the component and which will appear as the unknown type τ.

Cooperation: In PEPA direct interaction, or *cooperation*, between components is the basis of compositionality. The set which is used as the subscript to the cooperation symbol, the *cooperation set* L, determines those activities on which the *co-operands* are forced to synchronise. For action types not in L, the components proceed independently and concurrently with their enabled activities. However, an activity whose action type is in the cooperation set cannot proceed until both components enable an activity of that type. The two components then proceed together to complete the *shared activity*. The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity (for details see [13]). We write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty.

In some cases, when an activity is known to be carried out in cooperation with another component, a component may be *passive* with respect to that activity. This means that the rate of the activity is left unspecified (denoted $\top$) and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronised in the final model.

The dynamic behaviour of a PEPA model is represented by the evolution of its components, either individually or in cooperation. The form of this evolution is governed by a set of formal rules which give an operational semantics of PEPA terms (see [13]). Thus, as in classical process algebra, the semantics of each term in PEPA is given via a labelled *multi-transition* system (the multiplicities of arcs are significant). In the transition system a state corresponds to each syntactic term of the language, or *derivative*, and an arc represents the activity which causes one derivative to evolve into another. The complete set of reachable states is termed the *derivative set* of a model and these form the nodes of the *derivation graph* which is formed by applying the semantic rules exhaustively.

The derivation graph is the basis of the underlying Continuous Time Markov Chain (CTMC) which is used to derive performance measures from a PEPA model. The graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying CTMC. Each derivative is then a state in the CTMC. The *transition rate* between two derivatives P and Q in the derivation graph is the rate at which the system changes from behaving as component P to behaving as Q. It is the sum of the activity rates labelling arcs connecting node P to node Q.

2.3. Pipeline model. To model a pipeline application, we decompose the problem into the stages, the processors and the network. The model is expressed in PEPA (cf. Section 2.2).

The stages
The first part of the model is the *application model*, which is specified independently of the resources on which the application will be computed. We define one PEPA component per stage. For $i = 1..N_s$, the component $Stage_i$ works sequentially. At first, it gets data (activity $move_i$), then processes it (activity $process_i$), and finally moves the data to the next stage (activity $move_{i+1}$).

$$Stage_i \stackrel{def}{=} (move_i, \top).(process_i, \top).(move_{i+1}, \top).Stage_i$$

All the rates are unspecified, denoted by the distinguished symbol $\top$, since the processing and move times depend on the resources where the application is running. These rates will be defined later, in another part of the model.

The pipeline application is then defined as a cooperation of the different stages over the $move_i$ activities, for $i = 2..N_s$.

The activities $move_1$ and $move_{N_s+1}$ represent, respectively, the arrival of an input in the application and the transfer of the final output out of the pipeline. They do not represent any data transfer between stages, so they are not synchronizing the pipeline application. Finally, we have:

$$Pipeline \stackrel{def}{=} Stage_1 \underset{\{move_2\}}{\bowtie} Stage_2 \underset{\{move_3\}}{\bowtie} \ldots \underset{\{move_{N_s}\}}{\bowtie} Stage_{N_s}$$

The processors
We consider that the application must be mapped on a set of N_p processors. Each stage is processed by a given (unique) processor, but a processor may process several stages (in the case where $N_p < N_s$). In order to keep the model simple, we decide to put information about the processor (such as the load of the processor or the number of stages being processed) directly in the rate μ_i of the activities $process_i$, $i = 1..N_s$ (these activities have been defined for the components $Stage_i$).

Each processor is then represented by a PEPA component which has a cyclic behaviour, consisting of processing sequentially inputs for a stage. Some examples follow.

- In the case when $N_p = N_s$, we map one stage per processor:

$$Processor_i \stackrel{def}{=} (process_i, \mu_i).Processor_i$$

- If several stages are processed by a same processor, we use a choice composition. In the following example ($N_p = 2$ and $N_s = 3$), the first processor processes the two first stages, and the second processor processes the third stage.

$$\begin{aligned} Processor_1 &\stackrel{def}{=} (process_1, \mu_1).Processor_1 + (process_2, \mu_2).Processor_1 \\ Processor_2 &\stackrel{def}{=} (process_3, \mu_3).Processor_2 \end{aligned}$$

Since all processors are independent, the set of processors is defined as a parallel composition of the processor components:

$$Processors \stackrel{def}{=} Processor_1 || Processor_2 || \ldots || Processor_{N_p}$$

The network
The last part of the model is the network. We do not need to directly model the architecture and the topology of the network for what we aim to do, but we want to get some information about the efficiency of the link connection between pairs of processors. This information is given by affecting the rates λ_i of the $move_i$ activities ($i = 1..N_s + 1$).

— λ_1 represents the connection between the user (providing inputs to the pipeline) and the processor hosting the first stage.

— For $i = 2..N_s$, λ_i represents the connection between the processor hosting stage $i-1$ and the processor hosting stage i.

— λ_{N_s+1} represents the connection between the processor hosting the last stage and the user (the site where we want the output to be delivered).

Note that λ_i will encode information both about the load on the links and the size of the data processed by $process_{i-1}$. When the data is "transferred" on the same computer, the rate is really high, meaning that the connection is fast (compared to a transfer between different sites).

The network is then modelled by the following component:

$$Network \stackrel{def}{=} (move_1, \lambda_1).Network + \cdots + (move_{N_s+1}, \lambda_{N_s+1}).Network$$

The pipeline model

Once we have defined the different components of our model, we just have to map the stages onto the processors and the network by using the cooperation combinator. For this, we define the following sets of action types:

— $L_p = \{process_i\}_{i=1..N_s}$ to synchronize the *Pipeline* and the *Processors*

— $L_m = \{move_i\}_{i=1..N_s+1}$ to synchronize the *Pipeline* and the *Network*

$$Mapping \stackrel{def}{=} Network \underset{L_m}{\bowtie} Pipeline \underset{L_p}{\bowtie} Processors$$

PEPA input file

An example of an input file for the PEPA Workbench can be found in Appendix B.

2.4. State transition diagram for the pipeline model. Figure 2.2 represents the state transition diagram of a three stage, three process pipeline. This picture shows all of the possible interleavings of the components of the model with arcs of various kinds showing the different types of transitions from state to state.

In Table 2.1 we show the correspondence between the state numbers in Figure 2.2 and the PEPA terms. Since the PEPA terms are long we have omitted the cooperation sets, showing only the local state of each component. Moreover to keep the table compact we have named the derivatives of the *Stage* components as follows:

$$\begin{aligned} Stage_{i0} &\stackrel{def}{=} (move_i, \top).Stage_{i1} \\ Stage_{i1} &\stackrel{def}{=} (process_i, \top).Stage_{i2} \\ Stage_{i2} &\stackrel{def}{=} (move_{i+1}, \top).Stage_{i0} \end{aligned}$$

3. Solving the models. One reason to work with a formal modelling language such as PEPA is that models are unambiguous and can serve to support reliable communication between those who design systems, those who develop them and those who maintain them. Another reason to work with a formal modelling language is that formal models can be automatically processed by tools in order to derive information from them which otherwise would have to be produced by manual calculation or reasoning.

The tool which we have used for processing our PEPA models and computing the steady-state probability distribution of our system is the PEPA Workbench. A full description of the functioning of this software can be found in [11]; the reference manual for the latest release is [12]. We include a brief description of the functioning of the Workbench in Appendix C.1 in order to make the present paper self-contained.

Notice however that the steady-state probability distribution of the system is rarely the desired result of the performance analysis process and so to progress we must identify a significant *performance result*. The performance result that is pertinent for the pipeline application is the throughput of the $process_i$ activities ($i = 1..N_s$). Since data passes sequentially through each stage, the throughput is identical for all i, and we need to compute only the throughput of $process_1$ to obtain significant results. This is done by adding the steady-state probabilities of each state in which $process_1$ can happen, and multiplying this by μ_1.

We have made some changes to the Java edition of the PEPA Workbench in order to allow the user to specify performance results which will then be automatically computed. This new functionality is then used to compute numerical results from the pipeline models. Some more technical details are provided in Appendix C.2.

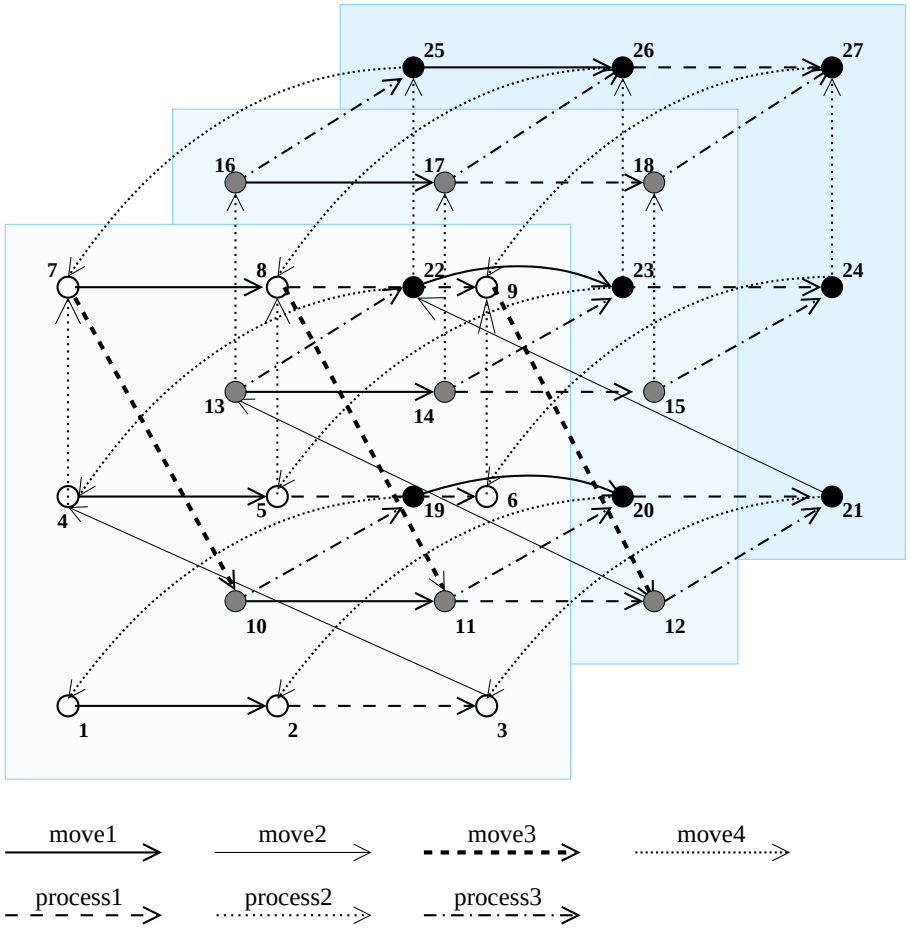


FIG. 2.2. *State transition diagram of a three stage, three process pipeline with states numbered according to Table 2.1*

4. AMoGeT: The Automatic Model Generation Tool. We investigate in this paper how to enhance the performance of grid applications with the use of algorithmic skeletons and process algebras. To do this, we have created a tool which automatically generates performance models for the pipeline case study, and then solves the models. These results could be used to reschedule the application.

We give at first an overview of the tool. Then we describe the information which is provided to the tool via a *description file*. Finally, we explain the functioning of the tool.

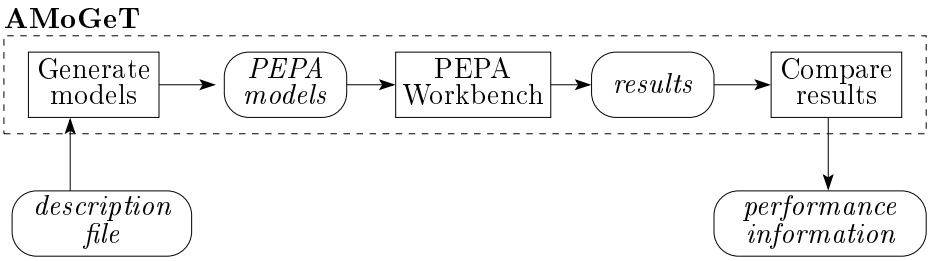


FIG. 4.1. *The principle of AMoGeT*

4.1. AMoGeT description. Fig. 4.1 illustrates the principle of the tool. In its current form, the tool is a generic, reusable software component. Its ultimate role will be as an integrated component of a run-time scheduler and re-scheduler, adapting the mapping from application to resources in response to changes in resource availability and performance.

TABLE 2.1
Correspondence between state numbers in Figure 2.2 and PEPA terms (cooperation sets are omitted but remain constant)

state no.	*PEPA state*
1	$(Network, (Stage_{10}, Stage_{20}, Stage_{30}), (Processor1, Processor2, Processor3))$
2	$(Network, (Stage_{11}, Stage_{20}, Stage_{30}), (Processor1, Processor2, Processor3))$
3	$(Network, (Stage_{12}, Stage_{20}, Stage_{30}), (Processor1, Processor2, Processor3))$
4	$(Network, (Stage_{10}, Stage_{21}, Stage_{30}), (Processor1, Processor2, Processor3))$
5	$(Network, (Stage_{11}, Stage_{21}, Stage_{30}), (Processor1, Processor2, Processor3))$
6	$(Network, (Stage_{12}, Stage_{21}, Stage_{30}), (Processor1, Processor2, Processor3))$
7	$(Network, (Stage_{10}, Stage_{22}, Stage_{30}), (Processor1, Processor2, Processor3))$
8	$(Network, (Stage_{11}, Stage_{22}, Stage_{30}), (Processor1, Processor2, Processor3))$
9	$(Network, (Stage_{12}, Stage_{22}, Stage_{30}), (Processor1, Processor2, Processor3))$
10	$(Network, (Stage_{10}, Stage_{20}, Stage_{31}), (Processor1, Processor2, Processor3))$
11	$(Network, (Stage_{11}, Stage_{20}, Stage_{31}), (Processor1, Processor2, Processor3))$
12	$(Network, (Stage_{12}, Stage_{20}, Stage_{31}), (Processor1, Processor2, Processor3))$
13	$(Network, (Stage_{10}, Stage_{21}, Stage_{31}), (Processor1, Processor2, Processor3))$
14	$(Network, (Stage_{11}, Stage_{21}, Stage_{31}), (Processor1, Processor2, Processor3))$
15	$(Network, (Stage_{12}, Stage_{21}, Stage_{31}), (Processor1, Processor2, Processor3))$
16	$(Network, (Stage_{10}, Stage_{22}, Stage_{31}), (Processor1, Processor2, Processor3))$
17	$(Network, (Stage_{11}, Stage_{22}, Stage_{31}), (Processor1, Processor2, Processor3))$
18	$(Network, (Stage_{12}, Stage_{22}, Stage_{31}), (Processor1, Processor2, Processor3))$
19	$(Network, (Stage_{10}, Stage_{20}, Stage_{32}), (Processor1, Processor2, Processor3))$
20	$(Network, (Stage_{11}, Stage_{20}, Stage_{32}), (Processor1, Processor2, Processor3))$
21	$(Network, (Stage_{12}, Stage_{20}, Stage_{32}), (Processor1, Processor2, Processor3))$
22	$(Network, (Stage_{10}, Stage_{21}, Stage_{32}), (Processor1, Processor2, Processor3))$
23	$(Network, (Stage_{11}, Stage_{21}, Stage_{32}), (Processor1, Processor2, Processor3))$
24	$(Network, (Stage_{12}, Stage_{21}, Stage_{32}), (Processor1, Processor2, Processor3))$
25	$(Network, (Stage_{10}, Stage_{22}, Stage_{32}), (Processor1, Processor2, Processor3))$
26	$(Network, (Stage_{11}, Stage_{22}, Stage_{32}), (Processor1, Processor2, Processor3))$
27	$(Network, (Stage_{12}, Stage_{22}, Stage_{32}), (Processor1, Processor2, Processor3))$

Information is provided to the tool via a *description file* (cf. Section 4.2). This information can be gathered from the Grid resources and from the application definition. In the following experiments, it is provided by the user, but we can also get it automatically from grid services, for example from the Network Weather Service [17].

The tool allows everything to be done in a single step through a simple Perl script (cf. Section 4.3): it generates the models, solves them with the PEPA Workbench, and then compares the results. This allows us to have feedback on the application when the performance of the available resources is modified.

4.2. Description file for AMoGeT. The aim of this file is to provide information about the available grid resources and the modelled application, in our case the pipeline.

This description file is named `mymodel.des`, where *mymodel* is the name of the application.

- The first information provided is the type of the model. Since we study here the pipeline skeleton, the first line is

  `type` = `pipeline`;

- We then have the information about the Grid resources and Network links, as a list of parameters. The number of processors N must at first be specified:

  `nbproc` =N;

  And then, for $i = 1..N$ and $j = 1..N$, we specify the available computing power of the processor i (`cp`i), and the performance of the network link between processors i and j (`nl`i-j):

  ```
  cp1=10; cp2=5;
  nl1-1=10000; nl1-2=8;
  ```

  `cp`i captures the fact that a processor's full power may not be available to our application (e. g. because of time-sharing with other activities).

- Concerning the application, we have some information about the stages of the pipeline. N_s is the number of stages.
  `nbstage=`N_s`;`
  The amount of work `w`i required to compute one output for stage i must be specified for $i = 1..N_s$:
  `w1=2; w2=4; ...`
  Finally, we need to specify the size of the data transferred to and from each stage. For $i = 1..N_s + 1$, `ds`i is the size of the data transferred to stage i, with the boundary case `ds`$N_s + 1$ which represents the size of the output data.
  `ds1=100; ds2=5; ...`
- Next we define a set of candidate mappings of stages to processors. Each mapping specifies where the initial data is located, where the output data must be left and (as a tuple) the processor where each stage is processed. For example, the tuple $(1, 1, 2)$ means that the two first stages are on processor 1, with the third stage on processor 2. A mapping is then of the form $[input, tuple, output]$. The mapping definition is a set of mappings, it can be as follows:
  `mappings=[1,(1,2,3),3],[1,(1,1,2),2],[1,(1,1,1),1];`
- The last thing is the performance result we want to compute. For the pipeline application, we can ask for the *throughput* with the line:
  `throughput;`

4.3. The AMoGeT Perl script. The tool allows everything to be done in a single step through a simple Perl script. The model generation is done by calling an auxiliary function. Models are then solved with the PEPA Workbench as seen in Section 3. Finally, the results are compared. This allows us to have feedback on the application when the performance of the available resources is modified.

One model is generated from each mapping of the description file. Each model is as described in Section 2.3. The difficult point consists of generating the rates from the information gathered before. The model generation itself is then straightforward.

To compute the rates of the $process_i$ activities for a given model ($i = 1..N_s$), we need to know how many stages are hosted on each processor, and we assume that the work sharing between the stages is equitable. The rate associated with the $process_i$ activity is then:

$$\mu_i = \mathtt{w}i \times \frac{\mathtt{cp}j}{\mathtt{nbst}j}$$

where j is the number of the processor hosting the stage i, and `nbst`j is the number of stages being processed on processor j. In effect, the available computing power `cp`j is further diluted by our own internal timesharing factor `nbst`j, before being applied to the workload associated with the stage, `w`i.

The rates of communication between stages depend on the mapping too, since the rate of a $move_i$ activity depends on the connection link between the processor j_1 hosting stage $i-1$ and the processor j_2 hosting stage i, which is given by `nl`j_1`-`j_2. Since the mapping specifies where the input and output data are, we can also find the connection link for the data arriving into the pipeline and the data exiting the application. These rates depend also on the size of the data transferred from one stage of the pipeline to the next, given by `ds`i. The boundary cases are applied to compute the rates of the $move_1$ and $move_{N_s+1}$ activities. The rate associated with the $move_i$ activity is therefore:

$$\lambda_i = \frac{\mathtt{nl}j_1\mathtt{-j}_2}{\mathtt{ds}i}$$

Once these rates are derived, generating the model is straightforward. We add into the file the description of the throughput of the $process_1$ activity as a required result to allow an automatic computation of this result. The models can then be solved with the PEPA Workbench, and the throughput of the pipeline is automatically computed (Section 3). During the resolution, all the results are saved in a single file, and the last step of results comparison finds out which mapping produces the best throughput. This mapping is the one we should use to run the application.

5. Numerical results. We present in this section some numerical results. We explain through them how the information obtained with AMoGeT can be relevant for optimizing the application.

In the present paper we do not apply this method to a given "real-world" example. We use an abstract pipeline for which we arbitrarily fix the time required to complete each stage. This is sufficient to show that AMoGeT can help to optimize an application.

5.1. Experiment 1: Pipeline with 3 stages—fixed data size. We give here a few numerical results on an example with 3 pipeline stages (and up to 3 processors). The models that we need to solve are really small (in this case, the model has 27 states and 51 transitions, cf. Figure 2.2).

We suppose in this experiment that `nl`i`-`i`=10000` for $i = 1..3$, and that there is no need to transfer the input or the output data. Moreover, we suppose that the network is symmetrical (`nl`i`-`j`=nl`j`-`i for all $i, j = 1..3$). Concerning the pipeline parameters, the amount of work `w`i required to compute each stage is 1, as well as the size of the data `ds`i which is transferred from one stage to another. The relevant parameters are therefore `nl1-2`, `nl2-3`, `nl1-3`, and `cp`i for $i = 1..3$. We compare different mappings, and just specify the tuple indicating which stage is on which processor. We compare the mappings (1,1,1), (1,1,2), (1,1,3), (1,2,1), (1,2,2), (1,2,3), (1,3,1), (1,3,2) and (1,3,3) (the first stage is always on processor 1). The results are displayed in Table 5.1, and we only put the best of the mappings which were investigated in the relevant line of the table.

TABLE 5.1
Result table for Experiment 1

Set of results	Parameters						Mapping & Throughput
	`nl1-2`	`nl2-3`	`nl1-3`	`cp1`	`cp2`	`cp3`	
1	10000	10000	10000	10	10	10	(1,2,3): 5.63467
	10000	10000	10000	5	5	5	(1,2,3): 2.81892
2	10000	10000	10000	10	10	1	(1,2,1): 3.36671
	10	10	10	10	10	1	(1,1,2): 2.59914
	1	1	1	10	10	1	(1,1,1): 1.87963
3	10	1	1	10	10	10	(1,1,2): 2.59914
	10	1	1	1	1	100	(1,3,3): 0.49988

In the first set of results, all the processors are identical and the network links are really fast. In these cases, the best mapping always consists of putting one stage on each processor (the results for the mapping $(1, 3, 2)$ are identical to the best mapping). If we divide the time allocated by the processor to the application by 2, the resulting throughput is also divided by 2, since only the processing power has an impact on the throughput.

The second set of results illustrates the case when one processor is becoming really busy, in this case processor 3. We should not use it any more, but depending on the network links, the best mapping may change. If the links are not efficient, we should indeed avoid data transfer and try to put consecutive stages on the same processor. When `nl1-2` = `nl2-3` = `nl1-3` = 10, the mapping $(1, 2, 2)$ provides the same results as $(1, 1, 2)$.

Finally, the third set of results shows what happens if the network link to processor 3 is really slow. In this case again, the use of the processor should be avoided, and the best mappings are $(1, 1, 2)$ and $(1, 2, 2)$. However, if processor 3 is a really fast processor compared to the other ones (last line), we process stage 2 and stage 3 on the third processor (mapping $(1, 3, 3)$).

5.2. Experiment 2: Pipeline with 3 stages—data size changing. The third experiment keeps the 3 stage pipeline, but considers changes in the size of the data. The assumptions are the same as for Experiment 1, but more parameters have a fixed value.

In this experiment, the network connection between processors 1 and 2 is slightly less effective than the others. So, we have `nl1-2` = 100, `nl2-3` = `nl1-3` = 1000. Moreover, the computing power of each stage is `cp`i = 10. The size of the data is now fixed to 100, except from the data transiting from stage 1 to stage 2 (`ds2`), whose size is varying.

Figure 5.1 presents the throughput obtained with each mapping, as a function of the data size `ds2`.

Notice first that some of the mappings are not influenced by the change of the data size, i. e. (1,1,1), (1,1,2) and (1,1,3). This is due to the fact that the connection between stages 1 and 2 is good because the data stays on the same processor. The influence of the size of the data transferred is much more important when the connection is less effective (mappings (1,2,2) and (1,2,3)), since the $move_2$ activity is then the bottleneck of the system.

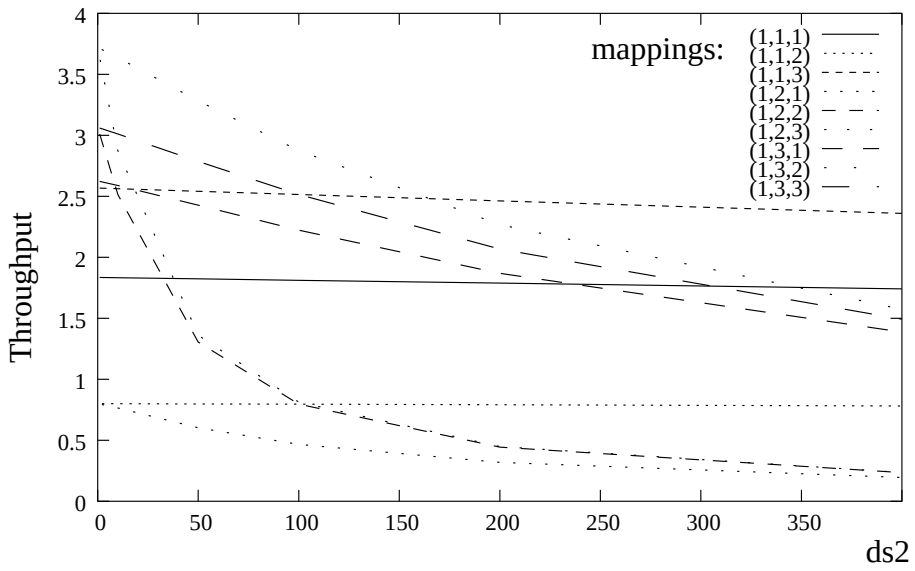


FIG. 5.1. *Experiment 2: Throughput function of* `ds2`

The best mapping is (1,3,2) when `ds2` < 150, and (1,1,3) for greater values. Both of them avoid the slow connection `nl1-2`, and they use several processors so the processing power is better than for mappings like (1,1,1). When the size of the data transferred between the first two stages becomes high, the bottleneck is the connection link between them, so it is better to put them on the same processor, even if we may lose some processing power.

5.3. Experiment 3: Pipeline with 8 stages. The last experiment considers a larger pipeline, composed of 8 stages. We use up to 8 processors, and compare four different mappings, depending on the number of processors we wish to use:

- **8 processors**, the mapping is $[1, (1, 2, 3, 4, 5, 6, 7, 8), 8]$
- **4 processors**, the mapping is $[1, (1, 1, 2, 2, 3, 3, 4, 4), 4]$
- **2 processors**, the mapping is $[1, (1, 1, 1, 1, 2, 2, 2, 2), 2]$
- **1 processor**, the mapping is $[1, (1, 1, 1, 1, 1, 1, 1, 1), 1]$

The parameters are the same as for Experiment 1, with `cp`$i = 10$, `w`$i = 1$, `ds`i`=1` and `nl`i`-`$i = 10000$ for all i. We vary the parameters `nl`i`-`j, for $i \neq j$, assuming that all these links are equal, and we compute the throughput for the different mappings. Figure 5.2 displays the results.

The curves obtained confirm that we should avoid data transfer when the network connections are less efficient. When `nl`i`-`$j > 7$, the network performs well enough to allow the use of the 8 processors. However, when the performance decreases, we should use only 4 processors, then two, and only one when `nl`i`-`$j < 0.8$.

When we need to transfer the output data back to the first processor (for example, the mapping

$$[1, (1, 2, 3, 4, 5, 6, 7, 8), 1]$$

for the case with 8 processors), we obtain almost the same results, with a slightly smaller throughput due to this additional transfer.

6. Feasibility of the approach. We envisage the use of our approach within a scheduling and rescheduling platform for long-running grid applications. In this context it is anticipated that after initial analysis and scheduling, the system would be monitored and that rescheduling would be needed only relatively infrequently, for example, once an hour. Nevertheless it is important that the use of the tool does not contribute an overhead which eliminates the benefit to be obtained from its use. In this section we present evidence which suggests that this is not likely to be the case in practice. The reader should note that here we are reflecting on the performance of the analysis tools themselves rather than on the performance of the application which they monitor (as presented in the previous section).

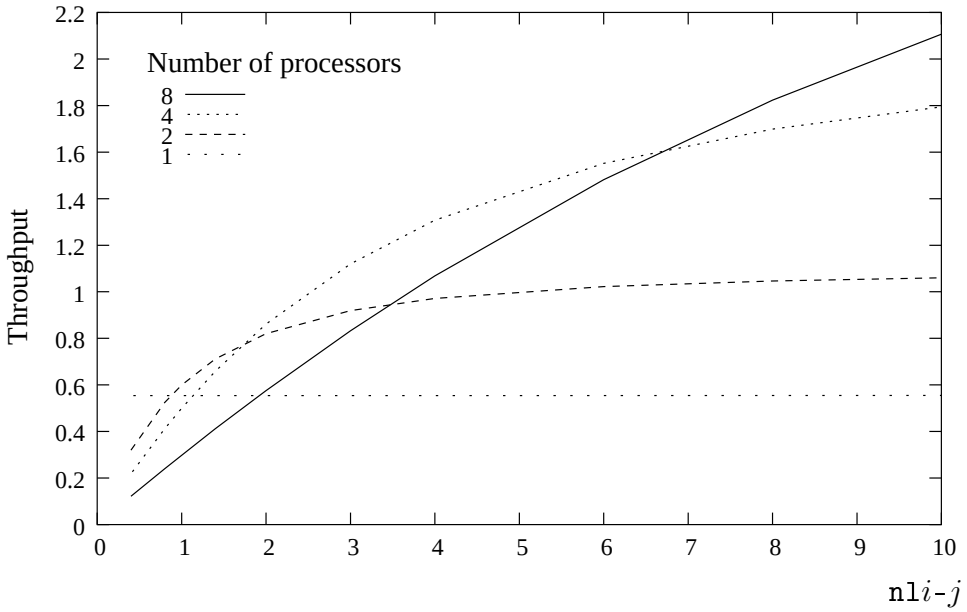


FIG. 5.2. *Experiment 3: Pipeline with* 8 *stages*

We ran an experiment to assess the time taken to generate and solve models using AMoGeT, which will, of course, be dependent on the size of the generated model. Fig. 6.1 illustrates the number of states and transitions of the models as a function of the parameters of the skeleton. These numbers are independent of the number of processors in the model; they depend only on the number of pipeline stages.

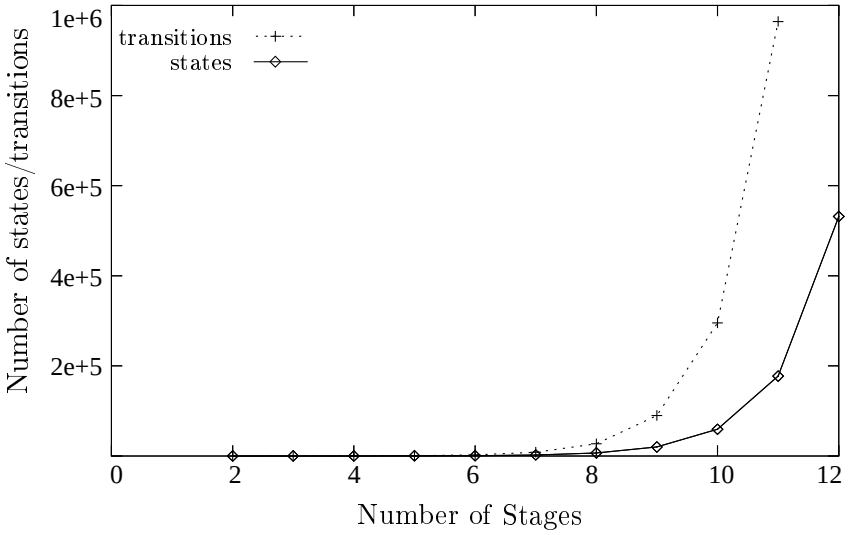


FIG. 6.1. *States and Transitions*

The time required to generate and solve the models must be carefully considered. The generation is always very quick: it takes less than 0.01 seconds to generate 20 models. The time required to solve the models is usually more important, especially when the models have a large state space. However, if we consider only relatively small models (up to 20,000 states), the resolution with the PEPA workbench takes only a few seconds. Fig. 6.1 shows that when the number of stages is less than 9, the size of the model is small enough to have a fast resolution. However, the model grows exponentially when the number of stages is increased, making AMoGeT less effective for a large number of stages. Since real applications usually do not have very many stages, this is not a limitation of the tool in practice.

The overall use of AMoGeT takes usually less than one minute for complex applications running on several processors, even when we consider several models to solve.

As stated earlier, in a scenario of long computing grid applications, with eventually dynamic rescheduling of the application, we consider that the tool may be run once per hour. We therefore believe that the amount of time required may be quite negligible and that the gain obtained by using the best of the mappings which were investigated can outperform the cost of the use of the tool.

7. Conclusions. In the context of grid applications, the availability and performance of the resources change dynamically. We have shown through this study that the use of skeletons, and performance models of these, can produce some relevant information to improve the performance of the application. This has been illustrated on the pipeline skeleton, which is a commonly used algorithmic skeleton. The models help us to choose the mapping, of the stages onto the processors, which will produce the best throughput. A tool automates all the steps to obtain the result easily.

The pipeline skeleton is a simple control skeleton. The deal skeleton has already been modelled in a similar way [3], and experiments are ongoing using deal skeletons nested into a pipeline application. This approach will also be developed on some other skeletons so it may be useful for a larger class of applications.

Our recent work considers the generation of models which take into account information from the Grid resources, which is gathered with the help of the Network Weather Service [17]. This will allow us to have models fitted to the real-time conditions of the resources. This first case study has already shown that we can use such information productively and that we have the potential to enhance the performance of grid applications with the use of skeletons and process algebras.

Having process algebra models of our skeletons also potentially offers other benefits such as the ability to formally verify the correct functioning of the skeleton. We intend to explore this aspect in future work.

Appendix A. Structured Operational Semantics for PEPA.

The semantic rules, in the structured operational style, are presented in Figure A.1; the interested reader is referred to [13] for more details. The rules are read as follows: if the transition(s) above the inference line can be inferred, then we can infer the transition below the line. The notation $r_\alpha(E)$ which is used in the third cooperation rule denotes the *apparent rate* of α in E, i.e. the sum of the rates of all activities of type α in $\mathcal{A}ct(E)$.

Appendix B. Pipeline example: input file for the PEPA Workbench.

The input file for the PEPA Workbench is displayed in Fig. B.1, for a small example with $N_s = N_p = 3$, and where each processor is hosting one of the stages.

Appendix C. The PEPA Workbench.

C.1. Functioning of the Workbench. The PEPA Workbench begins by generating the reachable state space of a PEPA model as found from all possible interleavings of its transitions from state to state. For a finite state model with n states we can enumerate this state space as $C = \{C_1, \ldots, C_n\}$. As the workbench carries out this task it compiles the *infinitesimal generator matrix* Q of the continuous-time Markov process underlying the PEPA model. The workbench adds a transition rate r to Q_{ij} every time that it finds a transition from state C_i to C_j at rate r. Additionally it subtracts r from Q_{ii} in order that the row sum of the matrix remains in balance.

The conditions which must be satisfied in order to guarantee the existence of an equilibrium distribution for a Markov process, and for this to be the same as the limiting distribution, are well-known—*a* stationary *or* equilibrium *probability distribution,* $\mathbf{\Pi}$*, exists for every time-homogeneous irreducible Markov chain whose states are all positive-recurrent.*

The intuition behind this distribution is the obvious one, namely that in the long run the probability that the PEPA model is in state C_i is given by $\mathbf{\Pi}(C_i)$.

For finite state PEPA models whose derivation graph is strongly connected, and which therefore have generated an ergodic Markov process, the equilibrium distribution of the model, $\mathbf{\Pi}$, is found by solving the matrix equation

$$\mathbf{\Pi} Q = \mathbf{0} \tag{C.1}$$

Prefix

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Cooperation

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F} \ (\alpha \notin L) \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'} \ (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} \ (\alpha \in L) \qquad \text{where } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

Choice

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

Hiding

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\alpha,r)} E'/L} \ (\alpha \notin L) \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\tau,r)} E'/L} \ (\alpha \in L)$$

Constant

$$\frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'} \ (A \stackrel{def}{=} E)$$

FIG. A.1. *The operational semantics of PEPA*

subject to the normalisation condition which ensures that $\mathbf{\Pi}$ is a well-formed probability distribution

$$\sum \mathbf{\Pi}(C_i) = 1. \tag{C.2}$$

The equations C.1 and C.2 are combined by replacing a column of $\mathbf{Q}$ by a column of ones and placing a 1 in the corresponding row of $\mathbf{0}$.

Because the connectivity graph of the state transition system of the model will in general have low degree, the transition matrix of the Markov process is best stored as a sparse matrix. The PEPA Workbench uses a Java implementation of the preconditioned biconjugate gradient method. This is an iterative procedure as described in [15] storing the infinitesimal generator matrix in *row-indexed sparse storage mode*, a compact storage mode which requires storage of only about two times the number of nonzero matrix elements. An advantage of conjugate gradient methods for large sparse systems is that they reference the matrix only through its multiplication of a vector, or the multiplication of its transpose and a vector.

C.2. Computing performance results with the PEPA Workbench. The new functionality of the workbench is described through a tiny example [10], which we shall first describe. We then explain how to add the description of the results in the PEPA input file and how to compute them.

A tiny example. We describe the components of the PEPA input language for the Workbench via a simple example, described in the file `tiny.pepa`:

```
r1=2; r2=10; r3=1;
P1=(start,r1).P2;
```

```
// PIPELINE APPLICATION
// 3 stages, 3 processors (1 stage per processor)

// Variables declaration (all identical)
mu1=10; mu2=10; mu3=10;
la1=10; la2=10; la3=10; la4=10;

// Definition of the Stages
Stage1 = (move1, infty).(process1, infty).(move2, infty).Stage1;
Stage2 = (move2, infty).(process2, infty).(move3, infty).Stage2;
Stage3 = (move3, infty).(process3, infty).(move4, infty).Stage3;

// Definition of the Processors
Processor1 = (process1, mu1).Processor1;
Processor2 = (process2, mu2).Processor2;
Processor3 = (process3, mu3).Processor3;

// Definition of the Network
Network = (move1,la1).Network + (move2,la2).Network
    + (move3,la3).Network + (move4,la4).Network;

// The pipeline model
Network <move1,move2,move3,move4>
    (Stage1 <move2> Stage2 <move3> Stage3)
    <process1,process2,process3> (Processor1||Processor2||Processor3)
```

FIG. B.1. *The input file for the PEPA Workbench:* `pipeline.pepa`

```
P2=(run,r2).P3;
P3=(stop,r3).P1;
P1 || P1
```

This model is composed of two copies of a component, P_1, executing in a pure parallel synchronization. P_1 is a simple sequential process which undergoes a *start* activity with rate r_1 to become P_2 which runs with rate r_2 to become P_3 which goes back to P_1 via a *stop* activity with rate r_3.

The first line of the file is defining the rates. Then the sequential process is defined, and the final line is the *system equation*, which describes the behaviour of the modelled system.

Adding results to the input file. In order to automatically compute some performance results, the user just needs to specify them in the PEPA input file, for example in the file `tiny.pepa` presented before. This is done by including at the end of the file one line per result, of the form:

```
result_name = {result_description};
result_name = rate * {result_description};
```

The name of the performance result that is described is `result_name`, and the description of the result for the PEPA State Finder is `result_description`.

The states of interest are described through the use of a simple pattern language, with double stars (`**`) for wild cards, and double vertical bars (`||`) for separators between model components. The model components are described in the order used in the system equation.

A `rate` can be added; in this case the final result obtained by the PEPA State Finder will be multiplied by this rate. This is quite useful to compute throughput.

For our example, we can add some results concerning the first process, independently of the state of the second one:

```
start1 = {P1 || **};
```

```
run1 = {P2 || **};
Trun1 = r2 * {P2 || **};
stop1 = {P3 || **};
```

For example, the performance result `run1` matches all the states in which the first process is ready to perform the *run* activity. The state of the second process can be anything. `Trun1` is the same, multiplied by the rate of the *run* activity `r2`. It corresponds therefore to the throughput of *run* for the first process.

For the pipeline application, the required performance result is specified in the PEPA input file `pipeline.pepa` (Fig. B.1). This is done by adding the following line at the end of this file:

```
Throughput = mu1 * { ** <move1,move2,move3,move4>
((process1, infty).(move2,infty).Stage1 <move2> ** <move3> **)
<process1,process2,process3> (** || ** || **)}
```

Computing the results. The results can be computed by using the command line interface. This is done by invoking the following command:

```
java pepa.workbench.Main -run lr ./tiny.pepa
```

The `-run lr` (or `-run lnbcg+results`) option means that we use the linear biconjugate gradient method to compute the steady state solution of the model described in the file `./tiny.pepa`, and then we compute the performance results specified in this file.

This execution prints the results to the screen, and it also saves one file per performance result (`./results/model_name.result_name`). This file is the output of the PEPA State Finder for the result description specified in the input file. It contains the state matching the description, and the sum of the steady-state probabilities for these states. It does not take the multiplicative rate into account. The results are also appended to the file `./model_root.res`, where `model_root` is the beginning of the `model_name`, until a "–" or a "." is found. This is used to automatically compare results of similar models.

Only a few files have been modified to include the new functionality in the Java Workbench. The interested reader should refer to [12].

REFERENCES

[1] 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.

[2] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, *Evaluating the performance of skeleton-based high level parallel programs*, in The International Conference on Computational Science (ICCS 2004), Part III, M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, eds., LNCS, Springer Verlag, 2004, pp. 299–306.

[3] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, *Scheduling skeleton-based grid applications using PEPA and NWS*, Submitted to a special issue of The Computer Journal on Grid Performability Modelling and Measurement, (2004).

[4] R. Biswas, M. Frumkin, W. Smith, and R. V. der Wijngaart, *Tools and Techniques for Measuring and Improving Grid Performance*, in Proc. of IWDC 2002 on Distributed Computing: Mobile and Wireless Computing, vol. 2571 of LNCS, Calcutta, India, Dec. 2002, Springer-Verlag, pp. 45–54.

[5] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press & Pitman, 1989. http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz.

[6] M. Cole, *eSkel: The* **e***dinburgh* **Ske***leton* **l***ibrary. Tutorial Introduction*, Internal Paper, School of Informatics, University of Edinburgh, (2002).
http://homepages.inf.ed.ac.uk/mic/eSkel/.

[7] M. Cole, *Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming*, Parallel Computing, 30 (2004), pp. 389–406.

[8] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1998.

[9] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington, *ICENI: Optimisation of Component Applications within a Grid Environment*, Parallel Computing, 28 (2002), pp. 1753–1772.

[10] S. Gilmore, *The PEPA Workbench: User's Manual*, Internal Paper, School of Informatics, University of Edinburgh, (2001). http://www.dcs.ed.ac.uk/pepa/pwb.pdf.

[11] S. Gilmore and J. Hillston, *The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling*, in Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, no. 794 in LNCS, Vienna, May 1994, Springer-Verlag, pp. 353–368.
http://www.dcs.ed.ac.uk/pepa/workbench.ps.gz.

[12] N. Haenel, *User Guide for the Java Edition of the PEPA Workbench - Tabasco release*, Internal Paper, School of Informatics, University of Edinburgh, (2003).
http://www.dcs.ed.ac.uk/pepa/.

[13] J. Hillston, *A Compositional Approach to Performance Modelling*, Cambridge University Press, 1996.
[14] N. Karonis, B. Toonen, and I. Foster, *MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface*, Journal of Parallel and Distributed Computing (JPDC), 63 (2003), pp. 551–563.
[15] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.
[16] F. Rabhi and S. Gorlatch, *Patterns and Skeletons for Parallel and Distributed Computing*, Springer Verlag, 2002.
[17] R. Wolski, N. Spring, and J. Hayes, *The network weather service: a distributed resource performance forecasting service for metacomputing*, Future Generation Computer Systems, 15 (1999), pp. 757–768.

Edited by: Frédéric Loulergue
Received: June 3, 2004
Accepted: June 14, 2005

Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 17–30. http://www.scpe.org
ISSN 1895-1767

EXTENDING RESOURCE-BOUNDED FUNCTIONAL PROGRAMMING LANGUAGES WITH MUTABLE STATE AND CONCURRENCY

STEPHEN GILMORE, KENNETH MACKENZIE AND NICHOLAS WOLVERSON*

Abstract. Camelot is a resource-bounded functional programming language which compiles to Java byte code to run on the Java Virtual Machine. We extend Camelot to include language support for Camelot-level threads which are compiled to native Java threads. We extend the existing Camelot resource-bounded type system to provide safety guarantees about the heap usage of Camelot threads. We demonstrate the usefulness of our concurrency extensions to the language by implementing a multi-threaded graphical network chat application which could not have been expressed as naturally in the sequential, object-free sublanguage of Camelot which was previously available.

1. Introduction. Functional programming languages allow programmers to express algorithms concisely using high-level language constructs operating over structured data, secured by strong type-systems. Together these properties support the production of high-quality software for complex application problems. Functional programs in strongly-typed languages typically have relatively few programming errors when compared to similar applications implemented in languages without these beneficial features.

These desirable language properties mean that developers shed the burdens of explicit memory management, but this has the associated cost that they typically lose all control over the allocation and deallocation of memory. The Camelot language provides an intermediate way between completely automatic memory management and unassisted allocation and deallocation in that it provides type-safe storage management by re-binding of addresses. The address of a datum can be obtained in a pattern match and used in an expression (to store a different data value at that address), overwriting the currently-held value.

The Camelot compiler targets the Java Virtual Machine but the JVM does not provide an instruction to free memory, consigning this to the garbage collector, a generational collector with three generations and implementations of stop-and-copy and mark-sweep collections. Camelot allows more precise control of memory allocation, allowing in-place modification of user-defined data structures. The Camelot compiler supports various resource-aware type systems which ensure that memory re-use takes place in a safe manner and also allow static prediction of heap-space usage. Camelot uses a uniform representation for types which are generated by the compiler, allowing data types to exchange storage cells. This uniform representation is called the *diamond type* [10, 12], implemented by a `Diamond` class in the Camelot run-time. The Camelot language implements a type system which assigns types to functions which record the number of parameters which they consume, and their types; the type of the result; and the number of diamonds consumed or freed. The outcome is that the storage consumption requirements of a function are statically computed at compile-time along with the traditional Hindley-Milner type inference procedure.

The novel contribution of the present paper is to explain how such an unusually rich programming model can be extended to incorporate object-oriented and concurrent programming idioms. This contribution is not just a design: it has been realised in the latest release of the Camelot compiler.

Structure of this paper. In Section 2 we present the Camelot language in order that the reader may understand the operational context of the work. We follow this in Section 3 with a discussion of our object-oriented extensions to Camelot. This leads on to a presentation of the use of threads in Section 4 followed by an analysis of the management of threads by the run-time system in Section 5. Section 6 explains the relationship between threads in Camelot and threads as traditionally implemented in concurrent functional languages using first-class continuations. Section 7 details the implications for verification of Camelot programs. Related work is surveyed in Section 8 and conclusions follow after that.

2. The Camelot language. The core of Camelot is a standard polymorphic ML-like functional language whose syntax is based upon that of O'Caml; the main novelty lies in extensions which allow the programmer to perform in-place modifications to heap-allocated data-structures. These features are similar to those described in by Hofmann in [11], but include some extra extensions for free list management. To retain a purely functional semantics for the language in the presence of these extensions a linear type system can be employed: in the present implementation, linearity can be enforced via a compiler switch. We are in the process of enhancing

*Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Edinburgh, EH9 3JZ, Scotland

the compiler by the addition of other, less restrictive type systems which still allow safe in-place modifications: more details will be given below.

Crucial design choices for the compilation are transparency and an exact specification of the compilation process. The former ensures that the compilation does not modify the resource consumption in an unpredictable way. The latter provides a formal basis for using resource information inferred for the high-level language in proofs on the intermediate language.

In the following sections we will give a brief description of the structure of the language. We will then outline how the language is compiled, and in particular how the memory-management extensions are implemented.

2.1. The structure of Camelot. We will give some examples to indicate the basic structure of Camelot; full details can be found in [20].

Datatypes are defined in the normal way:

```
type intlist = Nil | Cons of int * intlist
type 'a polylist = NIL | CONS of 'a * 'a polylist
type ('a, 'b) pair = Pair of 'a *'b
```

Values belonging to user-defined types are created by applying constructors and are deconstructed using the `match` statement:

```
let rec length l = match l with
       Nil -> 0
     | Cons (h,t) -> 1+length t

let test () = let l = Cons(2, Cons(7,Nil))
    in length l
```

As can be seen from this example, constructor arguments are enclosed in parentheses and are separated by commas. In contrast, function definitions and applications which require multiple arguments are written in a "curried" style:

```
let add a b = a+b
let f x y z = add x (add y z)
```

Despite this notation, the present version of Camelot does *not* support higher-order functions; any application of a function must involve exactly the same number of arguments as are specified in the definition of the function.

2.2. Diamonds and Resource Control. The Camelot compiler targets the Java Virtual Machine, and values from user-defined datatypes are represented by heap-allocated objects from a certain JVM class. Details of this representation will be given in Section 2.4.

Consider the following function which uses an accumulator to reverse a list of integers (as defined by the `intlist` type above).

```
let rec rev l acc = match l with
    Nil -> acc
  | Cons (h,t) -> rev t (Cons (h,acc))
let reverse l = rev l Nil
```

This function allocates an amount of memory equal to the amount occupied by the input list. If no further reference is made to the input list then the heap space which it occupies may eventually be reclaimed by the JVM garbage collector.

In order to allow more precise control of heap usage, Camelot includes constructs allowing re-use of heap cells. There is a special type known as the *diamond type* (denoted by `<>`) whose values represent blocks of heap-allocated memory, and Camelot allows explicit manipulation of diamond objects. This is achieved by equipping constructors and match rules with special annotations referring to diamond values. Here is the `reverse` function rewritten using diamonds so that it performs in-place reversal:

```
let rec rev l acc = match l with
    Nil -> acc
  | Cons (h,t)@d -> rev t (Cons (h,acc)@d)
let reverse l = rev l Nil
```

The annotation "`@d`" on the first occurrence of `Cons` tells the compiler that the diamond value `d` is to be bound to a reference to the space used by the list cell. The annotation on the second occurrence of `Cons` specifies

that the list cell `Cons(h,acc)` should be constructed in the diamond object referred to by `d`, and no new space should be allocated on the heap.

One might not always wish to re-use a diamond value immediately. This can sometimes cause difficulty since such diamonds might then have to be returned as part of a function result so that they can be recycled by other parts of the program. For example, the alert reader may have noticed that the list reversal function above does not in fact reverse lists entirely in place. When the user calls `reverse`, the invocation of the `Nil` constructor in the call to `rev` will cause a new list cell to be allocated. Also, the `Nil` value at the end of the input list occupies a diamond, and this is simply discarded in the second line of the `rev` function (and will be subject to garbage collection if there are no other references to it).

The overall effect is that we create a new diamond before calling the `rev` function and are left with an extra diamond after the call had completed. We could recover the extra diamond by making the `reverse` function return a pair consisting of the reversed list and the spare diamond, but this is rather clumsy and programs quickly become very complex when using this kind of technique.

To avoid this kind of problem, unwanted diamonds can be stored on a *free list* for later use. This is done by using the annotation "`@_`" as in the following example which returns the sum of the entries in an integer list, destroying the list in the process:

```
let rec sum l acc = match l with
    Nil@_ -> acc
  | Cons (h,t)@_ -> sum t (acc+h)
```

The question now is how the user retrieves a diamond from the free list. In fact, this happens automatically during constructor invocation. If a program uses an undecorated constructor such as `Nil` or `Cons(4,Nil)` then if the free list is empty the JVM `new` instruction is used to allocate memory for a new diamond object on the heap; otherwise, a diamond is removed from the head of the free list and is used to construct the value. It may occasionally be useful to explicitly return a diamond to the free list and an operator `free: <> -> unit` is provided for this purpose.

There is one final notational refinement. The in-place list reversal function above is still not entirely satisfactory since the `Nil` value carries no data but is nonetheless allocated on the heap. We can overcome this by redefining the `intlist` type as

```
type intlist = !Nil | Cons of int * intlist
```

The exclamation mark directs the compiler to represent the `Nil` constructor by the JVM `null` reference. With the new definition of `intlist` the original list-reversal function performs true in-place reversal: no heap space is consumed or destroyed when the `reverse` function is applied. The `!` annotation can be used for a single zero-argument constructor in any datatype definition. In addition, if every constructor for a particular datatype is nullary then they may all be preceded by`!`, in which case they will be represented by integer values at runtime. We have deliberately chosen to expose this choice to the programmer (rather than allowing the compiler to automatically choose the most efficient representation) in keeping with our policy of not allowing the compiler to perform optimisations which have unexpected results on resource consumption.

The features described above are very powerful and can lead to many kinds of program error. For example, if one applied the `reverse` function to a sublist of some larger list then the small list would be reversed properly, but the larger list could become partially reversed. Perhaps worse, a diamond object might be used in several different data structures of different types simultaneously. Thus a list cell might also be used as a tree node, and any modification of one structure might lead to modifications of the other. The simplest way of preventing this kind of problem is to require linear usage of heap-allocated objects, which means that variables bound to such objects may be used at most once after they are bound. Details of this approach can be found in Hofmann's paper [11]. Strict linearity would require one to write the list length function as something like

```
let rec length l = match l with
      Nil -> Pair (0, Nil)
    | Cons(h,t)@d ->
      let p = length t
      in match p with
           Pair(n, t1)@d1 -> Pair(n+1, Cons(h,t1)@d)@d1
```

It is necessary to return a new copy of the list since it is illegal to refer to `l` after calling `length l`.

Our compiler has a switch to enforce linearity, but the example demonstrates that the restrictive nature

of linear typing can lead to unnecessary complications. Aspinall and Hofmann [1] give a type system which relaxes the linearity condition while still allowing safe in-place updates, and Michal Konečný generalises this still further in [15, 16]. As part of the MRG project, Konečný has implemented a typechecker for a variant of the type system of [15] adapted to Camelot.

A different approach to providing heap-usage guarantees is given by Hofmann and Jost in [13], where an algorithm is presented which can be used to statically infer heap-usage bounds for functional programs of a suitable form. In collaboration with the MRG project, Steffen Jost has implemented a variant of this inference algorithm for Camelot: the implementation is described in [14]. Both of these implementations are currently stand-alone programs, but we are in the process of integrating them with the Camelot compiler.

One of our goals in the design of Camelot was to define a language which could be used as a testbed for different heap-usage analysis methods. The inclusion of explicit diamonds fits the type systems of [1, 15, 16], and the inclusion of the free list facilitates the Hofmann-Jost inference algorithm, which requires that all memory management takes place via a free list.

2.3. Compilation of expressions. Camelot is initially compiled into the Grail intermediate language [5, 19] which is essentially a functional form of Java bytecode. This process is facilitated by an initial phase in which several transformations are applied to the abstract syntax tree.

2.3.1. Monomorphisation. Firstly, all polymorphism is removed from the program. For polymorphic types $(\alpha_n, \ldots, \alpha_1)\ t$ such as α `list` we examine the entire program to determine all instantiations of the type variables, and compile a separate datatype for each distinct instantiation. Similarly, whenever a polymorphic function is defined the program is examined to find all uses of the function and a monomorphic function of the appropriate type is generated for each distinct instantiation of types.

2.3.2. Normalisation. After monomorphisation there is a phase referred to as *normalisation* which transforms the Camelot program into a form which closely resembles Grail.

Firstly the compiler ensures that all variables have unique names. Any duplications are resolved by generating new names. This allows us to map Camelot variable names directly onto Grail variable names (which in turn map onto JVM local variable locations) with no danger of clashes arising.

Next, we give names to intermediate results in many contexts by replacing complex expressions with variables. For example, the expression $f(a + b + c)$ would be replaced by an expression of the form `let` $t_1 = a + b$ `in let` $t_2 = t_1 + c$ `in` $f(t_2)$. The introduction of names for intermediate results can produce a large number of Grail (and hence JVM) variables. After the source code has been compiled to Grail the number of local variables is minimised by applying a standard register allocation algorithm (see [30]).

A final transformation ensures that `let`-expressions are in a "straight-line" form. After all of these transformations have been performed expressions have been reduced to a form which we refer to as *normalised Camelot*

The structure of normalised Camelot (which is in fact in a type of A-normal form [9]) is sufficiently close to that of Grail that it is fairly straightforward to translate from the former to the latter. Another benefit of normalisation is that it is easier to write and implement type systems for normalised Camelot. The fact that the components of many expressions are atoms rather than complex subexpressions means that typing rules can have very simple premisses.

2.4. Compilation of values. Camelot has various primitive types (`int`, `float`, etc.) which can be translated directly into corresponding JVM types. The compilation of user-defined datatypes, however, is rather more complicated. Objects belonging to datatypes are represented by members of a single JVM class which we will refer to as the *diamond class*. Objects of the diamond class contain enough fields to represent any member of *any* datatype defined in the program. Each instance X of the diamond class contains an integer tag field which identifies the constructor with which X is associated. The diamond class also contains a static field pointing to the free list. The free list is managed via the static methods `alloc` (which returns the diamond at the head of the free list, or creates a new diamond by calling `new` if the free list is empty), and `free` which places a diamond object on the free list. The diamond class also has overloaded static methods called `make` and `fill`, one instance of each for every sequence of types appearing in a constructor. The `make` methods are used to implement ordinary constructor application; each takes an integer tag value and a sequence of argument values and calls `alloc` to obtain an instance of the diamond class, and then calls a corresponding `fill` method

to fill in the appropriate fields with the tag and the arguments. The `fill` methods are also used when the programmer reuses an existing diamond to construct a datatype value.

It can be argued that this representation is inefficient in that datatype values are often represented by JVM objects which are larger than they need to be. This is true, but is difficult to avoid due to the type-safe nature of JVM memory management which prevents one from re-using the heap space occupied by a value of one type to store a value of a different type. We wish to be able to reuse heap space, but this can be impossible if objects can contain only one type of data. With the current scheme one can easily write a heapsort program which operates entirely in-place. List cells are large enough to be reused as heap nodes and this allows a heap to be built using cells obtained by destroying the input list. Once the heap has been built it can in turn be destroyed and the space reused to build the output list. In this case, the amount of memory occupied by a list cell is larger than it needs to be, but the overall amount of store required is less than would be the case if separate classes were used to contain list cells and heap nodes.

In the current context it can be claimed that it is better to have an inefficient representation about which we can give concrete guarantees than an efficient one which about we can say nothing. Most of the programs which we have written so far use a limited number of datatypes so that the overhead introduced by the monolithic representation for diamonds is not too severe. However, it is likely that for very large programs this overhead would become unacceptably large. One possibility which we have not yet explored is that it might be possible to achieve more efficient heap usage by using dataflow techniques to follow the flow of diamonds through the program and detect datatypes which are never used in an overlapping way. One could then equip a program with several smaller diamond classes which would represent such non-overlapping types.

These problems could be avoided by compiling to some platform other than the JVM (for example to C or to a specialised virtual machine) where compaction of heap regions would be possible. The Hofmann-Jost algorithm is still applicable in this situation, so it would still be feasible to produce resource guarantees. However, it was a fundamental decision of the MRG project to use the JVM, based on the facts that the JVM is widely deployed and very well-known, and that resource usage is a genuine concern in many contexts where the JVM is used. Our present approach allows us to produce concrete guarantees at the cost of some overhead; we hope that at a later stage a more sophisticated approach (such as the one suggested above) might allow us to reduce the overheads while still obtaining guaranteed resource bounds.

2.5. Remarks. There are various ways in which Camelot could be extended. The lack of higher-order functions is inconvenient, but the resource-aware type systems which we use are presently unable to deal with higher-order functions, partly because of the fact that these are normally implemented using heap-allocated closures whose size may be difficult to predict. A possible strategy for dealing with this which we are currently investigating is Reynolds' technique of *defunctionalization* [24] which transforms higher-order programs into first-order ones, essentially by performing a transformation of the source code which replaces closures with members of datatypes. This has the advantage that extra space required by closures is exposed at the source level, where it is amenable to analysis by the heap-usage inference techniques mentioned earlier.

3. Object-oriented extensions. The core Camelot language as described in Section 2 above enables the programmer to write a program with a predictable resource usage; however, only primitive interaction with the outside world is possible, through command line arguments, file input and printed output. To be able to write a full interface for a game or utility to be run on a mobile device, Camelot programs must be able to interface with external Java libraries. Similarly, the programmer may wish to utilise device-specific libraries, or Java's extensive class library.

This section describes our object-oriented extension to Camelot. This is primarily intended to allow Camelot programs to access Java libraries. It would also be possible to write resource-certified libraries in Camelot for consumption by standard Java programs, or indeed use the object system for OO programming for its own sake, but giving Camelot programs access to the outside world is the main objective.

In designing an object system for Camelot, many choices are made for us, or at least tightly constrained. We wish to create a system allowing inter-operation with Java, and we wish to compile an object system to JVML. So we are almost forced into drawing the object system of the JVM up to the Camelot level, and cannot seriously consider a fundamentally different system.

On the other hand, the type system is strongly influenced by the existing Camelot type system. There is more scope for choice, but implementation can become complex, and an overly complex type system is

undesirable from a programmer's point of view. We also do not want to interfere with type systems for resources as mentioned above.

We shall first attempt to make the essential features of Java objects visible in Camelot in a simple form, with the view that a simple abbreviation or module system can be added at a later date to make things more palatable if desired.

3.1. Basic Features. We shall view objects as records of possibly mutable fields together with related methods, although Camelot has no existing record system. We define the usual operations on these objects, namely object creation, method invocation, field access and update, and casting and matching. As one might expect we choose a class-based system closely modelling the Java object system. Consequently we must acknowledge Java's uses of classes for encapsulation, and associate static methods and fields with classes also.

We now consider these features. The examples below illustrate the new classes of expressions we add to Camelot.

Static method calls There is no conceptual difference between static methods and functions, ignoring the use of classes for encapsulation, so we can treat static method calls just like function calls.
`java.lang.Math.max a b`

Static field access Some libraries require the use of static fields. We should only need to provide access to constant static fields, so they correspond to simple values.
`java.math.BigInteger.ONE`

Object creation We clearly need a way to create objects, and there is no need to deviate from the `new` operator. By analogy with standard Camelot function application syntax (i.e. curried form) we have:
`new java.math.BigInteger "101010" 2`

Instance field access To retrieve the value of an instance variable, we write
`object#field`
whereas to update that value we use the syntax
`object#field <- value`
assuming that `field` is declared to be a *mutable* field.
It could be argued that allowing unfettered external access to an object's variables is against the spirit of OO, and more to the point inappropriate for our small language extension, but we wish to allow easy interoperability with any external Java code.

Method invocation Drawing inspiration from the O'Caml syntax, and again using a curried form, we have instance method invocation:
`myMap#put key value`

Null values In Java, any method with object return type may return the `null` object. For this reason we add a construct
`isnull` e
which tests if the expression e is a `null` value.

Casts and typecase It may be occasionally be necessary to cast objects up to superclasses, for example to force the intended choice between overloaded methods. We will also want to recover subclasses, such as when removing an object from a collection. Here we propose a simple notation for up-casting:
`obj :> Class`
This notation is that of O'Caml, also borrowed by MLj (described in [3]). To handle down-casting we shall extend patterns in the manner of `typecase` (again like MLj) as follows:

```
match obj with o :> C1 -> o.a()
             | o :> C2 -> o.b()
             | _ -> obj.c()
```

Here `o` is bound in the appropriate subexpressions to the object `obj` viewed as an object of type `C1` or `C2` respectively. As in datatype matches we require that every possible case is covered; here this means that the default case is mandatory. We also require that each class is a subclass of the type of `obj`, and suggest that a compiler warning should be given for any redundant matches.
Unlike MLj we choose not to allow downcasting outside of the new form of `match` statement, partly because at present Camelot has no exception support to handle invalid down-casts.

As usual, the arguments of a (static or instance) method invocation may be subclasses of the method's argument types, or classes implementing the specified interfaces.

The following example demonstrates some of the above features, and illustrates the ease of interoperability. Note that the type of the parameter l is specified by a constraint here. Type inference does not cross class boundaries in Camelot.

```
let convert (l: string list) =
  match l with [] -> new java.util.LinkedList ()
              | h::t ->
                let ll = convert t
                in let _ = ll#addFirst h
                in ll
```

3.2. Defining classes. Once we have the ability to write and compile programs using objects, we may as well start writing classes in Camelot. We must be able to create classes to implement callbacks, such as in the Swing GUI system which requires us to write stateful adaptor classes. Otherwise, as mentioned previously, we may wish to write Camelot code to be called from Java, for example to create a resource-certified library for use in a Java program, and defining a class is a natural way to do this. Implementation of these classes will obviously be tied to the JVM, but the form these take in Camelot has more scope for variation.

We allow the programmer to define a class which may explicitly subclass another class, and implement a number of interfaces. We also allow the programmer to define (possibly mutable) fields and methods, as well as static methods and fields for the purpose of creating a specific class for interfacing with Java. We naturally allow reference to `this`.

The form of a class declaration is given below. Items within angular brackets $\langle \ldots \rangle$ are optional.

$$\begin{aligned} \mathit{classdecl} &::= \texttt{class}\ \mathit{cname}\ \texttt{=}\ \langle \mathit{scname}\ \texttt{with} \rangle\ \mathit{body}\ \texttt{end} \\ \mathit{body} &::= \langle \mathit{interfaces} \rangle\ \langle \mathit{fields} \rangle\ \langle \mathit{methods} \rangle \\ \mathit{interfaces} &::= \texttt{implement}\ \mathit{iname}\ \langle \mathit{interfaces} \rangle \\ \mathit{fields} &::= \mathit{field}\ \langle \mathit{fields} \rangle \\ \mathit{methods} &::= \mathit{method}\ \langle \mathit{methods} \rangle \end{aligned}$$

This defines a class called *cname*, implementing the specified interfaces. The optional *scname* gives the name of the direct superclass; if it is not present, the superclass is taken to be the root of the class hierarchy, namely `java.lang.Object`. The class *cname* inherits the methods and values present in its superclass, and these may be referred to in its definition.

As well as a superclass, a class can declare that it implements one or more interfaces. These correspond directly to the Java notion of an interface. Java libraries often require the creation of a class implementing a particular interface—for example, to use a Swing GUI one must create classes implementing various interfaces to be used as callbacks. Note that at the current time it is not possible to define interfaces in Camelot, they are provided purely for the purpose of interoperability.

Now we describe field declarations.

$$\mathit{field} ::= \texttt{field}\ x : \tau \mid \texttt{field mutable}\ x : \tau \mid \texttt{val}\ x : \tau$$

Instance fields are defined using the keyword `field`, and can optionally be declared to be mutable. Static fields are defined using `val`, and are non-mutable. In a sense these mutable fields are the first introduction of side-effects into Camelot. While the Camelot language is defined to have an array type, this has largely been ignored in our more formal treatments as it is not fundamental to the language. Mutable fields, on the other hand, are fundamental to our notion of object orientation, so we expect any extension of Camelot resource-control features to object-oriented Camelot to have to deal with this properly.

Methods are defined as follows, where $1 \leq i_1, \ldots, i_m \leq n$.

$$\begin{aligned} \mathit{method} ::= {} & \texttt{maker}(x_1 \texttt{:} \tau_1) \ldots (x_n \texttt{:} \tau_n)\ \langle \texttt{: super}\ x_{i_1} \ldots x_{i_m} \rangle\ \texttt{=}\ \mathit{exp} \\ \mid {} & \texttt{method}\ m(x_1 \texttt{:} \tau_1) \ldots (x_n \texttt{:} \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp} \\ \mid {} & \texttt{method}\ m()\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp} \\ \mid {} & \texttt{let}\ m(x_1 \texttt{:} \tau_1) \ldots (x_n \texttt{:} \tau_n)\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp} \\ \mid {} & \texttt{let}\ m()\ \texttt{:}\ \tau\ \texttt{=}\ \mathit{exp} \end{aligned}$$

Again, we use the usual `let` syntax to declare what Java would call static methods. Static methods are simply *monomorphic* Camelot functions which happen to be defined within a class, although they are invoked using the syntax described earlier. Instance methods, on the other hand, are actually a fundamentally new addition to the language. We consider the instance methods of a class to be a set of mutually recursive monomorphic functions, in which the special variable `this` is bound to the current object of that class.

We can consider the methods as mutually recursive without using any additional syntax (such as `and` blocks) since they are monomorphic. ML uses `and` blocks to group mutually recursive functions because its *let-polymorphism* prevents any of these functions being used polymorphically in the body of the others, but this is not an issue here. In any case this implicit mutual recursion feels appropriate when we are compiling to the Java Virtual Machine, and have to come to terms with open recursion.

In addition to static and instance methods, we also allow a special kind of method called a *maker*. This is just what would be called a constructor in the Java world, but as in [8] we use the term maker in order to avoid confusion between object and datatype constructors. The `maker` term above defines a maker of the containing class C such that if `new` C is invoked with arguments of type $\tau_1 \ldots \tau_n$, an object of class C is created, the superclass maker is executed (this is the zero-argument maker of the superclass if none is explicitly specified), expression *exp* (of `unit` type) is executed, and the object is returned as the result of the `new` expression. Every class has at least one maker; a class with no explicit maker is taken to have the maker with no arguments which invokes the superclass zero-argument maker and does nothing. This implicit maker is inserted by the compiler.

3.3. Polymorphism. We remarked earlier that static methods are basically monomorphic Camelot functions together with a form of encapsulation, but it is worth considering polymorphism more explicitly. object-oriented Camelot methods, whether static or instance methods, are not polymorphic. That is, they have subtype polymorphism but not parametric polymorphism (genericity), unlike Camelot functions which have parametric but not subtype polymorphism. This is not generally a problem, as most polymorphic functions will involve manipulation of polymorphic datatypes, and can be placed in the main program, whereas most methods will be interfacing with the Java world and thus should conform to Java's subtyping polymorphism.

3.4. Translation. As mentioned earlier, the present Camelot compiler targets the JVM, via the intermediate language Grail. Translating the object-oriented features which have just been described is relatively straightforward, as the JVM (and Grail) provide what we need. A detailed formal description of the translation process can be found in [31]

3.5. Objects and Resource Types. As described earlier, the use of diamond annotations on Camelot programs in combination with certain resource-aware type systems allows the heap usage of those programs to be inferred, as well as allowing some in-place update to occur. Clearly the presence of mutable objects in object-oriented Camelot also provides for in-place update. However by allowing arbitrary object creation we also replicate the unbounded heap-usage problem solved for datatypes. Perhaps more seriously, we are allowing Camelot programs to invoke arbitrary Java code, which may use an unlimited amount of heap space.

Firstly consider the second problem. Even if we have some way to place a bound on the heap space used by our new OO features within a Camelot program, external Java code may use arbitrary amounts of heap. There seem to be a few possible approaches to this problem, none of which are particularly satisfactory. We could decide to only allow the use of external classes if they came with a proof of bounded heap usage. Constructing a resource-bounded Java class library or inferring resource bounds for an existing library would be a massive undertaking, although perhaps less problematic with the smaller class libraries used with mobile devices. This suggestion seems somewhat unrealistic.

Alternatively, we could simply allow the resource usage of external methods to be stated by the programmer or library creator. This extends the trusted computing base in the sense of resources, but seems a more reasonable solution. The other alternative—considering resource-bound proofs to only refer to the resources directly consumed by the Camelot code—seems unrealistic, as one could easily (and even accidentally) cheat by using Java libraries to do some memory-consuming "dirty work".

The issue of heap-usage *internal* to object-oriented Camelot programs seems more tractable, although we do not propose a solution here. A first attempt might mimic the techniques used earlier for datatypes; perhaps we can adapt the use of diamonds and linear type systems? The use of diamonds for in-place update is irrelevant here, and indeed relies on the uniform representation of datatypes by objects of a particular Java class. Since we are hardly going to represent every Java object by an object of one class we could not hope to have such a direct correlation between diamonds and chunks of storage.

However, we could imagine an abstract diamond which represents the heap storage used by an arbitrary object, and require any instance of `new` to supply one of these diamonds, in order that the total number of objects created is limited. Unfortunately reclamation of such an abstract diamond would only correspond to making an object available to garbage collection, rather than definitely being able to re-use the storage. Even so, such a system might be able to give a measure of the total number of objects created and the maximum number in active use simultaneously.

4. Using threads in Camelot. Previously the JVM had been used simply as a convenient run-time for the Camelot language but the object-oriented extensions described above allow the Java namespace to be accessed from a Camelot application. Thus a Camelot application can now create Java objects and invoke Java methods. Figure 4.1 shows the implementation of a remote input reader in RoundTable, a networked chat application written in Camelot. This example class streams input from a network connection and renders it in a display area in the graphical user interface of the application.

```
(* Thread to read from the network, passing data to a display object *)
class remote = java.lang.Thread
with
  field input : java.io.BufferedReader
  field disp : display
  maker (i : java.io.BufferedReader)(d : display) =
    let _ = input ← i in disp ← d
  method run() : unit =
    let line = this#input#readLine()
    in if isnullobj line then () else
    let _ = this#disp#append line
    in this#run()
end
```

FIG. 4.1. *An extract from the RoundTable chat application showing the OO extensions to Camelot*

This example shows the Camelot syntax for method invocation (`obj#meth()`), field access (`obj#field`) and mutable field update (`f <- exp`). Both of these are familiar from Objective Caml.

This example also shows that even in the object-oriented fragment of the Camelot language that the natural definition style for unbounded repetition is to write recursive method calls. The Camelot compiler converts tail-calls of instance methods (such as **this**#*run*) into while-loops so that methods implemented as in Figure 4.1 run in constant space and do not overflow the Java run-time stack. In contrast recursive method calls in Java are not optimised in this way and would lead to the program overflowing the stack.

A screenshot of a window from the RoundTable application is shown in Figure 4.2. This shows date-and-time-stamped messages arriving spontaneously in the window. The application offers the ability to thread messages by content or to sort them by time. The sorting routine is guaranteed by typechecking to run in constant space because addresses of cons cells in the list of messages are re-cycled using the free list as described in Section 2.2.

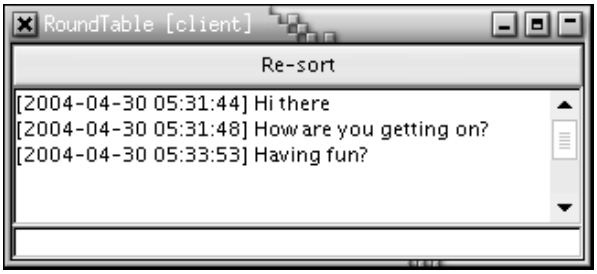


FIG. 4.2. *Screenshot of the Camelot RoundTable application*

The extension of the Camelot compiler to support interoperation with Java facilitates the implementation of graphical applications such as these. The Java APIs used by this application include the Swing graphical user interface components, networking, threads and pluggable look-and-feel components such as the Skin look-and-feel shown above.

5. Management of threads. In designing a thread management system for Camelot our strongest requirement was to have a system which works harmoniously with the storage management system already in place for Camelot. One aspect of this is that the resource consumption of a single-threaded Camelot program can be computed in line with the reasoning explained in Section 1.

In moving from one to multiple threads the most important question with respect to memory usage is the following. Should the free list of storage which can be reused be a single static instance shared across all threads; or should each thread separately maintain its own local instance of the free list?

In the former case the accessor methods for the free list must be synchronised in order for data structures not to become disordered by concurrent write operations. Synchronisation incurs an overhead of locking and unlocking the parent of the field when entering and leaving a critical region. This imposes a run-time penalty.

In the latter case there is no requirement for access to the free list to be synchronised; each thread has its own free list. In this case, though, the free memory on each free list is private, and not shared. This means that there will be times when one thread allocates memory (with a Java **new** instruction) while another thread has unused memory on its local free list. This imposes a penalty on the program memory usage, and this form of thread management would lead to programs typically using more memory overall.

We have chosen the former scheme; we have a single static instance of a free list shared across all threads. Our programs will take longer than their optimum run-time but memory performance will be improved. Crucially, predictability of memory consumption is retained.

There are several possible variants on this second scheme which we considered. They were not right for our purposes but might be right for others. One interesting alternative is a hybrid of the two approaches is where each thread had a bounded (small) local free list and flushes this to the global free list when it becomes full. This would reduce the overhead of calls to access the synchronised global free list, while preventing threads from keeping too many unused memory cells locally. This could be a suitable compromise between the two extremes but the analysis of this approach would inevitably be more complicated than the approach which we adopted (a single static free list).

A second alternative would be to implement weak local free lists. In this construction each thread would have its own private free list implemented using *weak references* which are references that are not strong enough by themselves to keep an object alive if no genuine references to it are retained. Weak references are typically used to implement caches and secondary indexes for data structures. Other high-level garbage-collected languages such as O'Caml implement weak references also. This scheme was not usable by us because the Camelot compiler also targets small JVMs on handheld devices and the J2ME does not provide the necessary class (`java.lang.ref.WeakReference`).

The analysis of memory consumption of Camelot programs is based on the consumption of memory by heap-allocated data structures. The present analysis of Camelot programs is based on a single-threaded architecture. To assist with the development of an analysis method for multi-threaded Camelot programs we require that data structures in a multi-threaded Camelot program are not shared across threads. For example, it is not possible to hold part of a list in one thread and the remainder in another. This requirement means that the space consumption of a multi-threaded Camelot program is obtained as the sum of per-thread space allocation plus the space requirements of the threads themselves.

At present our type system takes account of heap allocations but does not take account of stack growth. Thus Camelot programs can potentially (and sometimes do in practice) fail at runtime with a `java.lang.StackOverflowError` exception if the programmer overuses the idiom of working with families of mutually-recursive functions and methods which compute with deeply-nested recursion.

Even sophisticated functional language compilers for the JVM suffer from this problem and some, such as MLj [4, 3], do not even implement tail-call elimination in cases where the Camelot compiler does. Several authors consider the absence of support for tail call elimination to be a failing of the JVM [2, 22]. An approach to eliminating tail calls such as that used by Funnel [25] would be a useful next improvement to the Camelot compiler. Techniques such as *trampolining* have also been shown to work for the JVM [29]. The principal reason why the JVM does not automatically perform tail-call optimisation is that the Java security model may

require inspection of the stack to ensure that a particular method has sufficient privileges to execute another method; eliminating tail-calls would lead to the discarding of stack frames which contain the necessary security information. However, Clements and Felleisen have recently proposed another security model which allows safe tail-call optimisation [7]; they claim that this requires only a minor change to the mechanism currently used by the JVM (and other platforms), so there may be some hope that future JVM implementations will support proper tail-call optimisation and thus simplify the process of implementing functional languages for the JVM.

6. A simple thread model for Camelot. To retain predictability of memory behaviour in Camelot we restrict the programming model offered by Java's threads.

Firstly, we disallow use of the `stop` and `suspend` methods from Java's threads API. These are deprecated methods which have been shown to have poor programming properties in any case. Use of the `stop` method allows objects to be exposed in a damaged state, part-way through an update by a thread. Use of `suspend` freezes threads but these do not release the objects which they are holding locks on, thereby often leading to deadlocks. Dispensing with pre-emptive thread interruption means that there is a correspondence between Camelot threads and lightweight threads implemented using first-class continuations, `call/cc` and `throw`, as are usually to be found in multi-threaded functional programming languages [6, 18].

Secondly, we require that all threads are run, again for the purposes of supporting predictability of memory usage. In the Java language thread allocation (using **new**) is separated from thread initiation (using the `start` method in the `java.lang.Thread` class) and there is no guarantee that allocated threads will ever be run at all. In multi-threaded Camelot programs we require that all threads are started at the point where they are constructed.

Finally, we have a single constructor for classes in Camelot because our type system does not support overloading. This must be passed initial values for all the fields of the class (because the thread will initiate automatically). All Camelot threads except the main thread of control are daemon threads, which means that the Java Virtual Machine will not keep running if the main thread exits.

```
let rec threadname(args) =
    let locals = subexps in threadname(args)
let threadInstance =
    new threadname(actuals) in ...
```

$\rightsquigarrow$

```
class threadnameHolder(args) = java.lang.Thread
with
    let rec threadname() =
        let locals = subexps in threadname()
    method run() : unit =
        let _ = this#setDaemon(true)
        in threadname()
end
let threadInstance =
    new threadnameHolder(actuals) in
let _ = threadInstance#start() in ...
```

FIG. 6.1. *Derived forms for thread creation and use in Camelot*

This simplified idiom of thread use in Camelot allows us to define *derived forms* for Camelot threads which abbreviate the use of threads in the language. These derived forms can be implemented by *class hoisting*, moving a generated class definition to the top level of the program. This translation is outlined in Figure 6.1.

7. Threads and (non-)termination. The Camelot programming language is supported not only by a strong, expressive type system but also by a program logic which supports reasoning about the time and space usage of programs in the language. However, the logic is a logic of partial correctness, which is to say that the correctness of the program is guaranteed only under the assumption that the program terminates. It would

be possible to convert this logic into a logic of total correctness which would guarantee termination instead of assuming it but proofs in such a logic would be more difficult to produce than proofs in the partial correctness logic.

It might seem nonsensical to have a logic of partial correctness to guarantee execution times of programs ("this program either terminates in 20 seconds or it never does") but even these proofs about execution times have their use. They are used to provide a bound on the running time of a program so that if this time is exceeded the program may be terminated forcibly by the user or the operating system because after this point it seems that the program will not terminate. Such *a priori* information about execution times would be useful for scheduling purposes. In Grid-based computing environments Grid service providers schedule incoming jobs on the basis of estimated execution times supplied by Grid users. These estimates are sometimes significantly wrong, leading the scheduler either to forcibly terminate an over-running job due to an under-estimated execution time or to schedule other jobs poorly on the basis of an over-estimated execution time.

Because of the presence of threads in the language we now have meaningful (impure, side-effecting) functions which do not terminate so a strong functional programming approach [27] requiring proofs of termination for every function would be inappropriate for our purposes.

8. Related work. The core of the Camelot programming language is a strict, call-by-value first-order functional programming language in the ML family extended with explicit memory deallocation commands and an extended type system which expresses the cost of function application in terms of an increase in the size of the allocated memory on the heap. Other authors have addressed a similar programming model with some variations. Lee, Yang and Yi [17] present a static analysis approach which is used in applying a source-level transformation to insert explicit **free** commands into the program text. Their analysis allows uses of explicit memory deallocation which are not expressible in Camelot due to the linearity requirement of the Camelot type system. Vasconcelos and Hammond [28] present a type system which is superior to ours in applying to higher-order functional programs. Our primary cost computation is memory allocation whereas their primary focus is on run-time abstracted as the number of beta-reductions in the abstract semantic interpretation of the function term against the operational semantics of the language. Our work differs from both of these in considering multi-threaded, not only single-threaded programs.

We have made reference to MLj, the aspects of which related to Java interoperability are described in [3]. MLj is a fully formed implementation of Standard ML, and as such is a much larger language than we consider here. In particular, MLj can draw upon features from SML such as modules and functors, for example, allowing the creation of classes parameterised on types. Such flexibility comes with a price, and we hope that the restrictions of our system will make the certification of the resource usage of object-oriented Camelot programs more feasible.

By virtue of compiling an ML-like language to the JVM, we have made many of the same choices that have been made with MLj. In many cases there is one obvious translation from high level concept to implementation, and in others the appropriate language construct is suggested by the Java object system. However we have also made different choices more appropriate to our purpose, in terms of transparency of resource usage and the desire for a smaller language. For example, we represent objects as records of mutable fields whereas MLj uses immutable fields holding references.

There have been various other attempts to add object oriented features to ML and ML-like languages. O'Caml provides a clean, flexible object system with many features and impressive type inference—a formalised subset is described in [23]. As in object-oriented Camelot, objects are modelled as records of mutable fields plus a collection of methods. Many of the additional features of O'Caml could be added to object-oriented Camelot if desired, but there are some complications caused when we consider Java compatibility. For example, there are various ways to compile parameterised classes and polymorphic methods for the JVM, but making these features interact cleanly with the Java world is more subtle.

The power of the O'Caml object system seems to come more from the distinctive type system employed. O'Caml uses the notion of a *row variable*, a type variable standing for the types of a number of methods. This makes it possible to express "a class with these methods, and possibly more" as a type. Where we would have a method parameter taking a particular object type and by subsumption any subtype, in O'Caml the type of that parameter would include a row variable, so that any object with the appropriate methods and fields could be used. This allows O'Caml to preserve type inference, but this is less important for our application, and does not map cleanly to the JVM.

A class mechanism for Moby is defined in [8] with the principle that classes and modules should be orthogonal concepts. Lacking a module system, Camelot is unable to take such an approach, but both Moby and O'Caml have been a guide to concrete representation. Many other relevant issues are discussed in [21], but again lack of a module system—and our desire to avoid this to keep the language small—gives us a different perspective on the issues.

9. Conclusions and further work. Our ongoing programme of research on the Camelot functional programming language has been investigating resource consumption and providing static guarantees of resource consumption at the time of program compilation. Our thread management system provides a layer of abstraction over Java threads. This could allow us to modify the present implementation to multi-task several Camelot threads onto a single Java thread. The reason to do this would be to circumvent the ungenerous thread limit on some JVMs. This extension remains as future work but our present design strongly supports such an extension.

We have discussed a very simple thread package for Camelot. A more sophisticated one, perhaps based on Thimble [26], would provide a much more powerful programming model.

A possibly profitable extension of Camelot would be to use defunctionalization [24] to eliminate mutual tail-recursion. Given a set of mutually recursive functions $\mathcal{F}$ whose results are of type `t`, we define a datatype `s` which has for each of the functions in $\mathcal{F}$ a constructor with arguments corresponding to the function's arguments. The collection of functions $\mathcal{F}$ is then replaced by a single function `f: s -> t` whose body is a `match` statement which carries out the computations required by the individual functions in $\mathcal{F}$. In this way the mutually recursive functions can be replaced by a single tail-recursive function, and we already have an optimisation which eliminates recursion for such functions. This technique is somewhat clumsy, and care is required in recycling the diamonds which are required to contain members of the datatypes required by `s`. Another potential problem is that several small functions are effectively combined into one large one, and there is thus a danger that that 64k limit for JVM methods might be exceeded. Nevertheless, this technique does overcome the problems related to mutual recursion without affecting the transparency of the compilation process unduly, and it might be possible for the compiler to perform the appropriate transformations automatically. We intend to investigate this in more detail.

Acknowledgements. The authors are supported by the Mobile Resource Guarantees project (MRG, project IST-2001-33149). The MRG project is funded under the Global Computing pro-active initiative of the Future and Emerging Technologies part of the Information Society Technologies programme of the European Commission's Fifth Framework Programme. The other members of the MRG project provided helpful comments on an earlier presentation of this work. Java is a trademark of SUN Microsystems.

REFERENCES

[1] D. Aspinall and M. Hofmann, *Another type system for in-place update*, in Proc. 11th European Symposium on Programming, Grenoble, vol. 2305 of Lecture Notes in Computer Science, Springer, 2002.

[2] N. Benton, *Some shortcomings of, and possible improvements to, the Java Virtual Machine.* This is an unpublished note which is available on-line at `http://research.microsoft.com/~nick/jvmcritique.pdf`, June 1999.

[3] N. Benton and A. Kennedy, *Interlanguage working without tears: Blending SML with Java*, in Proceedings of the 4th ACM SIGPLAN Conference on Functional Programming, Paris, Sept. 1999, ACM Press.

[4] N. Benton, A. Kennedy, and G. Russell, *Compiling Standard ML to Java bytecodes*, in Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming, Baltimore, sep 1998, ACM Press.

[5] L. Beringer, K. MacKenzie, and I. Stark, *Grail: a functional form for imperative mobile code*, in Electronic Notes in Theoretical Computer Science, V. Sassone, ed., vol. 85, Elsevier, 2003.

[6] E. Biagioni, K. Cline, P. Lee, C. Okasaki, and C. Stone, *Safe-for-space threads in Standard ML*, Higher-Order and Symbolic Computation, 11 (1998), pp. 209–225.

[7] J. Clements and M. Felleisen, *A tail-recursive machine with stack inspection*, ACM Transactions on Programming Languages and Systems. To appear.

[8] K. Fisher and J. Reppy, *Moby objects and classes*, 1998. Unpublished manuscript.

[9] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, *The essence of compiling with continuations*, in Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993, vol. 28(6), ACM Press, New York, 1993, pp. 237–247.

[10] M. Hofmann, *A type system for bounded space and functional in-place update*, Nordic Journal of Computing, 7 (2000), pp. 258–289.

[11] ———, *A type system for bounded space and functional in-place update*, Nordic Journal of Computing, 7 (2000), pp. 258–289.

[12] M. Hofmann and S. Jost, *Static prediction of heap space usage for first-order functional programs*, in Proc. 30th ACM Symp. on Principles of Programming Languages, 2003.

[13] ———, *Static prediction of heap space usage for first-order functional programs*, in Proc. 30th ACM Symp. on Principles of Programming Languages, New Orleans, 2003.
[14] S. Jost, `lfd_infer`*: an implementation of a static inference on heap-space usage.*, in Proceedings of SPACE'04, Venice, 2004. To appear.
[15] M. Konečný, *Functional in-place update with layered datatype sharing*, in TLCA 2003, Valencia, Spain, Proceedings, Springer-Verlag, 2003, pp. 195–210. Lecture Notes in Computer Science 2701.
[16] ———, *Typing with conditions and guarantees for functional in-place update*, in TYPES 2002 Workshop, Nijmegen, Proceedings, Springer-Verlag, 2003, pp. 182–199. Lecture Notes in Computer Science 2646.
[17] O. Lee, H. Yang, and K. Yi, *Inserting safe memory reuse commands into ML-like programs*, in Proceedings of the 10th Annual International Static Analysis Symposium, vol. 2694 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 171–188.
[18] P. Lee, *Implementing threads in Standard ML*, in Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text, J. Launchbury, E. Meijer, and T. Sheard, eds., vol. 1129 of Lecture Notes in Computer Science, Springer, 1996, pp. 115–130.
[19] K. MacKenzie, *Grail: a functional intermediate language for resource-bounded computation.* LFCS, University of Edinburgh, 2002. Available at `http://groups.inf.ed.ac.uk/mrg/publications/`.
[20] K. MacKenzie and N. Wolverson, *Camelot and Grail: Resource-aware functional programming for the JVM*, in Trends in Functional Programming, Intellect, 2004, pp. 29–46.
[21] D. MacQueen, *Should ML be object-oriented?*, Formal Aspects of Computing, 13 (2002).
[22] E. Meijer and J. Miller, *Technical Overview of the Common Language Runtime (or why the JVM is not my favourite execution environment).* URL: `http://docs.msdnaa.net/ark/Webfiles/whitepapers.htm`, 2001.
[23] D. Remy and J. Vouillon, *Objective ML: An effective object-oriented extension to ML*, Theory and Practice of Object Systems, 4 (1998), pp. 27–50.
[24] J. C. Reynolds, *Definitional interpreters for higher-order programming languages*, Higher-Order and Symbolic Computation, 11 (1998), pp. 363–397.
[25] M. Schinz and M. Odersky, *Tail call elimination on the Java Virtual Machine*, in Proceedings of Babel'01, vol. 59 of Electronic Notes in Theoretical Computer Science, 2001.
[26] I. Stark, *Thimble — Threads for MLj*, in Proceedings of the First Scottish Functional Programming Workshop, no. RM/99/9 in Department of Computing and Electrical Engineering, Heriot-Watt University, Technical Report, 1999, pp. 337–346.
[27] D. Turner, *Elementary strong functional programming*, in Proceedings of the First International Symposium on Functional Programming Languages in Education, R.Plasmeijer and P.Hartel, eds., vol. LNCS 1022, Nijmegen, Netherlands, Dec. 1995, Springer.
[28] P. B. Vasconcelos and K. Hammond, *Inferring costs for recursive, polymorphic and higher-order functional programs*, in Proceedings of the 15th International Workshop on the Implementation of Functional Languages, G. Michaelson and P. Trinder, eds., LNCS, Springer-Verlag, 2003. To appear.
[29] D. Wakeling, *Compiling lazy functional programs for the Java Virtual Machine*, Journal of Functional Programming, 9 (1999), pp. 579–603.
[30] N. Wolverson, *Optimisation and resource bounds in Camelot compilation.* Final-year project report, University of Edinburgh, 2003. Available at `http://groups.inf.ed.ac.uk/mrg/publications/wolverson.ps`.
[31] N. Wolverson and K. MacKenzie, *O'Camelot: adding objects to a resource-aware functional language*, in Proceedings of TFP2003, Intellect, 2004, pp. 47–62.

Edited by: Frédéric Loulergue
Received: June 15, 2004
Accepted: June 9, 2005

Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 31–41. http://www.scpe.org
ISSN 1895-1767

E.V.E., AN OBJECT ORIENTED SIMD LIBRARY

JOEL FALCOU AND JOCELYN SEROT*

Abstract. This paper describes the EVE (Expressive Velocity Engine) library, an object oriented C++ library designed to ease the process of writing efficient numerical applications using AltiVec, the SIMD extension designed by Apple, Motorola and IBM. AltiVec-powered applications typically show off a relative speed up of 4 to 16 but need a complex and awkward programmation style. By using various template metaprogramming techniques, E.V.E. provides an easy to use, STL-like, interface that allows developer to quickly write efficient and easy to read code. Typical applications written with E.V.E. can benefit from a large fraction of theorical maximum speed up while being written as simple C++ arithmetic code.

1. Introduction.

1.1. The AltiVec Extension. Recently, SIMD enhanced instructions have been proposed as a solution for delivering higher microprocessor hardware utilisation. SIMD (Single Instruction, Multiple Data) extensions started appearing in 1994 in HP's MAX2 and Sun's VS extensions and can now be found in most of microprocessors, including Intel's Pentiums (MMX/SSE/SSE2) and Motorola/IBM's PowerPCs (Altivec). They have been proved particularly useful for accelerating applications based upon data-intensive, regular computations, such as signal or image processing.

AltiVec [10] is an extension designed to enhance PowerPC[1] processor performance on applications handling large amounts of data. The AltiVec architecture is based on a SIMD processing unit integrated with the PowerPC architecture. It introduces a new set of 128 bit wide registers distinct from the existing general purpose or floating-point registers. These registers are accessible through 160 new "vector" instructions that can be freely mixed with other instructions (there are no restriction on how vector instructions can be intermixed with branch, integer or floating-point instructions with no context switching nor overhead for doing so). Altivec handles data as 128 bit vectors that can contain sixteen 8 bit integers, eight 16 bit integers, four 32 bit integers or four 32 bit floating points values. For example, any vector operation performed on a `vector char` is in fact performed on sixteen `char` simultaneously and is theoretically running sixteen times faster as the scalar equivalent operation. AltiVec vector functions cover a large spectrum, extending from simple arithmetic functions (additions, subtractions) to boolean evaluation or lookup table solving.

Altivec is natively programmed by means of a C API [5]. Programming at this level can offer significant speedups (from 4 to 12 for typical signal processing algorithms) but is a rather tedious and error-prone task, because this C API is really "assembly in disguise". The application-level vectors (arrays, in variable number and with variable sizes) must be explicitly mapped onto the Altivec vectors (fixed number, fixed size) and the programmer must deal with several low-level details such as vector padding and alignment. To correctly turn a scalar function into a vector-accelerated one, a large part of code has to be rewritten.

Consider for example a simple 3x1 smoothing filter (Fig. 1.1):

```
void C_filter( char* d, short* r)
{
  for(int i=1; i<SIZE-1; i++ )
    r[i] = (d[i-1]+2*d[i]+d[i+1])/4;
}
```

FIG. 1.1. *A simple 3x1 gaussian filter written in standard C.*

This code can be rewritten ("vectorized") using Altivec vector functions. However, this rewriting is not trivial. We first have to look at the original algorithm in a *parallel* way. The `C_filter` function is based on an iterative algorithm that runs trough each item of the input data, applies the corresponding operations and writes the result into the output array. By contrast, AltiVec functions operate on a bunch of data simultaneously. We have to recraft the algorithm so that it works on vectors instead of single scalar values. This is done by

*LASMEA, UMR 6602 CNRS / U. Clermont Ferrand, France (falcou, jserot@lasmea.univ-bpclermont.fr).
[1]PPC 74xx (G4) and PPC 970 (G5).

loading data into AltiVec vectors, shifting these vectors left and right, and performing vector multiplication and addition. The resulting code—which is indeed significantly longer than the original one—is given in Appendix A. We have benchmarked both the scalar and vectorized implementation on a 2 GHz PowerPC G5 and obtained the results shown in Table 1.1. Both code were compiled using `gcc 3.3` using `-O3`. On this example, a ten fold acceleration can be observed with the AltiVec extension. However, the time spent to rewrite the algorithm in a "vectorized" way and the somehow awkward Altivec API can hinder the development of larger scale applications.

TABLE 1.1
Execution time and relative speed-up for 3x1 filters.

SIZE value	`C_filter`	`AV_filter`	Speed Up
16 K	0.209 ms	0.020 ms	10.5
64 K	0.854 ms	0.075 ms	11.4
256 K	3.737 ms	0.322 ms	11.6
1024 K	16.253 ms	1.440 ms	11.3

2. AltiVec in high level API. As evidenced in the previous section, writing AltiVec-based applications can be a tedious task. A possible approach to circumvent this problem is to encapsulate Altivec vectors and the associated operations within a C++ class. Instantiating this class and using classic infix notations will produce the AltiVec code. We actually built such a class (`AVector`) and used it to encode the filtering example introduced in section 1.1. The resulting code is shown below.

```
AVector<char> img(SIZE);
AVector<short> res(SIZE);
res = (img.sr(1)+2*img+img.sl(1))/4;
```

In this formulation, explicit iterations have been replaced by application of overloaded operators on `AVector` objects. The `sr` and `sl` methods implements the shifting operations. The performance of this code, however, is very disappointing. With the array sizes shown in Table 1.1, the measured speed-ups never exceed 1. The reasons for such behaviour are given below.

Consider a simple code fragment using overloaded operators as shown below:

```
AVector<char> r(SIZE),x(SIZE),y(SIZE),z(SIZE);
r = x + y + z;
```

When a C++ compiler analyses this code, it reduces the successive operator calls iteratively, resolving first `y+z` then `x+(y+z)` where `y+z` is in fact stored in a temporary object. Moreover, to actually compute `x+y+z`, the involved operations are carried out within a loop that applies the `vec_add` function to every single vector element of the array. An equivalent code, after operator reduction and loop expansion is:

```
AVector<char> r(SIZE),x(SIZE),y(SIZE),z(SIZE);
AVector<char> tmp1(SIZE),tmp2(SIZE);
for(i=0;i<SIZE/16);i++) tmp1[i] = vec_add(y[i],z[i]);
for(i=0;i<SIZE/16);i++) tmp2[i] = vec_add(x[i],tmp1[i]);
for(i=0;i<SIZE/16);i++) r[i] = tmp2[i];
```

FIG. 2.1. *Expanded code for overloaded operator compilation*

This code can be compared to an "optimal", *hand-written* Altivec code like the one shown on figure 2.2. The code generated by the "naive" AltiVec class clearly exhibits unnecessary loops and copies. When expressions get more complex, the situation gets worse. The time spent in loop index calculation and temporary object copies quickly overcomes the benefits of the SIMD parallelization, resulting in poor performances.

This can be explained by the fact that all C++ compilers use a dyadic reduction scheme to evaluate operators composition. Some compilers[2] can output a slightly better code when certain optimisations are

[2]Like Code Warrior or gcc.

```
AVector<char> r(SIZE),x(SIZE),y(SIZE),z(SIZE);
for(i=0;i<SIZE/16);i++) r[i] = vec_add(x[i],vec_add(y[i],z[i]));
```

FIG. 2.2. *Optimal, hand written AltiVec code for x+y+z computation*

turned on. However, large expressions or complex functions call can't be totally optimised. Another factor is the impact of the order of AltiVec instructions. When writing AltiVec code, one have to take in account the fact that cache lines have to be filled up to their maximum. The typical way for doing so is to pack the loading instructions together, then the operations and finally the storing instructions. When loading, computing and storing instructions are mixed in an unordered way, AltiVec performances generally drop.

The aforementioned problem has already been identified—in [13] for example—and is the major inconvenient of the C++ language when it is used for high-level scientific computations. In the domain of C++ scientific computing, it has led to the development of the so-called *Active Libraries* [15, 2, 14, 1], which both provide domain-specific abstractions and dedicated code optimisation mechanisms. This paper describes how this approach can be applied to the specific problem of generating efficient Altivec code from a high-level C++ API.

It is organized as follows. Sect. 3 explains why generating efficient code for vector expressions is not trivial and introduces the concept of template-based meta-programming. Sect. 4 explains how this concept can used to generate optimised Altivec code. Sect. 5 rapidly presents the API of the library we built upon these principles. Performance results are presented in Sect. 6. Sect. 7 is a brief survey of related work and Sect. 8 concludes.

3. Template based Meta Programming. The evaluation of any arithmetic expression can be viewed as a two stages process:

- A first step, performed at compile time, where the structure of the expression is analysed to produce a chain of function calls.
- A second step, performed at run time, where the actual operands are provided to the sequence of function calls and the afferent computations are carried out.

When the expression structure matches certain pattern or when certain operands are known at compile time, it is often possible to perform a given set of computations at compile time in order to produce an optimised chain of function calls. For example, if we consider the following code:

```
for( int i=0;i<SIZE;i++)
{
  table[i] = cos(2*i);
}
```

If the size of the table is known at compile time, the code could be optimised by removing the loop entirely and writing a linear sequence of operations:

```
table[0] = cos(0);
table[1] = cos(2);

// later \dots

table[98] = cos(196);
table[99] = cos(198);
```

Furthermore, the value `cos(0)`, ..., `cos(198)` can be computed once and for all at compile-time, so that the runtime cost of such initialisation boils down to 100 store operations.

Technically speaking, such a "lifting" of computations from runtime to compile-time can be implemented using a mechanism known as *template-based metaprogramming*. The sequel of this section gives a brief account of this technique and of its central concept, *expressions templates*. More details can be found, for example, in Veldhuizen's papers [11, 12, 13]. We focus here on how this technique can be used to remove unnecessary loops and object copies from the code produced for the evaluation of vector based expressions.

The basic idea behind *expressions templates* is to encode the abstract syntax tree (AST) of an expression as a C++ recursive template class and use overloaded operators to build this tree. Combined with an array-like

container class, it provides a way to build a static representation of an array-based expression. For example, if we consider an float `Array` class and an addition functor `add`, the expression `D=A+B+C` could be represented by the following C++ type:

```
Xpr<Array,add,Xpr<Array,add,Array>>
```

Where `Xpr` is defined by the following type:

```
template<class LEFT,class OP,class RIGHT>
class Xpr
{
   public:
   Xpr( float* lhs, float* rhs ) : mLHS(lhs), mRHS(rhs) {}

   private:
   LEFT  mLHS;
   RIGHT mRHS;
};
```

The `Array` class is defined as below:

```
class Array
{
   public:
   Array( size_t s ) { mData = new float[s]; mSize = s;}
~Array() {if(mData) delete[] mData; }
   float* begin() { return mData; }

   private:
   float *mData;
   size_t mSize;
};
```

This type can be automatically built from the concrete syntax of the expression using an overloaded version of the '+' operator that takes an `Array` and an `Xpr` object and returns a new `Xpr` object:

```
Xpr< Array,add,Array> operator+(Array a, Array b)
{
  return Xpr<T,add,Array>(a.begin(),b.begin());
}
```

Using this kind of operators, we can simulate the parsing of the above code ("A+B+C") and see how the classes get combined to build the expression tree:

```
Array A,B,C,D;
D = A+B+C;
D = Xpr<Array,add,Array> + C
D = Xpr<Xpr<Array,add,Array>,add,Array>
```

Following the classic C++ operator resolution, the A+B+C expression is parsed as (A+B)+C. The A+B part gets encoded into a first template type. Then, the compiler reduce the X+C part, producing the final type, encoding the whole expression.

Handling the assignation of A+B+C to D can then be done using an overloaded version of the assignment operator:

```
template<class XPR> Array& Array::operator=(const XPR& xpr)
{
  for(int i=0;i<mSize;i++) mData[i] = xpr[i];
  return *this;
}
```

The `Array` and `Xpr` classes have to provide a `operator[]` method to be able to evaluate `xpr[i]`:

```
int Array::operator[](size_t index)
{
    return mData[index];
}

template<class L,class OP,class R>
int Xpr<L,OP,R>::operator[](size_t index)
{
  return OP::eval(mLHS[i],mRHS[i]);
}
```

We still have to define the `add` class code. Simply enough, `add` is a functor that exposes a static method called `eval` performing the actual computation. Such functors can be freely extended to include any other arithmetic or mathematical functions.

```
class add
{
  static int eval(int x,int y) { return x+y; }
}
```

With these methods, each reference to `xpr[i]` can be evaluated. For the above example, this gives:

```
data[i] = xpr[i];
data[i] = add::eval(Xpr<Array,add,Array>,C[i]);
data[i] = add::eval(add::apply(A[i],B[i]),C[i]);
data[i] = add::eval(A[i]+B[i],C[i]);
data[i] = A[i]+B[i]+C[i];
```

4. Application to efficient AltiVec code generation. At this stage, we can add AltiVec support to this meta-programming engine. If we replace the scalar computations and the indexed accesses by vector operations and loads, we can write an AltiVec template code generator. These changes affect all the classes and functions shown in the previous sections.

The Array class now provides a `load` method that return a vector instead of a scalar:

```
int Array::load(size_t index) { return vec_ld(data_,index*16); }
```

The add functor now use `vec_add` functions instead of the standard + operator:

```
class add
{
  static vector int eval(vector int x,vector int y)
    { return vec_add(x,y); }
}
```

Finally, we use `vec_st` to store results:

```
template<class XPR> Array& Array::operator=(const XPR& xpr)
{
    for(size_t i=0;i<mSize/4;i++)   vec_st(xpr.load(i),0,mData);
    return *this;
}
```

The final result of this code generation can be observed on figure 4.1.b for the A+B+C example. Figure 4.1.a gives the code produced by `gcc` when using the `std::valarray` class.

For this simple task, one can easily see that the minimum number of loads operation is three and the minimum number of store operations is one. For the standard code, we have seven extraneous `lwz` instructions to load pointers, three `lsfx` to load the actual data and one `stfs` to store the result. For the optimised code, we have replaced the scalar `lsfx` with the AltiVec equivalent `lvx`, the scalar `fadds` with `vaddfp` and `stfsx` with the vector `stvx`. Only three load instructions and one store instructions, reducing opcode count from 17 to 9.

(a) std::valarray code

```
L253:
lwz r9,0(r3)
slwi r2,r12,2
lwz r4,4(r3)
addi r12,r12,1
lwz r11,4(r9)
lwz r10,0(r9)
lwz r7,4(r11)
lwz r6,4(r10)
lfsx f0,r7,r2
lfsx f1,r6,r2
lwz r0,4(r4)
fadds f2,f1,f0
lfsx f3,r2,r0
fadds f1,f2,f3
stfs f1,0(r5)
addi r5,r5,4
bdnz L253
```

(b) optimized code

```
L117:
slwi r2,r9,4
addi r9,r9,1
lvx v1,r5,r2
lvx v0,r4,r2
lvx v13,r6,r2
vaddfp v0,v0,v1
vaddfp v1,v0,v13
stvx v1,r2,r8
bdnz L117
```

FIG. 4.1. *Assembly code for a simple vector operation*

5. The EVE library. Using the code generation technique described in the previous section, we have produced a high-level array manipulation library aimed at scientific computing and taking advantage of the SIMD acceleration offered by the Altivec extension on PowerPC processors. This library, called EVE (for *Expressive Velocity Engine*) basically provides two classes, `vector` and `matrix`—for 1D and 2D arrays –, and a rich set of operators and functions to manipulate them. This set can be roughly divided in four families:

1. **Arithmetic and boolean operators**, which are the direct vector extension of their C++ counterparts. For example:

   ```
   vector<char>  a(64),b(64),c(64),d(64);
   d = (a+b)/c;
   ```

2. **Boolean predicates**. These functions can be used to manipulate boolean vectors and use them as selection masks. For example:

   ```
   vector <char>  a(64),b(64),c(64);
   // c[i] = a[i] if a[i]<b[i], b[i] otherwise
   c = where(a < b, a, b);
   ```

3. **Mathematical and STL functions**. These functions work like their STL or `math.h` counterparts. The only difference is that they take an array (or matrix) as a whole argument instead of a couple of iterators. Apart from this difference, EVE functions and operators are very similar to their STL counterparts (the interface to the EVE `array` class is actually very similar to the one offered by the STL `valarray` class. This allows algorithms developed with the STL to be ported (and accelerated) with a minimum effort on a PowerPC platform with EVE. For example:

   ```
   vector <float>  a(64),b(64);
   b = tan(a);
   float r = inner_product(a, b);
   ```

4. **Signal processing functions**. These functions allow the direct expression (without explicit decomposition into sums and products) of 1D and 2D FIR filters. For example:

   ```
   matrix<float>  image(320,240),res(320,240);
   filter<3,horizontal>  gauss_x = 0.25, 0.5, 0.25;
   res = gauss_x(image);
   ```

The EVE API allows the developer to write a large variety of algorithms as long as these algorithm can be expressed as a serie of global operation on vector.

6. Performance. Two kinds of performance tests have been performed: basic tests, involving only one vector operation and more complex tests, in which several vector operations are composed into more complex expressions. All tests involved vectors of different types (8 bit integers, 16 bit integers, 32 bit integers and 32 bit floats) but of the same total length (16 Kbytes) in order to reduce the impact of cache effects on the observed performances[3]. They have been conducted on a 2GHz PowerPC G5 with `gcc 3.3.1` and the following compilation switches: `-faltivec -ftemplate-deph-128 -O3`. A selection of performance results is given in Table 6.1. For each test, four numbers are given: the maximum theoretical speedup[4] (TM), the measured speedup for a hand-coded version of the test using the native C Altivec API (N.C), the measured speedup with a "naive" vector library—which does not use the expression template mechanism described in Sect. 3 (N.V), and the measured speedup with the EVE library.

TABLE 6.1
Selected performance results

Test	Vector type	TM	N.C	N.V	EVE
1. `v3=v1+v2`	8 bit integer	16	15.7	8.0	15.4
2. `v2=tan(v1)`	32 bit float	4	3.6	2.0	3.5
3. `v3=v1/v2`	32 bit float	4	4.8	2.1	4.6
4. `v3=v1/v2`	16 bit integer	8(4)	3.0	1.0	3.0
5. `v3=inner_prod(v1,v2)`	8 bit integer	8	7.8	4.5	7.2
6. `v3=inner_prod(v1,v2)`	32 bit float	4	14.1	4.8	13.8
7. 3x1 Filter	8 bit integer	8	7.9	0.1	7.8
8. 3x1 Filter	32 bit float	4	4.12	0.1	4.08
9. `v5=sqrt(tan(v1+v2)/cos(v3*v4))`	32 bit float	4	3.9	0.04	3.9

It can be observed that, for most of the tests, the speedup obtained with EVE is close to the one obtained with a hand-coded version of the algorithm using the native C API. By contrast, the performances of the "naive" class library are very disappointing (especially for tests 7-10). This clearly demonstrates the effectiveness of the metaprogramming-based optimisation.

Tests 1–3 correspond to basic operations, which are mapped directly to a single AltiVec instruction. In this case, the measured speedup is very close to the theoretical maximum. For test 3, it is even greater. This effect can be explained by the fact that on G5 processors, and even for non-SIMD operations, the Altivec FPU is already faster than the scalar FPU[5]. When added to the speedup offered by the SIMD parallelism, this leads to super-linear speedups. The same effect explains the result obtained for test 6. By contrast, test 4 exhibits a situation in which the observed performances are significantly lower than expected. In this case, this is due to the asymmetry of the Altivec instruction set, which does not provide the basic operations for all types of vectors. In particular, it does not include division on 16 bit integers. This operation must therefore be emulated using vector float division. This involves several type casting operations and practically reduces the maximum theoretical speedup from 8 to 4.

Tests 5-9 correspond to more complex operations, involving *several* AltiVec instructions. Note that for tests 5 and 7, despite the fact that the operands are vectors of 8 bit integers, the computations are actually carried out on vectors of 16 bit integers, in order to keep a reasonable precision. The theoretical maximum speedup is therefore 8 instead of 16.

6.1. Realistic Case Study. In order to show that EVE can be used to solve realistic problems, while still delivering significant speedups, we have used it to vectorize several complete image processing algorithms. This section describes the implementation of an algorithm performing the detection of *points of interest* in grey scale images using the Harris filter [7].

[3] I.e. the vector size (in elements) was 16K for 8 bit integers, 8K for 16 bit integers and 4K for 32 bits integers or floats.
[4] This depends on the type of the vector elements: 16 for 8 bit integers, 8 for 16 bit integers and 4 for 32 bit integers and floats.
[5] It has more pipeline stages and a shortest cycle time.

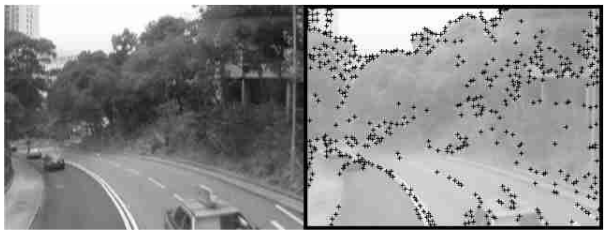
Starting from an input image $I(x,y)$, horizontal and vertical gaussian filters are applied to remove noise and the following matrix is computed:

$$M(x,y) = \begin{pmatrix} (\frac{\partial I}{\partial x})^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & (\frac{\partial I}{\partial y})^2 \end{pmatrix}$$

Where $(\frac{\partial I}{\partial x})$ and $(\frac{\partial I}{\partial y})$ are respectively the horizontal and vertical gradient of $I(x,y)$. $M(x,y)$ is filtered again with a gaussian filter and the following quantity is computed:

$$H(x,y) = Det(M) - k.trace(M)^2, k \in [0.04; 0.06]$$

H is viewed as a measure of pixel interest. Local maxima of H are then searched in 3x3 windows and the n^{th} first maxima are finally selected. Figure 6.1 shows the result of the detection algorithm on a video frame picturing an outdoor scene.



In this implementation, only the filtering and the pixel detection are vectorized. Sorting an array cannot be easily vectorized with the AltiVec instruction set. It's not worth it anyway, since the time spent in the final sorting and selection process only accounts for a small fraction (around 3%) of the total execution time of the algorithm. The code for computing M coefficients and H values is shown in Fig. 6.1. It can be split into three sections:

1. **A declarative section** where the needed `matrix` and `filter` objects are instantiated. `matrix` objects are declared as `float` containers to prevent overflow when filtering is applied on the input image and to speed up final computation by removing the need for type casting.
2. **A filtering section** where the coefficients of the M matrix are computed. We use EVE filter objects, instantiated for gaussian and gradient filters. Filter support an overloaded * operator that is semantically used as the composition operator.
3. **A computing section** where the final value of $H(x,y)$ is computed using the overloaded versions of arithmetic operators.

The performances of this detector implementation have been compared to those of the same algorithm written in C, both using 320*240 pixels video sequence as input. The tests were run on a 2GHz Power PC G5 and compiled with `gcc 3.3`. As the two steps of the algorithm (filtering and detection) use two different parts of the E.V.E. API, we give the execution time for each step along with the total execution time.

Step	Execution Time	Speed Up
Filtering	$1.4ms$	5.21
Evaluation	$0.45ms$	4.23
Total Time	$1.85ms$	4.98

The performance of both parts of the algorithm are satisfactory. The filtering section speed-up is near 65% of maximum speed-up while the second part benefits from a superlinear acceleration.

7. Related Work. Projects aiming at simplifying the exploitation of the SIMD extensions of modern micro-processors can be divided into two broad categories: compiler-based approaches and library-based approaches.

```
// Declarations
#define W 320
#define H 240
matrix<short> I(W,H),a(W,H),b(W,H);
matrix<short> c(W,H),t1(W,H),t2(W,H);
matrix<float> h(W,H);
float k = 0.05f;

filter<3,horizontal> smooth_x = 1,2,1;
filter<3,horizontal> grad_x = 1,0,1;
filter<3,vertical> smooth_y = 1,2,1;
filter<3,vertical> grad_y = -1,0,1;

// Computes matrix M:
//
//          | a c |
//     M =  | c b |

t1 = grad_x(I);
t2 = grad_y(I);
a  = (smooth_x*smooth_y)(t1*t1);
b  = (smooth_x*smooth_y)(t2*t2);
c  = (smooth_x*smooth_y)(t1*t2);

// Computes matrix H
H = (a*b-c*c)-k*(a+b)*(a+b);
```

FIG. 6.1. *The Harris detector, coded with* EVE

The SWAR (SIMD Within A register, [4]) project is an example of the first approach. Its goal is to propose a versatile data parallel C language making full SIMD-style programming models effective for commodity microprocessors. An experimental compiler (SCC) has been developed that extends C semantics and type system and can target several family of microprocessors. Started in 1998, the project seems to be in dormant state.

Another example of the compiler-based approach is given by Kyo *et al.* in [8]. They describe a compiler for a parallel C dialect (1DC, One Dimensional C) producing SIMD code for Pentium processors and aimed at the succinct description of parallel image processing algorithms. Benchmarks results show that speed-ups in the range of 2 to 7 (compared with code generated with a conventional C compiler) can be obtained for low-level image processing tasks. But the parallelization techniques described in the work—which are derived from the one used for programming linear processor arrays—seems to be only applicable to simple image filtering algorithms based upon sweeping a horizontal pixel-updating line row-wise across the image, which restricts its applicability. Moreover, and this can be viewed as a limitation of compiler-based approaches, retargeting another processor may be difficult, since it requires a good understanding of the compiler internal representations.

The VAST code optimiser [3] has a specific back-end for generating Altivec/Power PC code. This compiler offers automatic vectorization and parallelization from conventional C source code, automatically replacing loops with inline vector extensions. The speedups obtained with VAST are claimed to be closed to those obtained with hand-vectorized code. VAST is a commercial product.

There have been numerous attempts to provide a library-based approach to the exploitation of SIMD features in micro-processors. Apple VECLIB [6], which provides a set of Altivec-optimised functions for signal processing, is an example. But most of these attempts suffer from the weaknesses described in Sect. 2; namely, they cannot handle complex vector expressions and produce inefficient code when multiple vector operations are involved in the same algorithm. MacSTL [9] is the only work we are aware of that aims at eliminating these weaknesses while keeping the expressivity and portability of a library-based approach. MacSTL is actually very similar to EVE in goals and design principles. This C++ class library provides a fast `valarray` class

optimised for Altivec and relies on template-based metaprogramming techniques for code optimisation. The only difference is that it only provides STL-compliant functions and operators (it can actually be viewed as a specific implementation of the STL for G4/G5 computers) whereas EVE offers additional domain-specific functions for signal and image processing.

8. Conclusion. We have shown how a classical technique—template-based metaprogramming—can be applied to the design and implementation of an efficient high-level vector manipulation library aimed at scientific computing on PowerPC platforms. This library offers a significant improvement in terms of expressivity over the native C API traditionally used for taking advantage of the SIMD capabilities of this processor. It allows developers to obtain significant speedups without having to deal with low level implementation details. Moreover, The EVE API is largely compliant with the STL standard and therefore provides a smooth transition path for applications written with other scientific computing libraries. A prototype version of the library can be downloaded from the following URL: `http://wwwlasmea.univ-bpclermont.fr/Personnel/Joel.Falcou/eng/eve`. We are currently working on improving the performances obtained with this prototype. This involves, for instance, globally minimizing the number of vector load and store operations, using more judiciously Altivec-specific cache manipulation instructions or taking advantage of fused operations (e. g. multiply/add). Finally, it can be noted that, although the current version of EVE has been designed for PowerPC processors with Altivec, it could easily be retargeted to Pentium 4 processors with MMX/SSE2 because the code generator itself (using the expression template mechanism) can be made largely independent of the SIMD instruction set.

REFERENCES

[1] *The BOOST Library.* `http://www.boost.org/`.
[2] *The POOMA Library.* `http://www.codesourcery.com/pooma/`.
[3] *VAST.* `http://www.psrv.com/vast_altivec.html/`.
[4] *The SWAR Home Page* `http://shay.ecn.purdue.edu/~swar` Purdue University
[5] Apple, *The AltiVec Instructions References Page.* `http://developer.apple.com/hardware/ve`.
[6] Apple, *VecLib framework.* `http://developer.apple.com/hardware/ve/vector_libraries.html`
[7] C. Harris and M. Stephens, *A combined corner and edge detector.* In 4th Alvey Vision Conference, 1988.
[8] S. Kyo and S. Okasaki and I. Kuroda, *An extended C language and a SIMD compiler for efficient implementation of image filters on media extended micro-processors.* in Proceedings of Acivs 2003 (Advanced Concepts for Intelligent Vision Systems), Ghent, Belgium, Sept. 1998
[9] G. Low, *Mac STL.* `http://www.pixelglow.com/macstl/`.
[10] I. Ollman, *AltiVec Velocity Engine Tutorial.* `http://www.simdtech.org/altivec`. March 2001.
[11] T. Veldhuizen, *Using C++ Template Meta-Programs.* In C++ Report, vol. 7, p. 36-43,1995.
[12] ———, *Expression Templates.* In C++ Report, vol. 7, p. 26-31, 1995.
[13] ———, *Techniques for Scientific C++.* `http://osl.iu.edu/ tveldhui/papers/techniques/`.
[14] ———, *Arrays in Blitz++.* In Dr Dobb's Journal of Software Tools, p. 238-44, 1996.
[15] T. Veldhuizen and D. Gannon, *Active Libraries: Rethinking the roles of compilers and libraries* Proc. of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing. SIAM Press, 1998

Appendix A. A simple 3x1 gaussian filter written with the Altivec native C API .

```
void AV_filter( char* img, short* res)
{
  vector unsigned char zu8,t1,t2,t3,t4;
  vector signed short x1h,x1l,x2h;
  vector signed short x2l,x3h,x3l;
  vector signed short zs16 ,rh,rl,v0,v1,shift;

  // Generate constants
  v0    = vec_splat_s16(2);
  v1    = vec_splat_s16(4);
  zu8   = vec_splat_u8(0);
  zs16  = vec_splat_s16(0);
  shift = vec_splat_s16(8);

  for( int j = 0; j< SIZE/16 ; j++ )
  {
    // Load input vectors
    t1 =  vec_ld(j*16, img); t2 =  vec_ld(j*16+16, img);

    // Generate shifted vectors
    t3 =  vec_sld(t1,t2,1); t4 =  vec_sld(t1,t2,2);

    // Cast to short
    x1h = vec_mergeh(zu8,t1); x1l = vec_mergel(zu8,t1);
    x2h = vec_mergeh(zu8,t3); x2l = vec_mergel(zu8,t3);
    x3h = vec_mergeh(zu8,t4); x3l = vec_mergel(zu8,t4);

    // Actual filtering
    rh = vec_mladd(x1h,v0,zs16);
    rl = vec_mladd(x1l,v0,zs16);
    rh = vec_mladd(x2h,v1,rh);
    rl = vec_mladd(x2l,v1,rl);
    rh = vec_mladd(x3h,v0,rh);
    rl = vec_mladd(x3l,v0,rl);
    rh = vec_sr(rh,shift);
    rl = vec_sr(rl,shift);

    // Pack and store result vector
    t1 = vec_packsu(rh,rl);
    vec_st(t1,j,out);
  }
}
```

Edited by: Frédéric Loulergue
Received: June 26, 2004
Accepted: June 5, 2005

Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 43–69. http://www.scpe.org
ISSN 1895-1767

EXTERNAL MEMORY IN BULK-SYNCHRONOUS PARALLEL ML*

FRÉDÉRIC GAVA†

Abstract. A functional data-parallel language called BSML was designed for programming Bulk-Synchronous Parallel algorithms, a model of computing which allows parallel programs to be ported to a wide range of architectures. BSML is based on an extension of the ML language with parallel operations on a parallel data structure called parallel vector. The execution time can be estimated. Dead-locks and indeterminism are avoided. For large scale applications where parallel processing is helpful and where the total amount of data often exceeds the total main memory available, parallel disk I/O becomes a necessity. In this paper, we present a library of I/O features for BSML and its formal semantics. A cost model is also given and some preliminary performance results are shown for a commodity cluster.

Key words. Parallel Functional Programming, Parallel I/O, Semantics, BSP.

1. Introduction. Some problems require performance that can only be provided by massively parallel computers. Programming these kind of computers is still difficult. Many important computational applications involve solving problems with very large data sets [44]. Such applications are also referred as *out-of-core* applications. For example astronomical simulation [47], crash test simulation [10], geographic information systems [32], weather prediction [52], computational biology [17], graphs [40] or computational geometry [11] and many other scientific problems can involve data sets that are too large to fit in the main memory and therefore fall into this category. For another example, the Large Hadron Collider of the CERN laboratory for finding traces of exotic fundamental particles (web page at `lhc-new-homepage.web.cern.ch`), when starts running, this instrument will produces about 10 Petabytes a month. The earth-simulator, the most powerful parallel machine in the top500 list, has 1 Petabyte of total main memory and 100 Petabytes of secondary memories. Using the main memory is not enough to store all the data of an experiment.

Using parallelism can reduce the computation time and increase the available memory size, but for challenging applications, the memory is always insufficient in size. For instance, in a mesh decomposition of a mechanical problem, a scientist might want to increase the mesh size. To increase the available memory size, a trivial solution is to use the *virtual memory* mechanism present in modern operating systems. This has been established as a standard method for managing external memory. Its main advantage is that it allows the application to access to a *large* virtual memory without having to deal with the intricacies of blocked secondary memory accesses. Unfortunately, this solution is inefficient if standard *paging policy* is employed [7]. To get the best performances, the algorithms must be restructured with explicit I/O calls on this secondary memory.

Such algorithms are generally called *external memory* (EM) algorithms and are designed for *large computational* problems in which the size of the internal memory of the computer is only a small fraction of the size of the problem ([55, 53] for a survey). Parallel processing is an important issue for EM algorithms for the same reasons that parallel processing is of practical interest in non-EM algorithm design. Existing algorithm and data structures were often unsuitable for out-of-core applications. This is largely due to the need of locality on data references, which is not generally present when algorithms are designed for internal memory due to the permissive nature of the PRAM model: parallel EM algorithms [54] are "new" and do not work optimally and correctly in "classical" parallel environments.

Declarative parallel languages are needed to simplify the programming of massively parallel architectures. Functional languages are often considered. The design of parallel programming languages is a tradeoff between the possibility to express the parallel features that are necessary for predictable efficiency (but with programs that are more difficult to write, prove and port) and the abstraction of such features that are necessary to make parallel programming easier (but which should not hinder efficiency and performance prediction). On the one hand the programs should be efficient but without the price of non portability and unpredictability of performances. The portability of code is needed to allow code reuse on a wide variety of architectures. The predictability of performances is needed to guarantee that the efficiency will always be achieved, whatever architecture is used.

*This work is supported by the ACI Grid program from the French Ministry of Research,under the project Caraml (`http://www.caraml.org`)

†Laboratory of Algorithms, Complexity and Logic (LACL), University of Paris XII, Val-de-Marne, 61 avenue du Général de Gaulle, 94010 Créteil cedex – France, `gava@univ-paris12.fr`

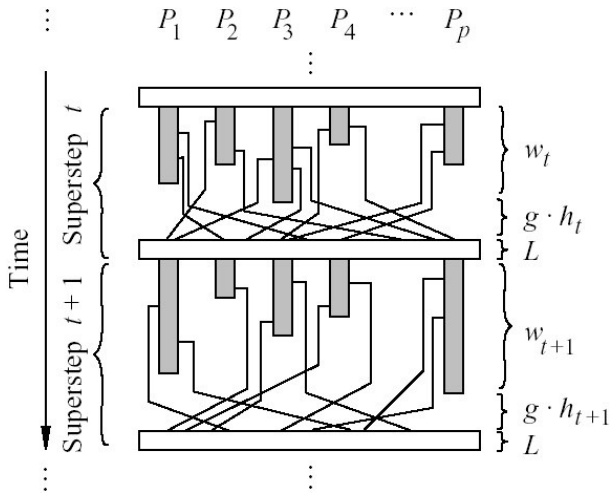


FIG. 2.1. *The BSP model of computation*

Another important characteristic of parallel programs is the complexity of their semantics. Deadlocks and non-determinism often hinder the practical use of parallelism by a large number of users. To avoid these undesirable properties, there is a trade-off between the expressiveness of the language and its structure which could decrease the expressiveness.

We are currently exploring the intermediate position of the paradigm of algorithmic skeletons [6, 42] in order to obtain universal parallel languages where the execution cost can easily be determined from the source code. In this context, cost means the estimate of parallel execution time. This last requirement forces the use of explicit processes corresponding to the processors of the parallel machine. Bulk-Synchronous Parallel ML or *BSML* is an extension of ML for programming Bulk-Synchronous Parallel algorithms as functional programs with a compositional cost model. Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [46, 50] to offer a high degree of abstraction like PRAM models and yet to allow portable and predictable performance on a wide variety of architectures with a realistic cost model based on a structured parallelism. Deadlocks and indeterminism are avoided. BSP programs are portable across many parallel architectures. Such algorithms offer predictable and scalable performances ([38] for a survey) and BSML expresses them with a small set of primitives taken from the *confluent* BSλ calculus [37]. Such operations are implemented as a library for the functional, with a strict evaluation strategy, programming language Objective Caml [33]. We refer to [27] for more details about the choice of this strategy for massively parallel computing.

Parallel disk I/O has been identified as a *critical component* of a suitable *high performance* computer. Research in EM algorithms has recently received considerable attention. Over the last few years, comprehensive computing and cost models that incorporate disks and multiple processors have been proposed [35, 55, 54], but not with all the above elements. [14, 16] showed how an EM machine can take full advantage of parallel disk I/O and multiple processors. This model is based on an extension of the BSP model for I/O accesses. Our research aims at combining the BSP model with functional programming. We naturally need to also extend BSML with I/O accesses for programming EM algorithms. This paper is the follow-up to our work on imperative features of our functional data-parallel language [22].

This paper describes a further step after [21] towards this direction. The remainder of this paper is organized as follows. First we review the BSP model in Section 2 and, then, briefly present the BSML language. In section 3 we introduce the EM-BSP model and the problems that appear in BSML. In section 4, we then give new primitives for our language. In section 5, we describe the formal semantics of our language with persistent features. Section 6 is devoted to the formal cost model associated to our language and Section 7 to some benchmarks of a parallel program. We discuss related work in section 8 and we end with conclusions and future research (section 9).

2. Functional Bulk-Synchronous Parallel ML.

2.1. Bulk-Synchronous Parallelism. A BSP computer contains a set of *processor-memory* pairs, a *communication network* allowing inter-processor delivery of messages and a *global synchronization unit* which

executes collective requests of a *synchronization barrier*. For the sake of conciseness, we refer to [5, 46] for more details. In this model, a parallel computation is subdivided into *supersteps* (Figure 2.1) at the end of which a barrier synchronization and a routing are performed. After that, all requests for data posted during a preceding superstep are fulfilled. The performance of the machine is characterized by 3 parameters expressed as multiples of the local processing speed r:

(i) p is the number of processor-memory pairs;
(ii) l is the time required for a global synchronization and
(iii) g is the time for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word. The network can deliver an h-relation in time $g \times h$ for any arity h.

These parameters can easily be obtained using benchmarks [28]. The execution time of a superstep s is thus the sum of the maximal local processing time, the maximal data delivery time and the global synchronization time, i.e, $\text{Time}(s) = \max_{i:processor} w_i^s + \max_{i:processor} h_i^s * g + l$ where $w_i^s =$ local processing time on processor i during superstep s and $h_i^s = \max\{h_{i+}^s, h_{i-}^s\}$ where h_{i+}^s (resp. h_{i-}^s) is the number of words transmitted (resp. received) by processor i during superstep s. The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of S supersteps is therefore the sum of 3 terms:

$$t_{comp} + t_{comm} + L \text{ where } \left\{ \begin{array}{rcl} t_{comp} & = & \sum_s \max_i w_i^s \\ t_{comm} & = & H \times g \text{ where } H = \sum_s \max_i h_i^s \\ L & = & S \times l. \end{array} \right.$$

In general t_{comp}, H and S are functions of p and of the size of data n, or of more complex parameters like data skew and histogram sizes. To minimize execution time, the BSP algorithm design must jointly minimize the number S of supersteps and the total volume h (resp. t_{comp}) and imbalance h^s (resp. t_{comm}) of communication (resp. local computation). Bulk Synchronous Parallelism and the Coarse-Grained Multicomputer (CGM), which can be seen as a special case of the BSP model are used for a large variety of applications. As stated in [13] "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures between the late eighties and the time from the mid-nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain".

bsp_p: unit→int **bsp_l**: unit→float **bsp_g**: unit→float
mkpar: (int→α)→α **par**
apply: ($\alpha \to \beta$)**par**→α **par**→β **par**
type α option = None | Some **of** α
put: (int→α option)**par**→(int→α option)**par**
at: α **par**→int→α

FIG. 2.2. *The Core Bsmllib Library*

2.2. Bulk-Synchronous Parallel ML. BSML does not rely on SPMD programming. Programs are usual "sequential" Objective Caml (OCaml) programs [33] but work on a parallel data structure. Some of the advantages are simpler semantics and better readability. The execution order follows the reading order in the source code (or, at least, the results are such as seems to follow the execution order). There is currently no implementation of a full BSML language but rather a partial implementation as a library for OCaml (web page at `http://bsmllib.free.fr/`).

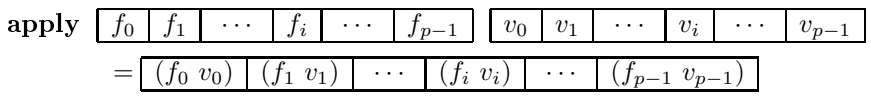
The so-called BSMLlib library is based on the elements given in Figure 2.2. They give access to the BSP parameters of the underling architecture: **bsp_p**() is p the *static* number of processes (this value does not change during execution), **bsp_g**() is g the time for collectively delivering a 1-relation and **bsp_l**() is l the time required for a global synchronization barrier.

There is an abstract polymorphic type α **par** which represents the type of p-wide parallel vectors of objects of type α one per processor. BSML parallel constructs operate on parallel vectors. Those parallel vectors are created by **mkpar** so that (**mkpar** f) stores (f i) on process i for i between 0 and $p-1$:

mkpar f = | (f 0) | (f 1) | ··· | (f i) | ··· | (f (p−1)) |

We usually write f as **fun** pid→e to show that the expression e may be different on each processor. This expression e is said to be *local*, i.e, a usual ML expression. The expression (**mkpar** f) is a parallel object and

it is said to be *global*. A usual ML expression which is not within a parallel vector is called *replicate*, i.e, identical to each processor. A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a superstep) and phases of global communication (second phase of a superstep) with global synchronization (third phase of a superstep). Asynchronous phases are programmed with **mkpar** and **apply** such that (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i) (e i)) on process i:



Let us consider the following expression:

let vf=**mkpar**(**fun** pid x→x+pid)
and vv=**mkpar**(**fun** pid→2∗pid+1)
in apply vf vv

The two parallel vectors are respectively equivalent to:

fun x→x + 0	**fun** x→x + 1	$\cdots$	**fun** x→x + i	$\cdots$	**fun** x→x + $(p-1)$

and

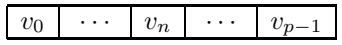
0	3	$\cdots$	$2\times i+1$	$\cdots$	$2\times(p-1)+1$

The expression **apply** vf vv is then evaluated to:

0	4	$\cdots$	$2\times i+2$	$\cdots$	$2\times(p-1)+2$

Readers familiar with BSPlib [28] will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by **put**. Consider the expression: **put**(**mkpar**(**fun** i→fs$_i$)) (∗). To send a value v from process j to process i, the function fs$_j$ at process j must be such that (fs$_j$ i) evaluates to Some v. To send no value from process j to process i, (fs$_j$ i) must evaluate to None. The expression (∗) evaluates to a parallel vector containing a function fd$_i$ of delivered messages on every process i. At process i, (fd$_i$ j) evaluates to None if process j sent no message to process i or evaluates to Some v if process j sent the value v to the process i.

The full language would also contain a synchronous projection operation **at**. (**at** vec n) returns the nth value of the parallel vector vec:



at expresses communication and synchronization phases. Without it, the global control cannot take into account data computed locally. Global conditional is necessary for expressing algorithms like: **Repeat** Parallel Iteration **Until** Max of local errors $< \epsilon$. The nesting of **par** types is prohibited and the projection should not be evaluated inside the scope of a **mkpar**. Our type system enforces these restrictions [23].

2.3. Examples.

2.3.1. Often Used Functions. Some useful functions can be defined by using only the primitives. For example the function replicate creates a parallel vector which contains the same value everywhere. The primitive **apply** can be used only for a parallel vector of functions which take only one argument. To deal with functions which take two arguments we need to define the apply2 function.

let replicate x = **mkpar**(**fun** pid→x)
let apply2 vf v1 v2 = **apply** (**apply** vf v1) v2

It is also common to apply the same sequential function at each process. This can be done using the parfun functions. They only differ in the number of arguments to apply:

let parfun f v = **apply**(replicate f) v
let parfun2 f v1 v2 = **apply**(parfun f v1) v2
let parfun3 f v1 v2 v3 = **apply**(parfun2 f v1 v2) v2

It is also common to apply a different function on a process. applyat n f_1 f_2 v applies function f_1 at process n and function f_2 at other processes:

```
let applyat n f1 f2 v =
 apply(mkpar(fun i→if i=n then f1 else f2)) v
```

2.3.2. Communication Function. Our example is the classical computation of the *prefix* of a list. Here we make the hypothesis that the elements of the list are distributed to all the processes as lists. Each processor performs a local reduction, then sends its partial result to the following processors and finally locally reduces its partial result with the sent values. Take for example the following expression:

scan_list_direct e (+) | [1; 2] | [3; 4] | [5] |

It will be evaluated to:

| $[e+1; e+1+2]$ | $[e+1+2+3; e+1+2+3+4;]$ | $[e+1+2+3+4+5]$ |

for a prefix of three processors and where e is the neutral element (here 0). To do this, we need first the computation of the prefix of a parallel vector:

```
(* scan_ direct:(α →α →α )→α →α par→α par *)
let scan_direct op e vv =
 let mkmsg pid v dst=if dst<pid then None else Some v in
 let procs_lists=mkpar(fun pid→from_to 0 pid) in
 let receivedmsgs=put(apply(mkpar mkmsg) vv) in
 let values_lists= parfun2 List.map
       (parfun (compose noSome) receivedmsgs) procs_lists in
  applyat 0 (fun _ →e) (List.fold_left op e) values_lists
```

$$\text{where} \begin{cases} \text{List.map } f\ [v_0; \ldots; v_n] & = & [(f\ v_0); \ldots; (f\ v_n)] \\ \text{List.fold\_left } f\ e\ [v_0; \ldots; v_n] & = & f\ (\cdots (f\ (f\ e\ v_0)\ v_1) \cdots)\ v_n \\ \text{from\_to } n\ m & = & [n; n+1; n+2; \ldots; m] \\ \text{noSome (Some } v) & = & v \\ \text{compose } f\ g\ x & = & (f\ (g\ x)). \end{cases}$$

Then, we can directly have the prefix of lists using some generic scan:

```
let scan_wide scan seq_scan_last map op e vv =
 let local_scan=parfun (seq_scan_last op e) vv in
 let last_elements=parfun fst local_scan in
 let values_to_add=(scan op e last_elements) in
 let pop=applyat 0 (fun x y→y) op in
  parfun2 map (pop values_to_add) (parfun snd local_scan)

let scan_wide_direct seq_scan_last map op e vv =
  scan_wide scan_direct seq_scan_last map op e vv

let scan_list scan op e vl =
 scan_wide scan seq_scan_last List.map op e vl
(* scan_list_direct:(α →α →α )→α →α list par→α list par *)
let scan_list_direct op e vl = scan_list scan_direct op e vl
```

where seq_scan_last $f\ e\ [v_0; v_1; \ldots; v_n] = (\text{last}, [(f\ e\ v_0); f(f\ e\ v_0)\ v_1; \ldots; \text{last}])$ where
last $= f\ (\cdots (f\ (f\ e\ v_0)\ v_1) \cdots)\ v_n$. The BSP cost formula of the above function (assuming op has a constant cost c_{op}) is thus $2 \times N \times c_{op} \times r\ + (p-1) \times s \times g\ +\ l$ where s denotes the size in words of a value compute by the scan and N the length of the biggest list held at a process. We have thus the time to compute the partial prefix, the time to send the partial results, time to perform the global synchronization and the time to finish the prefix.

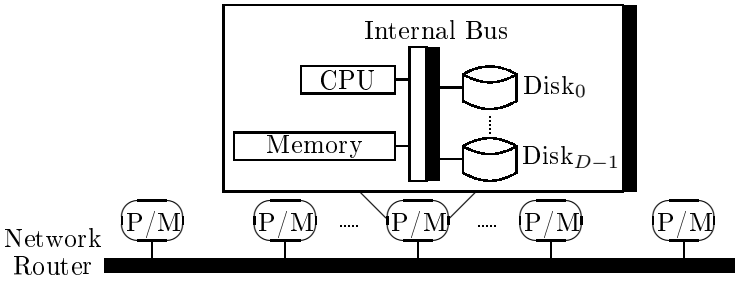


FIG. 3.1. *A BSP computer with external memories*

2.4. Advantages of Functional BSP Programming. One important benefit of the BSP model is the ability to accurately predict the execution time requirements of parallel algorithms. Communications are clearly separated from synchronization, i. e., this avoids deadlocks and it can be performed in any order, provided that the information is delivered at the beginning of the next superstep. This is achieved by constructing *analytical formulas* that are parameterized by a few values which captured the computation, communication and synchronization performance of the parallel system.

The clarity, abstraction and formal semantics of functional language make them desirable vehicles for complex software. The functional approach of this parallel model allows the re-use of suitable techniques from functional languages because a few number of parallel primitives is needed. Primitives of the BSML language with a strict strategy are derived from a confluent calculus [37] so parallel algorithms are also confluent and keep the advantages of the BSP models: no deadlock, efficient implementation using optimized communication algorithms, static cost formulas and cost previsions. The lazy evaluation strategy of pure functional language is not suited for the need of the massively parallel programmer. Lazy evaluation has the unwanted property of hiding complexity from the programmer [27]. The strict strategy of OCaml makes the BSMLlib a better tool for high performance applications because programs are transparent in the sense of making complexity explicit in the syntax.

Also, as in functional languages, we could easily *prove* and *certify* functional implementation of such algorithms with a proof assistant [1, 4] as in [20]. Using the *extraction* possibility of the proof assistant, we could generate a *certified* implementation to be used independently of the sequential or parallel implementation of the BSML primitives.

3. External Memory.

3.1. The EM-BSP model. Modern computers typically have several layers of memories which include the main memory and caches as well as disks. We restrict ourselves to the two-level model [54] because the speed difference between disks and the main memory is much more significant than between other layers of memories. [16] extended the BSP model to include secondary local memories. The basic idea is simple and it is illustrated in Figure 3.1. Each processor has, in addition to its local memory, an external memory (EM) in the form of a set of *disks*. This idea is applied to extend the BSP model to its EM version called **EM-BSP** by adding the following parameters to the standard BSP parameters:

(i) M is the local memory *size* of each processor;
(ii) D is the number of *disk drives* of each processor;
(iii) B is the *transfer block size* of a disk drive, and
(iv) G is the ratio of local computational capacity (number of local computation operations) divided by local I/O capacity (number of blocks of size B that can be transferred between the local disks and memory) per unit time.

In many practical cases, all processors have the *same number* of disks and, thus, the model is restricted to that case (although the model forbids different memory sizes). The disk drives of each processor are denoted by $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. Each drive consists of a sequence of *tracks* which can be accessed by direct random access. A track stores exactly one block of B words. Each processor can use all its D disk drives *concurrently* and transfer $D \times B$ words from/to the local disks to/from its local memory in a single I/O operation being at cost G. In such an operation, only one track per disk is permitted to be accessed *without any restriction* and each track is set on each disk. Note that an operation involving fewer disk drives incurs the same cost. Each processor is

assumed to be able to store in its local main memory at least some blocks from each disk at the same time, i. e., $M >> DB$.

Like computation on the BSP model, the computation of the EM-BSP model proceeds in a succession of supersteps. The communication costs are the same as for the BSP model. The EM-BSP model allows multiple I/O operations during the computation phase of the superstep. The total cost of each superstep is thus defined as $t_{comp,io} + t_{comm} + L$ where $t_{comp,io}$ is the computational cost and additional I/O cost charged for the supersteps, i.e, $t_{comp,io} = \sum_s \max_i(w_i^s + m_i^s)$ where m_i^s is the I/O cost incurred by processor i during superstep s. We refer to [16] to have the EM-BSP complexity of some classical BSP algorithms.

3.2. Examples of EM algorithms. Our first example is the matrix inversion which is used by many applications as a direct method to solve *linear systems*. The computation of the inverse of a matrix A can be derived from its LU factorization. [8] presents the LU factorization by blocks. For this parallel out-of-core factorization, the matrix is divided in blocks of columns called *superblocks*. The width of the superblock is determined by the amount of physical available memory: only blocks of the current superblock are in the main memory, the others are on disks. The algorithm factorize the matrix from left to right, superblock by superblock. Each time a new superblock of the matrix is fetched in the main memory (called the *active* superblock), all previous pivoting and update of a history of the right-looking algorithm are applied to the active superblocks. Once the last superblock is factorized, the matrix is re-read to apply the remaining row pivoting of the recursive phases. Note that the computation is done *data in place*, the matrix has been first distributed on processors and thus, for load balancing, a cyclic distribution of the data is used.

[9] presents PRAM algorithms using external-memory for graph problems as biconnected components of a graph or minimum spanning forest. One of them is the 3-coloring of a cycle applied to finding large independents sets for the problem of list ranking (determine, for each node v of a list, the rank of v define as the number of links from v to the end of the list). The methods for solving it is to update scattered successor and predecessor colors as needed after re-coloring a group of nodes of the list without sorting or scanning the entire list. As before, the algorithms works group by groups with only one group in the main memory.

The last example is the multi-string search problem which consists of determining which of k pattern strings occur in another string. Important applications on biological databases make use of very large text collections requiring specialized nontrivial search operations. [19] describes an algorithm for this problem with a constant number of supersteps and based on the distribution of a proper data structure among the processors and the disks to reduce and balance the communication cost. This data structure is based on a bind tree built on the suffixes of the strings and the algorithm works on longest common prefix on such trees and by lexicographic order. The algorithm takes advantage of disks by only keeping a part of a bind tree in the main memory and by collecting subpart of trees during the supersteps.

4. External Memory in BSML.

4.1. Problems by Adding I/O in BSML. The main problem by adding external memory and so I/O operators to BSML is to keep safe the fact that in the global context, the replicate values, i.e, usual OCaml values replicate on each processor, are the same. Such values are dedicated to the global control of the parallel algorithms. Take for example the following expression:

```
let chan=open_in "file.dat" in
 if (at (mkpar(fun pid→(pid mod 2)=0)) (input_value chan))
  then scan_direct (+) 0 (replicate 1)
 else (replicate 1)
```

It is not true that the file on each processor contains the same value. In this case, each processor reads on its secondary memory a different value. We would have obtained an incoherent result because each processor reads a different integer on the *channel* chan and some of them would execute scan_direct which need a synchronization. Others would execute replicate which does not need a synchronization. This breaks the confluent result of the BSML language and the BSP model of computation with its global synchronizations. If this expression had been evaluated with the BSMLlib library, we would have a breakdown of the BSP computer because **at** is a global synchronous primitive. Note that we also have this kind of problems in the BSPlib [28] where the authors note that only the I/O operations of the first processor are "safe". Another problem comes from *side-effects* that can occur on each processor. Take for example the following expression:

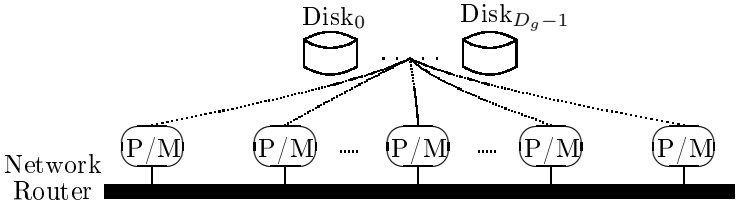


FIG. 4.1. *A BSP computer with shared disks*

```
let a=mkpar(fun i→if i=0 then(open_in "file.dat");()else ())
   in (open_out "file.dat")
```

where () is an empty value. If this expression had been evaluated with the BSMLlib library, only the first processor would have opened the file in a read mode. After, each processor opened the file with the same name in a write mode except the first one. This file has already been opened in read mode. We would also have an incoherent result because the first processor raised an exception which is not caught at all by other processes in the global context. This problem of side-effects could also be combined with the first problem if there is no file at the beginning of the computation. Take for example the following expression:

```
let chan=open_out "file.dat" in
let x=mkpar(fun i→if i=0 then (ouput_value 0) else ()) in
 ouput_value 1; close cha;
 let chan=open_in "file.dat" in
  if (at (mkpar(fun pid→(pid mod 2)=0)) (input_value chan))
   then scan_direct (+) 0 (replicate 1)
  else (replicate 1)
```

The first processor adds the integers 1 and 2 on its file and other processors add the integer 2 on their files. As in the first example, we would have a breakdown of the BSP computer because the integer read would not be the same and **at** is a global synchronous primitive.

4.2. The proposed solution. Our solution is to have two kinds of files: global and local ones. In this way, we have two kinds of I/O operators. Local I/O operators do not have to occur in the global context and global I/O ones do not have to occur locally. Local files are in local file systems which are presents in each processor as in the EM-BSP model. Global files are in a global file system. These files need to be the same from the point of view of each node. The global file system is thus in *shared disks* (as in Figure 4.1) or as a copy in each processor. They thus always give the same values for the global context. Note that if they are only shared disks and not local ones, the local file systems could be in different directories, one per processor in the global file system.

An advantage of having shared disks is the case of some algorithms which do not have distributed data at the beginning of the computation. As those which sort, the list of data to sort is in a global file at the beginning of the program and in another global file at the end. On the other hand, in the case of a distributed global file system, the global data are also distributed and programs are less sensitive to the problem of *faults*. Thus, we have two important cases for the global file system which could be seen as a new parameter of the EM-BSP machine: have we shared disks or not?

In the first case, the condition that the global files are the same for each processor point of view requires some synchronizations for some global I/O operators as created, opened or deleted a file. For example, it is impossible or un-deterministic for a processor to create a file in the global file system if at the same time another processor deleted it. On the other hand, reading (resp. writing) values from (resp. to) files do not need any synchronization. All the processors read the same values in the global file and only one of the processors needs to really write the value on the shared disks. In the case of a global output operator only one of the processors writes the value and in the case of a global input operator the value is first read from the disks by a processor and then is read by other processors from the operating system buffers. In this way, for all global operators, there is not a bottleneck of the shared disks.

In the second case, all the files, local and global ones, are distributed and no synchronization is needed at all. Each processor reads/writes/deletes etc. in its own file system. But at the beginning, the global file system needs to be empty or replicated to each processor and the global and local file systems in different directories.

Note that many modern parallel machines have concurrent shared disks. Such disks are always considered as *user disks*, i.e, disks where the users put the data needed for the computations whereas local disks are only generally used for the parallel computations of programs. For example, the earth simulator has 1,5 Petabytes for users as mass storage disks and a special network to access them. If there are no shared disks, NFS or scalable low level libraries as in [36] are able to simulate concurrent shared disks. Note also that if they are only shared disks, local disks could be simulated by using different directories for the local disks of the processors (one directory for one processor).

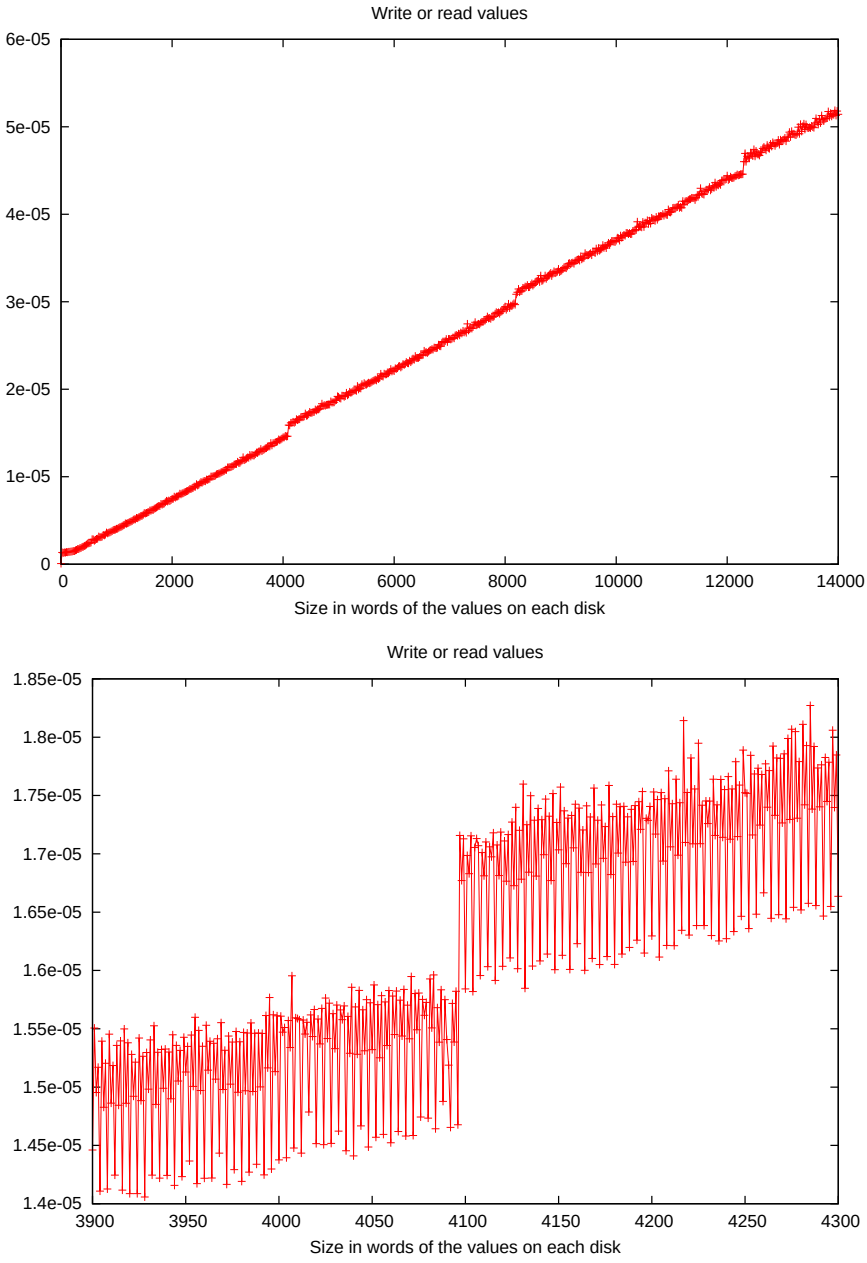


FIG. 4.2. *Benchmarks of EM parameters*

4.3. Our new model. After some experiments to determine the EM-BSP parameters of our parallel machine, we have found that operating systems do not read/write data in a constant time but in a linear time depending on the size of the data. We also notice that there is an overhead depending on the size of the blocks, i. e., if we have $n \times (DB) < s < (n+1) \times DB$, where s is the size in words of the data, there is $n+1$ overheads

to read/write this value from/to the D concurrent disks. Figure 4.2 gives the results of this experiment on a PC with 3 disks, each disk with blocks of 4096 words (seconds are plotted on the vertical axis). This program was run 10000 times and the average was taken. Such results are not altered if we decrease the number of disks.

Our proposed solution gives the processors access to two kinds of files: global and local ones. By this way, our model called **EM^2-BSP** extends the BSP model to its EM^2 version with two kinds of external memories, local and global ones. Each local file system will be on local concurrent disks as in the EM-BSP model. The global one will be on concurrent shared disks (as in Figure 4.1) if they exist or replicate on the local disks. The EM^2-BSP model is thus able to take into account the time to read the data and to distributed them into the processors. The following parameters are thus adding to the standard BSP parameters:

(i) M is the local memory size of each processor;
(ii) D^l is the number of independent disks of each processor;
(iii) B^l is the transfer block size of a local disk;
(iv) G^l is the time to read or write in parallel one word on each local disk;
(v) O^l is the overhead of the concurrent local disks;
(vi) D^g is the number of independent shared disks (or global disks);
(vii) B^g is the transfer block size of a global disk;
(viii) G^g is the time to read or write in parallel one word on each global disk and
(ix) O^g is the overhead of the concurrent global disks.

Of course, if there are no shared disks or no local disks: $D^l = D^g$, $B^l = B^g$, $G^l = G^g$ and $O^l = O^g$. A processor is able to read/write n words to its local disks in time $\lceil \frac{n}{D^l} \rceil \times G^l + \lceil \frac{n+1}{D^l B^l} \rceil \times O^l$ and n words to the global disks in time $\lceil \frac{n}{D^g} \rceil \times G^g + \lceil \frac{n+1}{D^g B^g} \rceil \times O^g$.

As in the EM-BSP model, the computation of the EM^2-BSP model proceeds in a succession of supersteps. The communication costs are the same as for the EM-BSP model and multiple I/O operations are also allowed during the computation phase of a superstep.

Note that G^g is not g even if processors access to the shared disks by the network (in case of some parallel machines): g is the time to perform a 1-relation and G^g is the time to read/write D words on the shared concurrent disks. It could depend on g in some parallel machine as clusters but it could depend on many other hardware parameters if, for example, there is a special network to access to the shared concurrent disks.

4.4. New Primitives. In this section we describe the core of our I/O library, i. e., the minimal set of primitives for programming EM^2-BSP algorithms. This library will be incorporated in the next release of the BSMLlib. This I/O library is based on the elements given in Figure 4.3. As in the BSMLlib library, we have functions to access to the EM^2-BSP parameters of the underlining architecture. For example, **embsp_loc_D**() is D^l the number of local disks and **glo_shared**() gives if the global file system is shared or not. Since we have two file systems, we need two kinds of names and two kinds of abstract types of output channels (resp. input channels): glo_out_channel (resp. glo_in_channel) and loc_out_channel (resp. loc_in_channel) to read/write values from/to global or local files.

We can open a named file for writing. The primitive returns a new output channel on that file. The file is truncated to zero length if it already exists. Either it is created or the primitive will raise an exception if the file could not be opened. For this, we have two kinds of functions for global and local files: (**glo_open_out** F) which opens the global file F in write mode and returns a global channel positioned at the beginning of that file and (**loc_open_out** f) which opens the local file f in write mode and returns a local channel positioned at the beginning of that file. In the same manner, we have two functions, **glo_open_in** and **loc_open_in** for opening a named file in read mode. Such functions return new local or global input channels positioned at the beginning of the files. In the case of global shared disks, a synchronization occurs for each global "**open**". With this global synchronization, each processor could signal to the other ones if it managed to open the file without errors or not and each processor would raise an exception if one of them has failed to open the file.

Now, with our channels, we can read/write values from/to the files. This feature is generally called *persistence*. To write the representation of a structured value of any type to a channel (global or local), we used the following functions: (**glo_output_value** Cha v) which writes the replicate value v to the opened global file and (**loc_output_value** cha v) which locally writes the local value v to the opened local file. The object can be then read back, by the reading functions: (**glo_input_value** Cha) (resp. (**loc_input_value** cha)) which returns from the global channel Cha (resp. local channel cha) the replicate value Some v (resp. local value) or None if there is no more value in the opened global file (resp. local file). This is the end of the file.

EM^2-BSP parameters

embsp_loc_D:unit→int	**embsp_loc_B**:unit→int	**embsp_loc_G**:unit→float
embsp_glo_D:unit→int	**embsp_glo_B**:unit→int	**embsp_glo_G**:unit→float
embsp_loc_O:unit→float	**embsp_glo_O**:unit→float	**glo_shared**:unit→bool

Global I/O primitives	Local I/O primitives
glo_open_out:glo_name→glo_out_channel	**loc_open_out**: loc_name→loc_out_channel
glo_open_in:glo_name→glo_in_channel	**loc_open_in**:loc_name→loc_out_channel
glo_output_value:glo_out_channel→α→unit	**loc_output_value**:loc_out_channel→α→unit
glo_input_value:glo_in_channel→α option	**loc_input_value**:loc_in_channel→α option
glo_close_out:glo_out_channel→unit	**loc_close_out**:loc_out_channel→unit
glo_close_in:glo_in_channel→unit	**loc_close_in**:loc_in_channel→unit
glo_delete:glo_name→unit	**loc_delete**:loc_name→unit
glo_seek:glo_in_channel→int→unit	**loc_seek**:loc_in_channel→int→unit

From local to global

glo_copy:int→loc_name→glo_name→unit

FIG. 4.3. *The Core I/O Bsmllib Library*

Such functions read the representation of a structured value and we refer to [34] about having type safety in channels and reading them in a safe way. We also have (**glo_seek** Cha n) (resp. **loc_seek**) which allows to positioned the channel at the nth value of a global file (resp. local file). The behavior is unspecified if any of the above functions is called with a closed channel.

Note that only local or replicate values could be written on local or global files. Nesting of parallel vectors is prohibited and thus **loc_output_value** could only write local values. It is also impossible to write on a shared global file a parallel vector of values (global values) because these values are different on each processor and **glo_output_value** is an asynchronous primitive. Such values could be written in any order and could be mixed with other values. This is why only local and replicate values should be read/write from/to disks (see section 6 for more details).

After, read/write values from/to channels, we need to close them. As previously, we need four kinds of functions: two for the input channels (local and global ones) and two for the output channels. For example, (**glo_close_out** Cha), closes the global output channel Cha which had been created by **glo_open_out**. The **glo_delete** and **loc_delete** primitives delete a global or a local file if it is first closed.

The last primitive copies a local file from a processor to the global file system. It is thus a global primitive. (**glo_copy** n f F) copies the file f from the processor n to the global file system with the name F. This primitive could be used at the end of a BSML program to copy the local results from local files to the global (user) file system. It is not a communication primitive because used as a communication primitive, **glo_copy** has a more expensive cost than any communication primitive (see section 6). In the case of a distributed global file system, the file is duplicated on all the global file systems of each processor and thus all the data of the file are all put into the network. On the contrary, in the case of global shared disks, it is just a copy of the file because, access to the global shared disks is generally slower than putting values into the network and read them back by another processor.

Using these primitives, the final result of any program would be the same (but naturally without the same total time, i. e., without the same costs) with shared disk or not. Now, to better understand how these new primitives work, we describe a formal semantics of our language with such persistent features.

5. High Order Formal Semantics.

5.1. Mini-BSML. Reasoning on the complete definition of a functional and parallel language such as BSML, would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this section introduces a core language as a mini programming language. It is an attempt to trade between integrating the principal features of persistence, functional, BSP language and being simple. The

expressions of mini-BSML, written e possibly with a prime or subscript, have the following abstract syntax:

e ::=	x	variables	$\mid$ c	constants
$\mid$	**op**	operators	$\mid$ **fun** $x \rightarrow e$	abstraction
$\mid$	$(e\ e)$	application	$\mid$ **let** $x = e$ **in** e	binding
$\mid$	(e, e)	pairs	$\mid$ **if** e **then** e **else** e	conditional
$\mid$	(**mkpar** e)	parallel vector	$\mid$ (**apply** $e\ e$)	parallel application
$\mid$	(**put** e)	communication	$\mid$ (**at** $e\ e$)	projection
$\mid$	f	file names or channels		

In this grammar, x ranges over a countable set of identifiers. The form $(e\ e')$ stands for the application of a function or an operator e, to an argument e'. The form **fun** $x \rightarrow e$ is the so-called and well-known lambda-abstraction that defines the first-class function of which the parameter is x and the result is the value of e. Constants c are the integers, the booleans, the number of processes **p** and we assume having a unique value for the type `unit`: **()**. The set of primitive operations *op* contains arithmetic operations, pair operators, test function **isnc** of the **nc** constructor which plays the role of the None constructor in OCaml, fixpoint to defined natural iteration functions and our I/O operators: $\mathbf{open^r}$ (resp. $\mathbf{open^w}$) to open a file as a channel in read mode (resp. write mode), $\mathbf{close^r}$ (resp. $\mathbf{close^w}$) to close a channel in read mode (resp. write mode), **read**, **write** to read or write in a channel, **delete** to delete a file and **seek** to change the reading position. All those operators are distinguished with a subscript which is l for a local operator and g for a global one. We also have our parallel operators: **mkpar**, **apply**, **put** and **at**. We also have two kinds of file systems, the local and the global ones, defined with (possibly with a prime):

- f for a file name;
- $f_{\mathbf{w}}$ for a write channel, $f_{\mathbf{r}}$ for a read channel and g_k^{ξ} for a channel pointed on the kth value of a file where ξ is the name of the channel;
- ${}^{?}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right]$ is a file where ? is **c**, **r** or **w** for a close file or an open file in read or write mode and where $v_0, \ldots, v_n$ the values hold in the file.

When a file is opened in read mode, it contains the name $[g_n^a, \ldots, g_m^z]$ of the channels that pointed to it and the position of these channels. Before presenting the dynamic semantics of the language, i. e., how the expressions of mini-BSML are computed to *values*, we present the values themselves and the simple ML types [39] of the values. There is one semantics per value of p, the number of processes of the parallel machine. In the following, the expressions are extended with the parallel vectors: $\langle e, \ldots, e \rangle$ (nesting of parallel vectors is prohibited; our static analysis enforces this restriction [23]). The values of mini-BSML are defined by the following grammar:

v ::=	**fun** $x\ \rightarrow e$	functional value	$\mid$ c	constant
	$\mid$ **op**	primitive	$\mid$ (v, v)	pair value
	$\mid$ $\langle v, \ldots, v \rangle$	p-wide parallel vector value	$\mid$ f	file names or channels

and the simple ML types of values are defined by the following grammar:

τ ::=	κ	base type (bool, int, unit, file names or channels)	$\mid$ α	type variables
	$\mid$ $\tau_1 \rightarrow \tau_2$	type of functional values from τ_1 to τ_2	$\mid$ $\tau_1 * \tau_2$	type for pair values

We note $\vdash v : \tau$ to say that the value v has the type τ and we refer to [39] for an introduction to the types of the ML language and to [23] for those of BSML.

5.2. High Order Semantics. The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we used a small-step semantics. It consists of a predicate between an expression and another expression defined by a set of axioms and rules called steps. The small-step semantics describes all the steps of the language from an expression to a value. We suppose that we evaluate only expressions that have been type-checked [23] (nesting of parallel vectors has been prohibited). Unlike in a sequential computer with a sequential programming language, a unique file system (a set of files) for persistent operators is not sufficient. We need to express the file system of all our processors and our global file system. We assume a finite set $\mathcal{N} = \{0, \ldots, p-1\}$ which represents the set of processor names and we write i for these names and $\bowtie$ for the whole parallel computer. Now, we can formalize the files for each processor and for the network. We write $\{f_i\}$ for the file system of processor i with $i \in \mathcal{N}$. We assume that each processor has a file system as an infinite mapping of files which are different at each processor. We write $\{f\} = \{\{f_0\}, \ldots, \{f_{p-1}\}\}$ for all the local file systems of our parallel machine and $\{\mathcal{F}\}$ for our global file

$$
\begin{array}{llll|llll}
(\mathbf{bsp\_p}\ ()) & \xrightarrow[\delta]{} & \mathbf{p} & & (+\ (n_1, n_2)) & \xrightarrow[\delta]{} & n & \text{with } n = n_1 + n_2 \\
(\mathbf{fst}\ (v_1, v_2)) & \xrightarrow[\delta]{} & v_1 & & (\mathbf{snd}\ (v_1, v_2)) & \xrightarrow[\delta]{} & v_2 & \\
\mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2 & \xrightarrow[\delta]{} & e_1 & & \mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2 & \xrightarrow[\delta]{} & e_2 & \\
(\mathbf{isnc\ nc}) & \xrightarrow[\delta]{} & \mathbf{true} & & (\mathbf{isnc}\ v) & \xrightarrow[\delta]{} & \mathbf{false} & \text{if } v \neq \mathbf{nc} \\
(\mathbf{fix\ op}) & \xrightarrow[\delta]{} & \mathbf{op} & & (\mathbf{fix}\ (\mathbf{fun}\ x \to e)) & \xrightarrow[\delta]{} & e[x \leftarrow (\mathbf{fix}\ (\mathbf{fun}\ x \to e))] &
\end{array}
$$

FIG. 5.1. *Functional δ-rules*

system. The persistent version of the small-steps semantics has the following form: $\{\mathcal{F}\}/e/\{f\} \rightharpoonup \{\mathcal{F}'\}/e'/\{f'\}$. We note $\stackrel{*}{\rightharpoonup}$, for the transitive and reflexive closure of $\rightharpoonup$, e. g., we note $\{\mathcal{F}^0\}/e_0/\{f^0\} \stackrel{*}{\rightharpoonup} \{\mathcal{F}\}/v/\{f\}$ for $\{\mathcal{F}^0\}/e_0/\{f^0\} \rightharpoonup \{\mathcal{F}^1\}/e_1/\{f^1\} \rightharpoonup \{\mathcal{F}^2\}/e_2/\{f^2\} \rightharpoonup \ldots \rightharpoonup \{\mathcal{F}\}/v/\{f\}$. To define the relation $\rightharpoonup$, we begin with some rules for two kinds of reductions:

(i) $e/\{f_i\} \stackrel{i}{\rightharpoonup} e'/\{f_i'\}$ which could be read as "with the initial local file system $\{f_i\}$, at processor i, the expression e is reduced to e' with the file system $\{f_i'\}$";

(ii) $\{\mathcal{F}\}/e/\{f\} \stackrel{\bowtie}{\rightharpoonup} \{\mathcal{F}'\}/e'/\{f\}$ which could be read as "with the initial global file system $\{\mathcal{F}\}$ and with the initial set of local file systems, the expression e is reduced to e' with the global file system $\mathcal{F}'$ and with the same set of local file systems".

To define these relations, we begin with some axioms for the functional head reduction $\stackrel{\varepsilon}{\rightharpoonup}$:

$$(\mathbf{fun}\ x \to e)\ v \quad \stackrel{\varepsilon}{\rightharpoonup} \quad e[x \leftarrow v] \qquad \text{and} \qquad \mathbf{let}\ x = v\ \mathbf{in}\ e \quad \stackrel{\varepsilon}{\rightharpoonup} \quad e[x \leftarrow v]$$

We write $e[x \leftarrow v]$ for the expression obtained by substituting all the free occurrences of x in e by v. Free occurrences of a variable are defined as a classical and trivial inductive function on our expressions. This functional head reduction has two versions. First, a local reduction, $\stackrel{\varepsilon}{\underset{i}{\rightharpoonup}}$, of just the processor i and second, a global reduction, $\stackrel{\varepsilon}{\underset{\bowtie}{\rightharpoonup}}$, of the whole parallel machine:

$$\frac{e \stackrel{\varepsilon}{\rightharpoonup} e'}{e\ /\ \{f_i\} \stackrel{\varepsilon}{\underset{i}{\rightharpoonup}} e'\ /\ \{f_i\}}\ (1) \qquad \frac{e \stackrel{\varepsilon}{\rightharpoonup} e'}{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \stackrel{\varepsilon}{\underset{\bowtie}{\rightharpoonup}} \{\mathcal{F}\}\ /\ e'\ /\ \{f\}}\ (2)$$

For primitive operators we also have some axioms, the δ-rules. The functional δ-rules $\underset{\delta}{\rightharpoonup}$ are given in Figure 5.1. First, we have functional δ-rules which could be used by one processor i, $\underset{\delta_i}{\rightharpoonup}$ or by the parallel machine, $\underset{\delta_\bowtie}{\rightharpoonup}$. As in the functional head reduction, we have two different cases for using functional δ-rules:

$$\frac{e \underset{\delta}{\rightharpoonup} e'}{e\ /\ \{f_i\} \underset{\delta_i}{\rightharpoonup} e'\ /\ \{f_i\}}\ (3) \qquad \frac{e \underset{\delta}{\rightharpoonup} e'}{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \underset{\delta_\bowtie}{\rightharpoonup} \{\mathcal{F}\}\ /\ e'\ /\ \{f\}}\ (4)$$

Such reductions, which are not persistent reductions, do not change and do not need the files. Only persistent operators change and need them.

$$
\begin{array}{l}
\{\mathcal{F}\}\ /\ (\mathbf{mkpar}\ v)\ /\ \{f\} \quad \underset{\delta_\diamond}{\rightharpoonup} \quad \{\mathcal{F}\}\ /\ \langle (v\ 0), \ldots, (v\ (p-1)) \rangle\ /\ \{f\} \\
\{\mathcal{F}\}/(\mathbf{apply}\ \langle v_0, \ldots, v_{p-1} \rangle\ \langle v_0', \ldots, v_{p-1}' \rangle)\ /\ \{f\} \quad \underset{\delta_\diamond}{\rightharpoonup} \quad \{\mathcal{F}\}/\langle (v_0\ v_0'), \ldots, (v_{p-1}\ v_{p-1}') \rangle/\{f\} \\
\{\mathcal{F}\}\ /\ (\mathbf{at}\ \langle \ldots, v_n, \ldots \rangle\ n)\ /\ \{f\} \quad \underset{\delta_\diamond}{\rightharpoonup} \quad \{\mathcal{F}\}\ /\ v_n\ /\ \{f\} \ \text{ if } \mathcal{A}_c(v_n) \neq \mathbf{True} \\
\{\mathcal{F}\}\ /\ (\mathbf{put}\ \langle v_0, \ldots, v_{p-1} \rangle)\ /\ \{f\} \underset{\delta_\diamond}{\rightharpoonup} \{\mathcal{F}\}\ /\ (\mathbf{mkfun}\ (\langle \mathbf{send}\ (\mathbf{init}\ v_0\ \mathbf{p}), \ldots, \mathbf{send}\ (\mathbf{init}\ v_{p-1}\ \mathbf{p}) \rangle))\ /\ \{f\} \\
\{\mathcal{F}\}\ /\ \langle \mathbf{send}\ [v_0^0, .., v_0^{p-1}], \ldots, \mathbf{send}\ [v_{p-1}^0, .., v_{p-1}^{p-1}] \rangle\ /\ \{f\} \\
\qquad \underset{\delta_\diamond}{\rightharpoonup} \{\mathcal{F}\}\ /\ \langle [v_0^0, .., v_{p-1}^0], \ldots, [v_0^{p-1}, .., v_{p-1}^{p-1}] \rangle\ /\ \{f\} \text{ if } \forall n, m \in 0, \ldots, p-1\ \mathcal{A}_c(v_n^m) \neq \mathbf{True}
\end{array}
$$

where $\mathbf{mkfun} = \mathbf{apply}\ (\mathbf{mkpar}\ (\mathbf{fun}\ j\ t\ i \to \mathbf{if}\ (\mathbf{and}\ (\leq (0, i), < (i, \mathbf{p})))\ \mathbf{then}\ (\mathbf{access}\ t\ i)\ \mathbf{else}\ \mathbf{nc}))$

FIG. 5.2. *Parallel δ-rules*

Second, for the parallel primitives, we naturally have δ-rules but we do not have those δ-rules on a single processor but only for the parallel machine (Figure 5.2). For simple reasons it is impossible for a processor to send a channel to another processor. This second processor does not have to read in this channel because it could be seen as a hidden communication. In this way, we have to test if the sent values contain channels or not. To do this, we used a trivial inductive function $\mathcal{A}_c$ which tells whether an expression contains a channel or not. Note that this work is done when OCaml serializes values. This raises an exception when an abstract datum like a channel has been found. The evaluation of a **put** primitive proceeds in two steps. In a first step, each processor creates a *pure functional array* of values. Thus, we need a new kind of expression, arrays written $[e, \ldots, e]$. **init** and **access** operators are used to manipulate these functional arrays:

$$\textbf{access}\ [v_0, \ldots, v_n, \ldots, v_m]\ n \ \underset{\delta}{\rightarrow}\ v_n \qquad \text{and} \qquad \textbf{init}\ f\ m \ \underset{\delta}{\rightarrow}\ [(f\ 0), \ldots, (f\ (m\text{–}1))]$$

In the second step, the **send** operations exchange these arrays. For example, the value at the index j of the array held at process i is sent to process j and is stored at index i of the result. The function **mkfun** constructs a parallel vector of functions from the resulting vector of arrays.

$$(\textbf{open}^{\mathbf{r}}\ f)/\{f', \ldots, {}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}, \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad f^a_{\mathbf{r}}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a_0], \ldots, f''\}$$

$$(\textbf{open}^{\mathbf{r}}\ f)/\{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, \ldots, g^z], \ldots, f''\} \ \underset{\delta}{\overset{io}{\rightarrow}}\ f^\xi_{\mathbf{r}}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, \ldots, g^z, g^\xi_0], \ldots, f''\}$$

$$(\textbf{open}^{\mathbf{w}}\ f)/\{f', \ldots, {}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}, \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad f^\xi_{\mathbf{w}}\ /\ \{f', \ldots, {}^{\mathbf{w}}f\boxed{\emptyset}, \ldots, f''\}$$

$$(\textbf{open}^{\mathbf{w}}_l\ f)/\{f', \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad f^\xi_{\mathbf{w}}\ /\ \{f', \ldots, {}^{\mathbf{w}}f\boxed{\emptyset}, \ldots, f''\} \text{ if } f \notin \{f', \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, \ldots, g^\xi_k, \ldots, g^z], \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, \ldots, g^z], \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, \ldots, g^z], \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, \ldots, g^z], \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^\xi_k], \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}, \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{r}}\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}, \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}, \ldots, f''\}$$

$$(\textbf{close}^{\mathbf{w}}\ f^\xi_{\mathbf{w}}/\{f', \ldots, {}^{?}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}, \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad ()\ /\ \{f', \ldots, {}^{\mathbf{c}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}, \ldots, f''\} \text{ where } ? = \mathbf{w} \text{ or } ? = \mathbf{c}$$

$$(\textbf{read}^\tau\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}\vdots\\ v_k\\ \vdots\end{smallmatrix}}[g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\} \ \underset{\delta}{\overset{io}{\rightarrow}}\ v_k\ /\ \{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}\vdots\\ v_k\\ \vdots\end{smallmatrix}}[g^a, .., g^\xi_m, \ldots, g^z], \ldots, f''\}$$
$$\text{if } \vdash v_k : \tau \text{ and } m = k + 1.\ v_k \text{ is the } k\text{th value of } f$$

$$(\textbf{read}^\tau\ f^\xi_{\mathbf{r}})/\{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\} \ \underset{\delta}{\overset{io}{\rightarrow}}\ \mathbf{nc}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\}$$
$$\text{if } k > n$$

$$(\textbf{seek}\ f^\xi_{\mathbf{r}}\ k)/\{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, .., g^\xi_m, \ldots, g^z], \ldots, f''\} \ \underset{\delta}{\overset{io}{\rightarrow}}\ \mathbf{nc}\ /\ \{f', \ldots, {}^{\mathbf{r}}f\boxed{\begin{smallmatrix}v_n\\ \vdots\\ v_0\end{smallmatrix}}[g^a, .., g^\xi_k, \ldots, g^z], \ldots, f''\}$$

$$(\textbf{write}\ (v, f^\xi_{\mathbf{w}}))/\{f', \ldots, {}^{\mathbf{w}}f\boxed{\begin{smallmatrix}\vdots\\ \vdots\end{smallmatrix}}, \ldots, f''\} \ \underset{\delta}{\overset{io}{\rightarrow}}\ ()/\{f', \ldots, {}^{\mathbf{w}}f\boxed{\begin{smallmatrix}v\\ \vdots\end{smallmatrix}}, \ldots, f''\} \text{ if } \mathcal{A}_c(v_n) \neq \mathbf{True}$$

$$(\textbf{delete}\ f)/\{f', \ldots, {}^{\mathbf{c}}f\boxed{\begin{smallmatrix}\vdots\\ \vdots\end{smallmatrix}}, \ldots, f''\} \quad \underset{\delta}{\overset{io}{\rightarrow}} \quad ()/\{f', \ldots, f''\}$$

FIG. 5.3. *δ-rules of the persistent operators*

Third, we complete our semantics by giving the δ-rules $\underset{\delta}{\overset{io}{\rightarrow}}$ of the I/O operators given in Figure 5.3. The **open** operation opens a file (in read or write mode) and returns a new channel, pointing to this file, to access to the values of the file or write values in this file. Opening a file in write mode, gives an empty file. If possible, $\textbf{read}^\tau$ gives the value of type τ contained in the file from the channel. If no more value could be read then $\textbf{read}^\tau$ returns an empty value. The **write** operation writes a new value into the file using the channel. **delete**

$$
\begin{array}{rcl}
\Gamma & ::= & [] \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & \textbf{let}\ x = \Gamma\ \textbf{in}\ e \\
& | & (\Gamma, e) \\
& | & (v, \Gamma) \\
& | & \textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Gamma) \\
& | & (\textbf{apply}\ \Gamma\ e) \\
& | & (\textbf{apply}\ v\ \Gamma) \\
& | & (\textbf{put}\ \Gamma) \\
& | & (\textbf{at}\ \Gamma\ e) \\
& | & (\textbf{at}\ v\ \Gamma)
\end{array}
\qquad
\begin{array}{rcl}
\Gamma_l^i & ::= & \Gamma_l^i\ e \\
& | & v\ \Gamma_l^i \\
& | & \textbf{let}\ x = \Gamma_l^i\ \textbf{in}\ e \\
& | & (\Gamma_l^i, e) \\
& | & (v, \Gamma_l^i) \\
& | & \textbf{if}\ \Gamma_l^i\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{mkpar}\ \Gamma_l^i) \\
& | & (\textbf{apply}\ \Gamma_l^i\ e) \\
& | & (\textbf{apply}\ v\ \Gamma_l^i) \\
& | & (\textbf{put}\ \Gamma_l^i) \\
& | & (\textbf{at}\ \Gamma_l^i\ e) \\
& | & (\textbf{at}\ v\ \Gamma_l^i) \\
& | & \langle e, \ldots, \overbrace{\Gamma_l}^{i}, e, \ldots, e \rangle
\end{array}
\qquad
\begin{array}{rcl}
\Gamma_l & ::= & [] \\
& | & \Gamma_l\ e \\
& | & v\ \Gamma_l \\
& | & \textbf{let}\ x = \Gamma_l\ \textbf{in}\ e \\
& | & \textbf{let rec}\ g\ x = \Gamma_l\ \textbf{in}\ e \\
& | & (\Gamma_l, e) \\
& | & (v, \Gamma_l) \\
& | & \textbf{if}\ \Gamma_l\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & (\textbf{send}\ \Gamma_l) \\
& | & [\Gamma_l, e_1, \ldots, e_n] \\
& | & [v_0, \Gamma_l, \ldots, e_n] \\
& | & \ldots \\
& | & [v_0, v_1, \ldots, \Gamma_l]
\end{array}
$$

FIG. 5.4. *Context of evaluation*

deletes a file from the file system if it has been fully closed. **close** closes a channel or do nothing if the channel has been first closed. All those rules could be distinguished with a subscript (l or g) for the local or the global operators. Thus, we need two kinds of reductions, one for the local reduction $\xrightarrow[\delta_i]{io}$ and another one for the global reduction $\xrightarrow[\delta_{\bowtie}]{io}$:

$$\frac{e\ /\ \{f_i\} \xrightarrow[\delta]{io} e'\ /\ \{f_i'\}}{e\ /\ \{f_i\} \xrightarrow[\delta_i]{io} e'\ /\ \{f_i'\}}\ (5) \qquad \frac{e\ /\ \{\mathcal{F}\} \xrightarrow[\delta]{io} e'\ /\ \{\mathcal{F}'\}}{\{\mathcal{F}\}\ /\ e\ /\ \{f\} \xrightarrow[\delta_{\bowtie}]{io} \{\mathcal{F}'\}\ /\ e'\ /\ \{f\}}\ (6)$$

First, for a single processor i such persistent operations work on the local file system of the processor i where they are executed. Second, for the whole parallel machine, we have the same operations except for the global file system. The special operator $\mathbf{copy}_{\bowtie}$ copies one file of one processor into the global file system:

$$\{F', .., F''\}/(\mathbf{copy}\ i\ f\ \mathcal{F})/\{f_0, \ldots, f_i, \ldots, f_{p-1}\} \xrightarrow[\delta_{\bowtie}]{io} \{F', .., F'', {}^{\mathbf{c}}F\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right]\}/()/\{f_0, \ldots, f_i, \ldots, f_{p-1}\}$$
$$\text{if } F \notin \{F', .., F''\} \text{ and } f_i = \{f', \ldots, {}^{\mathbf{c}}f\left[\begin{smallmatrix} v_n \\ \vdots \\ v_0 \end{smallmatrix}\right], \ldots, f''\}$$

Now, the complete definitions of our two kinds of reductions are:

$$\xrightarrow{i} \ = \ \xrightarrow[i]{\varepsilon} \cup \xrightarrow[\delta_i]{} \cup \xrightarrow[\delta_i]{io} \qquad \text{and} \qquad \xrightarrow{\bowtie} \ = \ \xrightarrow[\bowtie]{\varepsilon} \cup \xrightarrow[\delta_{\bowtie}]{} \cup \xrightarrow[\delta_{\not\bowtie}]{} \cup \xrightarrow[\delta_{\bowtie}]{io}$$

5.3. Contexts of evaluation. It is easy to see that we cannot always make a head reduction. We have to reduce "in depth" in the sub-expressions. To define this deep reduction, we define two kinds of contexts, i.e, expressions with a *hole* noted [] that have the abstract syntax given in Figure 5.4. The hole gives where expressions could be reduced. In this way, the contexts give the order of evaluation of the arguments of the construction of the language, i.e, the strategy of the language.

The Γ context is used to define a global reduction of the parallel machine. For example:

$$\Gamma = \textbf{let}\ x = []\ \textbf{in mkpar}\ (\textbf{fun}\ pid \rightarrow e)$$

The reduction will occur at the hole to first compute the value of x. The Γ_l^i context is used to define in which component i of a parallel vector the reduction is done, i.e., which processor i reduces its local expression. This context uses the Γ_l context which defines a local reduction on a processor i. Note that, in this way, the hole is always inside a parallel vector. For example, the following context: $\Gamma_l^i = \textbf{apply}\ v\ \langle v_0, e_1, \ldots, \Gamma_l \rangle$ and $\Gamma_l = \textbf{open}_l^{\mathbf{r}}\ []$ is used to define that the last processor first computes the argument of the $\textbf{open}_l^{\mathbf{r}}$ primitive.

Now we can reduce "in depth" in the sub-expressions. To define this deep reduction, we use the inference rules of the local context rule:

$$\frac{e\ /\ \{f_i\} \xrightarrow{i} e'\ /\ \{f_i'\}}{\{\mathcal{F}\}\ /\ \Gamma_l^i(e)\ /\ \{f\} \rightharpoonup \{\mathcal{F}\}\ /\ \Gamma_l^i(e')\ /\ \{f'\}} \quad \text{where} \quad \begin{cases} \{f\} = \{\{f_0\}, \ldots, \{f_i\}, \ldots, \{f_{p-1}\}\} \\ \{f'\} = \{\{f_0\}, \ldots, \{f_i'\}, \ldots, \{f_{p-1}\}\} \end{cases} \quad (7)$$

So, we can reduce the parallel vectors and the context gives the name of the processor where the expression is reduced. The global context rule is:

$$\frac{\{\mathcal{F}\} \,/\, e \,/\, \{f\} \overset{\bowtie}{\rightharpoonup} \{\mathcal{F}'\} \,/\, e' \,/\, \{f\}}{\{\mathcal{F}\} \,/\, \Gamma(e) \,/\, \{f\} \rightharpoonup \{\mathcal{F}'\} \,/\, \Gamma(e') \,/\, \{f\}} \quad (8)$$

We can remark that the context gives an order to evaluate an expression but not for the parallel vectors and this rule is not deterministic. It is not a problem because the BSλ-calculus is confluent [37]. We can also notice that our two kinds of contexts used in the rules exclude each other by construction because the hole in a Γ_l^i context is always in a component of a parallel vector and never for a Γ one. Thus, we have a rule to reduce global expressions and another one to reduce usual expressions within the parallel vectors and we have the following result of confluence:

THEOREM 5.1. *if* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^1\}/v_1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^2\}/v_2/\{f^2\}$ *then* $v_1 = v_2$, $\mathcal{F}^1 = \mathcal{F}^2$ *and* $f^1 = f^2$.

Proof. (Sketch of) The BSML language is known to be confluent [37]. With our two kinds of file systems, it is easy to see that a global rule never modifies a local file system and never a local rule modifies the global one. To be more formal, the global (resp. local) files are always the same before and after a local (resp. global) reduction. Thus, the global values are the same on all the processors as proof of confluent of the BSML language needed. All the δ-rules working on files are deterministic (local and global ones). So, the BSML language with parallel I/O features is confluent.□

We refer to appendix 9 for a full proof. Note that the semantics is not deterministic. Several rules can be applied at the same time, parallelism comes from context rules.

6. Formal Cost Model. A formal cost model can be associated to reductions in the BSML language. "cost terms" are defined and each rule of the semantics is associated to a cost rule on cost terms. Given the *weak call-by-value strategy*, i.e., arguments to functions and operators need to be values (see section 5), a program is always reduced in the "same way". As stated in [41], "Each evaluation order has its advantages and disanvatages, but strict evaluation is clearly superior in at least one area: ease of reasoning about asymptotic complexity". In this case, costs can be associated with terms rather than reductions. It is the way we choose to ease the discussion about the compositional nature of the cost model of our language and the cost of our I/O primitives.

6.1. Costs of the Parallel Operators. No order of reduction is given between the different components of a parallel vector and their evaluations are done in parallel. The cost in this case is independent from the order of reduction. We will not describe the costs of the evaluation of local terms, i. e., functional terms. They are the same as those of a strict functional language (OCaml for example) but we give the costs of the evaluation of global and I/O operations.

The cost model associated to our programs follows the EM^2-BSP cost model. We noted $\mathcal{C}(e)$ the cost term associated to an expression, $\mathcal{S}(v)$ the size in words of a serialized value v and $\oplus$ for the sum of cost with the following rules:

$$\begin{array}{rcl} c \oplus \phi c_0, \ldots, c_{p-1} \flat & = & \phi c + c_0, \ldots, c + c_{p-1} \flat \\ c^1 \oplus c^2 & = & c^2 \oplus c^1 \\ \phi c_0^1, \ldots, c_{p-1}^1 \flat \oplus \phi c_0^2, \ldots, c_{p-1}^2 \flat & = & \phi c_0^1 + c_0^2, \ldots, c_{p-1}^1 + c_{p-1}^2 \flat \end{array}$$

where c, c^1 c^2 are cost terms and $\phi c_0, \ldots, c_{p-1} \flat$ is the cost term associated to a parallel vector. Such rules say that the cost of replicate terms could be inside or outside a parallel vector cost term and when we have the cost term of a full-evaluated superstep, this cost could also be inside or outside a parallel vector cost term. This is not a problem because, using the BSP model of computation, at the end of a superstep, we take the maximal of the costs. $+$ and $\times$ are classical cost addition and multiplication using the EM^2-BSP parameters (g, l, r, G^l etc.). We also noted $\biguplus$ for the maximal cost of parallel vector cost terms with this rules: $\biguplus \phi c_0, \ldots, c_n, \ldots, c_{p-1} \flat = c_n$ if c_n is the maximal cost of the component of the parallel vector cost term. We also noted $\bigoplus_{i=0}^{p-1} h_i$ for the maximal of sent/received words. The EM^2-BSP costs of the parallel primitives are given in Figure 6.1. The cost of a program e is thus $\biguplus(\mathcal{C}(e))$ the maximal time for a processor to perform all the supersteps of the program. Let us explain such formal rules with more details and more "readable notations".

If the computational and I/O time for the evalution of the functional parameter e of **mkpar** is w_{all} (it is a replicate function and thus computed by all the processors) and if the sequential evaluation time of each

$$
\begin{array}{lcl}
\mathcal{C}(\textbf{mkpar } e) & \rightsquigarrow & \mathcal{C}(e) \oplus \phi\langle \mathcal{C}((f\ 0)), \ldots, \mathcal{C}((f\ (p-1)))\rangle \quad \text{if } e \stackrel{*}{\rightharpoonup} f \\
\mathcal{C}(\textbf{apply } e_1\ e_2) & \rightsquigarrow & \mathcal{C}(e_1) \oplus \mathcal{C}(e_2) \oplus \phi\langle \mathcal{C}((f_0\ v_0)), \ldots, \mathcal{C}((f_{p-1}\ v_{p-1}))\rangle \\
 & & \qquad \text{if } \left\{ \begin{array}{l} e_1 \stackrel{*}{\rightharpoonup} \langle f_0, \ldots, f_{p-1}\rangle \\ e_2 \stackrel{*}{\rightharpoonup} \langle v_0, \ldots, v_{p-1}\rangle \end{array} \right. \\
\mathcal{C}(\textbf{put } e) & \rightsquigarrow & \biguplus(\mathcal{C}(e) \oplus \phi\langle \sum_{j=0}^{p-1} \mathcal{C}((f_0\ j)), \ldots, \sum_{j=0}^{p-1} \mathcal{C}((f_{p-1}\ j))\rangle) \oplus (\bigoplus_{i=0}^{p-1} h_i) \times g \oplus l \\
 & & \qquad \text{where } \left\{ \begin{array}{l} \text{if } e \stackrel{*}{\rightharpoonup} \langle f_0, \ldots, f_{p-1}\rangle \\ \text{if } \forall i, j \in \{0, \ldots, p-1\}\ (f_i\ j) \stackrel{*}{\rightharpoonup} v_j^i \\ \text{and } h_i = \bigoplus(\sum_{j=0}^{p-1} \mathcal{S}(v_j^i), \sum_{j=0}^{p-1} \mathcal{S}(v_i^j)) \end{array} \right. \\
\mathcal{C}(\textbf{at } e_1\ e_2) & \rightsquigarrow & \biguplus(\mathcal{C}(e_1) \oplus \mathcal{C}(e_2)) \oplus (p-1) \times \mathcal{S}(v_n) \times g \oplus l \\
 & & \qquad \text{if } \left\{ \begin{array}{l} e_2 \stackrel{*}{\rightharpoonup} n \\ e_1 \stackrel{*}{\rightharpoonup} \langle v_0, \ldots, v_n, \ldots, v_{p-1}\rangle \end{array} \right.
\end{array}
$$

FIG. 6.1. *Costs of our parallel operators*

component of the parallel vector is $w_i + m_i$ (computational time and I/O time) then, the parallel evaluation time of the parallel vector is $\phi\langle w_{all} + w_0 + m_0, \ldots, w_{all} + w_{p-1} + m_{p-1}\rangle$, i.e, it is a local computation.

Provided the two arguments of the parallel application are parallel vectors of values, and if w_i (resp. m_i) is the computational time (resp. I/O time) of $(f_i\ v_i)$ at processor i, the parallel evaluation time of (**apply** $\langle f_0, \ldots, f_{p-1}\rangle\ \langle v_0, \ldots, v_{p-1}\rangle$) is $\phi\langle w_{all} + w_0 + m_0, \ldots, w_{all} + w_{p-1} + m_{p-1}\rangle$ where w_{all} is the computational and I/O time to create the two parallel vectors.

The evaluation of **put** $\langle f_0, \ldots, f_{p-1}\rangle$ requires a full superstep. Each processor evaluates the p local terms $(f_i\ j)$, $0 \leq j < p$ leading to p^2 sending values v_i^j (first phase of the superstep). If the value v_i^j of processor i is different from **None**, it is sent to processor j (communication phase of the superstep). Once all values have been exchanged, a synchronization barrier occurs. So, the parallel evaluation time is:

$$\max_{0 \leq i < p} (w_i + m_i + w_{all}) \oplus \max_{0 \leq i < p} (h_i \times g) \oplus l$$

where w_i (resp. m_i) is the computation time (resp. I/O time) of $(f_i\ j)$, h_i is the number of words transmitted (or received) by processor i and w_{all} is the computation time to create the parallel vector $\langle f_0, \ldots, f_{p-1}\rangle$.

The evaluation of a global projection (**at** $\langle v_0, \ldots, v_{p-1}\rangle\ n$) where n is an integer value also requires a full superstep. First the processor n sends the value v_n to all other processors and then a synchronization barrier occurs. The parallel evaluation time is thus the time to send this data, the time for compute n and the maximal local computation and I/O time to create the parallel vector $\langle v_0, \ldots, v_{p-1}\rangle$.

6.2. Cost of I/O operators. Our I/O operators have naturally some computational and I/O costs. We also made sure that arguments of the I/O operators be evaluated first (*weak call-by-value strategy*). As explained in the EM^2-BSP model, each transfer from (resp. to) the local external memory to (resp. from) the main memory has the cost $\lceil \frac{n}{D^l} \rceil \times G^l + \lceil \frac{n+1}{D^l B^l} \rceil \times O^l$ (resp. $\lceil \frac{n}{D^g} \rceil \times G^g + \lceil \frac{n+1}{D^g B^g} \rceil \times O^g$ for the global external memory) for n words. Note that, in the case of an empty file, the value to be read would be an empty value with an empty size. Thus the cost would just be the overhead. In this way, we have the cost of the "operating system I/O calls". Depending on whether the global file system is shared or not, the global I/O operators have different costs and some barrier synchronizations are needed (Figure 6.2).

Local operators are asynchronous operators. They belong to the first phase of a superstep. In the case of a distributed global file system, a global operator has the same cost as a local operator. But, in the case of global shared disks, global operators are synchronous operators because they modify the global behaviour of the EM^2-BSP computer. The two exceptions are **glo_output_value** and **glo_input_value** which are asynchronous global operators because only one process really has to write this replicate value (which is thus the same on each processor) or each processor read this value. The reading of this value could be done in any order. Different channels are positioned at different places in the file but read the same value for the same position. For example, opening a global file needs a synchronization because **glo_output_value** and **glo_input_value**

Operator	Cost
loc_open_in (resp. out)	constant time t^l_{or} (resp. t^l_{ow})
(**loc_output_value** v)	$\lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l$
loc_input_value	$\lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l$ where v is the readed value
loc_close_in (resp. out)	constant time t^l_{cr} (resp. t^l_{cw})
loc_delete	constant time t^l_d
glo_open_in	$\begin{cases} (p-1)\times g \ + \ t^g_{or} \ + \ l & \text{If global file system shared} \\ t^l_{or} & \text{Otherwise} \end{cases}$
glo_open_out	$\begin{cases} (p-1)\times g \ + \ t^g_{or} \ + \ l & \text{If global file system shared} \\ t^l_{ow} & \text{Otherwise} \end{cases}$
(**glo_output_value** v)	$\begin{cases} \lceil\frac{size(v)}{D^g}\rceil \times G^g + \lceil\frac{size(v)+1}{D^g B^g}\rceil \times O^g & \text{If shared} \\ \lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l & \text{Otherwise} \end{cases}$
glo_input_value	$\begin{cases} \lceil\frac{size(v)}{D^g}\rceil \times G^g + \lceil\frac{size(v)+1}{D^g B^g}\rceil \times O^g & \text{If shared} \\ \lceil\frac{size(v)}{D^l}\rceil \times G^l + \lceil\frac{size(v)+1}{D^l B^l}\rceil \times O^l & \text{Otherwise} \\ \text{and where } v \text{ is the readed value} \end{cases}$
glo_close_in	$\begin{cases} (p-1)\times g \ + \ t^g_{cr} \ + \ l & \text{If global file system shared} \\ t^l_{cr} & \text{Otherwise} \end{cases}$
glo_close_out	$\begin{cases} (p-1)\times g \ + \ t^g_{cw} \ + \ l & \text{If global file system shared} \\ t^l_{cw} & \text{Otherwise} \end{cases}$
glo_delete	$\begin{cases} (p-1)\times g \ + \ t^g_{d} \ + \ l & \text{If global file system shared} \\ t^l_{d} & \text{Otherwise} \end{cases}$
(**glo_copy** F f)	$\begin{cases} \lceil\frac{size(F)}{D^g}\rceil \times G^g + \lceil\frac{size(F)}{D^g B^g}\rceil \times O^g + \lceil\frac{size(F)}{D^l}\rceil \times G^l + \lceil\frac{size(F)}{D^l B^l}\rceil \times O^l + l & \text{If global file system shared} \\ (\lceil\frac{size(F)}{D^l}\rceil \times G^l + \lceil\frac{size(F)}{D^l B^l}\rceil \times O^l) \times 2 + size(F) \times g + l & \end{cases}$

FIG. 6.2. *Formal costs of our I/O operators*

are asynchronous operators and a processor could never write in a global file when another reads in this file or opens it in read mode. With this barrier of synchronization, all the processors open (resp. close) the file and they could communicate to each other whether they managed to open (resp. close) that file without errors or not. In this way, $p-1$ booleans are sent on the network and a global exception will be raised if there are any problems.

6.3. Formal Cost Composition. The costs (parallel evaluation time) above are context independents. This is why our cost model is compositional. The compositional nature of this cost model relies on the absence of nesting of parallel vectors (our static analysis enforces this condition [23]) and the fact of having two kinds of file systems. A global I/O operator which accesses a global file and which could make some communications and synchronizations never occurs locally. If the nesting was not forbidden, for a parallel vector v and a **scan** function, the following expression (**mkpar** (**fun** $i \rightarrow$ **if** $i=0$ **then** (**scan** e $(+)$ v) **else** v)) would be a correct one. The main problem is the meaning of this expression.

We said that (**mkpar** f) evaluates to a parallel vector such that processor i holds value (f i). In the case of our example, this means that processor 0 should hold the value of (**scan** e $(+)$ v). Since the semantics of the language is confluent, it is possible to evaluate (**scan** e $(+)$ v) locally. But in this case, processor 0 would not have all the needed values. We could choose that another processors broadcast there own values to processor 0 and then processor 0 evaluates (**scan** e $(+)$ v) locally. The execution time will not follow the formula

given by the above cost model because the broadcasting of these values need additional communications and a synchronization. Thus, we have additional costs which are context dependent. The cost of this expression will then depend on its context. The cost model will no be compositional. This preliminary broadcast is not needed if (**scan** e $(+)$ v) could be not under a **mkpar**. Furthermore, the above solution would imply the use of a scheduler for each processor to know, at every time, if the processor need the values of other processors or not. Such constraints make the cost formulas very difficult to write.

As explained above, if the global file system is shared, only one process has to actually write a value to a global file. In this way, if this value is different on each processor (case of a parallel vector of values) then processors would asynchronously write different values on a shared file and we will not be able to reconstruct this value. The confluence of the language would be lost. In the case of a distributed global file system, this problem does not occur because each processor writes the value on a different file system. Programs would not be portables because they would be architecture dependent. The compositional nature of the cost model is also lost because the final results would depend on the EM^2-BSP architecture and not on the program. This is why it is forbidden to write global values to keep safe the compositional nature of the cost model. Note that the semantics forbids a parallel operator or a parallel persistent operator to be used inside a parallel vector and also forbid a local persistent operator to be used outside a parallel vector.

7. Experiments.

7.1. Implementation. The glo_channel and loc_channel are abstract types and are implemented as arrays of channels, one channel per disk. The current implementation used the thread facilities of OCaml to write (or read) on the D-disks of the computers: we create D-threads which write (or read) on the D channels. Each thread has a part of the data represented as a sequence of bytes and write it in parallel with other threads. To do this, we need to *serialize* our values, i.e., transform our values into a sequence of bytes to be written on a file and decoded back into a data structure. The module Marshal of OCaml provides this feature.

In the case of global shared disks, one of the processors is selected to really write the value, in our first implementation, each of them in turn. To communicate booleans, we used the primitives of communication of BSML. A total exchange of the booleans indicates if the processors has well opened/closed the file or not. The global and the local file systems are in different directories that are parameters of the language. The global directory is supposed to be mounted to access to the shared disks or is in different directories in the case of a distributed global file system. Therefore, global operators accessed to the global directory and local operators accessed to the local directories. In the case of shared disks without local disks, for example, using the library in a sequential machine as a PC, local operators use the "pid" of the processor to distinguish the local files of the different processors.

7.2. Example of functions using our library. Our example is the classical computation of the *prefix* of a list. Here we make the hypothesis that the elements of the list are distributed on all the processes as files which contain sub-parts of the initial list. Each file is cut out on sub-lists with $\frac{D^l \times B^l}{s}$ elements where s is the size of an element. We now describe the algorithm. We first recal the sequential OCaml code part of our algorithm:

```
let isnc=function None→true | _→false

(* seq_scan_last:(α →α →α )→α →α list→α *α list*)
let seq_scan_last op e l =
 let rec seq_scan' last l accu = match l with
            []→(last,(List.rev accu))
     | hd::tl→(let new_last = (op last hd)
                  in seq_scan' new_last tl (new_last::accu))
  in seq_scan' e l []
```

where List.rev $[v_0; v_1; \ldots; v_n] = [v_n; \ldots; v_1; v_0]$. To compute the prefix of a list, we first locally compute the prefix of the local lists located on the local files. For this, we used the following code:

```
(* seq_scan_list_io:(α →α →α )→α →loc_name→loc_name→α  *)
let seq_scan_list_io op e name_in name_tmp=
 let cha_in =loc_open_in name_in in
```

```
 let cha_tmp=loc_open_out name_tmp in
 let rec seq_scan' last =
  let block=(loc_input_value cha_in) in
   if (isnc block) then last
    else let block2=(seq_scan_last op last (noSome block)) in
      loc_output_value cha_tmp (snd block2);
      seq_scan' (fst block2) in
 let res=seq_scan' e in
  loc_close_in cha_in;loc_close_out cha_tmp;res
```

The local file is opened as well as another temporary file. For all the sub-lists of the file, we compute the prefix and the last elements of these prefixes. Then, we write these prefixes to the temporary file and we close the two files. Second, we compute the parallel prefix of the last elements of each prefix of that file. Third, we add those values to the temporary prefixes.

```
(* add_last:(α →α →α )−α →loc_name→loc_name→unit *)
let add_last op e name_tmp name_out =
 let cha_tmp=loc_open_in name_tmp in
 let cha_out=loc_open_out name_out in
 let rec seq_add () =
  let block = (loc_input_value cha_tmp) in
   if (isnc block) then () else
    loc_output_value cha_out(List.map (op e)(noSome block));
    seq_add () in
 seq_add ();loc_close_in cha_tmp;
 loc_close_out cha_out; loc_delete name_tmp
```

The operating of this function is similar to seq_scan_list_io and the full function is thus the composition of the above functions.

```
(*scan:(α →α →α )→α →loc_name→loc_name→loc_name→unit par*)
let scan_list_direct_io op e name_in name_tmp name_out =
 let lasts=parfun (seq_scan_list_io op e name_in)
                                        (replicate name_tmp) in
 let tmp_values=scan_direct op lasts in
  parfun3 (add_last op) tmp_values
                  (replicate name_tmp) (replicate name_out)
```

For example of the use of global files, we give the code of the distribution of the sub-lists to the processors: for each block of the initial list, one processor writes it to its local file.

```
(* distribut:glo_name→loc_name→unit *)
let distribut name_in name_out =
 let cha_in=glo_open_in name_in in
 let cha_outs=parfun loc_open_in (replicate name_out) in
 let rec distri m =
  let block=glo_input_value cha_in in
   if (isnc block) then () else
    (apply2 (mkpar (fun pid→if pid=m then loc_output_value
                                  else (fun a b→()))))
          cha_outs (replicate (noSome block)));
          distri ((m+1) mod (bsp_p())) in
  distri 0;parfun loc_close_out cha_outs;glo_close_in cha_in
```

We have the following cost formula for the I/O scan-list version using a direct scan algorithm:

$$(p-1) \times s \times g + 4 \times N \times (B^l \times G^l + O^l) + 2 \times r \times N \times (D^l \times B^l) + T^1 + l$$

if we read sub-lists of the files by block of size $D^l B^l$ where s denotes the size in words of a element, N is the

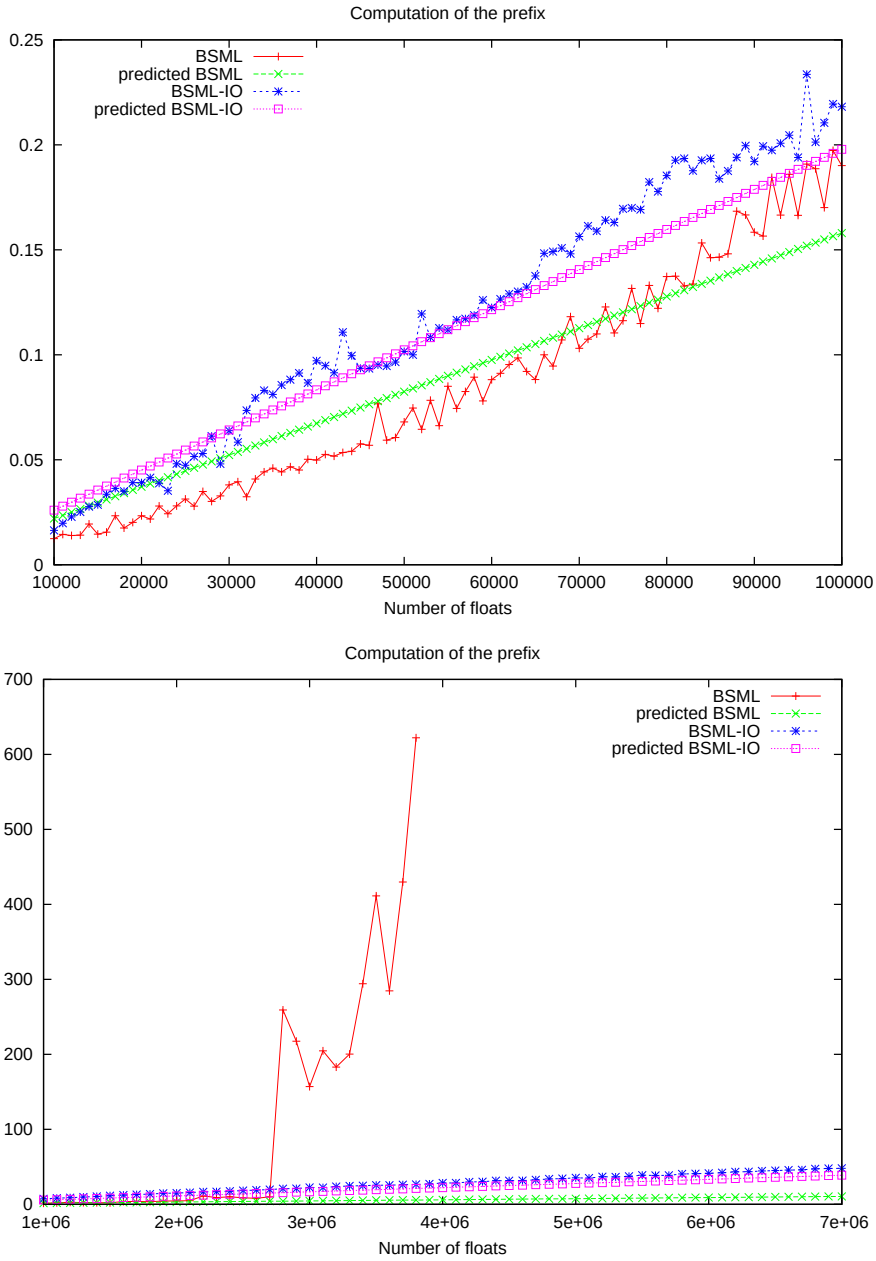


FIG. 7.1. *Benchmarks of prefix computations*

maximal length of a file on a process and where T^1 the time to open and close the files. We have the time to read the local files, write the temporary results of the temporary files, compute the local scan, read the local tempory files and write the final result in the final files. The cost formula of the distribution is:

$$\begin{cases} p \times N \times (\lceil \frac{D^l B^l}{D^g} \rceil \times G^g + \lceil \frac{D^l B^l}{D^g B^g} \rceil \times O^g) + N \times (B^l \times G^l + O^l) + 2 \times l + T^2 \\ \qquad\qquad \text{If shared global file system} \\ p \times N \times (B^l \times G^l + O^g) + N \times (B^l \times G^l + O^l) + T^2 \quad \text{Otherwise} \end{cases}$$

where T^2 the time to open and close the files. We have the time to read the data on the global file (read by block of size $D^l B^l$) and to write them on the local files. We also have two barriers of synchronization due to **glo_open_in** and **glo_close_in**. The cost formula for a global distributed file system is simpler than this

with shared disks but having a distributed file system makes the hypothesis that the global files are replicated on all the processors.

7.3. Benchmarks. Preliminary experiments have been done on a cluster with 6 Pentium IV nodes interconnected with a Gigabit Ethernet network to show a performance comparison between a BSP algorithm using only the BSMLlib and the corresponding EM^2-BSP algorithm using our library. The BSP algorithm reads the data from a global file and keeps them in the main memories. The EM^2-BSP algorithm distributed the data as in the above section. Figure 7.1 summarizes the timings. These programs were run 100 times and the average was taken. Only the local computation has been taken into account because the cluster do not have a true shared disk but a simulated shared disk using NFS. Therefore, the distribution of the data is very slow: G^g depends on g and the distribution of the two different algorithms takes approximately the same time.

The cluster has the following EM^2-BSP parameters:

p	=	6	nodes	D^l	=	1	bytes	D^g	=	1	bytes
r	=	469	Mflops/s	B^l	=	4096	bytes	B^g	=	4096	bytes
g	=	28	flops	G^l	=	1.2	flops	G^g	=	33.33	flops
l	=	22751	flops	O^l	=	100	flops	O^g	=	120	flops

using the MPI implementation of the BSMLlib and with 256 Mbytes of main memory per node. The BSP parameters have been obtained by using the bsmlprobe described in [5] and the I/O parameters have been obtained by using benchmarks as those of the Figure 4.2. The predicted performances using those parameters are also given. We have used floats as elements with $e = 0$, $\mathbf{op} = +$ and we have approximately 140 floats in one block and thus the lists are cut out on sub-lists with 140 elements.

For small lists and thus for a small number of data the overhead for the external memory mapping makes the BSML program outperform the EM^2-BSML one. However, once the main memory is all utilized, the performance of the BSML program degenerates (cost of the paging mechanism to have a virtual memory). The EM^2-BSML program continues "smoothly" and clearly outperforms the BSML code. Note that there is a step between the predictions of the performances and the true performances. This is due to the garbage collector of the OCaml language. In the ML family, the abstract machine manages the resources and the memory, unlike in C or C++ where the programmer has to allocate and de-allocate the data of the memory. Using I/O operators and thus a less naive algorithm achieved a scalability improvement for a big number of data.

8. Related Work. With few exceptions, previous authors focused on a uniprocessor EM model. The *Parallel Disk Model* (PDM) introduced by Vitter and Shriver [54] is used to model a two-level memory hierarchy consisting of D parallel disks connected to $v \geq 1$ processors via a shared memory or a network. The PDM cost measure is the number of I/O operations required by an algorithm where items can be transferred between internal memory and disks in a single I/O operation. While the PDM captures computation and I/O costs, it is designed for a specific type of communication network where a communication operation is expected to take a single unit of time, comparable to a single CPU instruction. BSP and similar parallel models capture communication and computational costs for a more general class of interconnection networks, but do not capture I/O costs. [8] presents an out-of-core parallel algorithm for inversions of big matrices. The algorithm only used broadcasts as primitive of communication with a cost as the BSP cost of a direct broadcast. The I/O costs are similar to ours: linear cost (and not constant cost) to read/write from/to the parallel disks.

Some other parallel functional languages like SAC [25], Eden [31] or GpH [49] offer some I/O features but without any cost model [30]. Parallel EM algorithms need to be carefully hand-crafted to work optimally and correctly in EM environments. I/O operators in SAC have been written for shared disks without formal semantics and the programmer is responsible for underterministic results of such operations. In parallel extensions of the Haskell language (web page `http://haskell.org`) like Eden and Gph, the safety and the confluence of I/O operators are ensured by the use of *monads* [56] and local external memories. Using shared disks is not specified in the semantics of these languages. These parallel languages also authorize processor to exchanged channels and give the possibility to read/write to/from them. It increases the expressiveness of the languages but decreases the cost prediction of the programs. Too many communications are hidden. It also makes the semantics difficult to write [3]. [24] presents a dynamic semantics of a mini functional language with a call-by-value strategy but I/O operators do not work on files. The semantics used a unique input entry (standard input) and a unique output. [18] develops a language for reasoning about concurrent pure functional I/O. They prove that under certain conditions the evaluation of this language is deterministic. But the files are only local files and no formal cost model is given.

In [12] the authors focused on optimization of some parallel EM sort algorithms using cache performances and the several layers of memories of the parallel machines. But they used low level languages and the large number of parameters in this model introduce a hardly tractable complexity. In [15] the authors have implemented some I/O operations to test their models but in a low level language and low level data. In the same manner, [26] describes an I/O library of an EM extension of its cost model which is a special case of the BSP model but also for a low level language. To our knowledge, our library is the first for an extension of the BSP model with I/O features called EM^2-BSP and for a parallel functional language with a formal semantics and a formal cost model. This cost model and our library could be used for large and parallel Data Base as in [2] where the authors used the BSP cost model to balance the communications and the local computations.

9. Conclusions and Future Works. The Bulk-Synchronous Parallel ML allows direct mode BSP programming and the current implementation of BSML is the BSMLlib library. But for some applications where the size of the problem is very significant, external memories are needed. In this paper we have presented an external memory extension of BSP model named EM^2-BSP and a way to extend the BSMLlib for I/O accesses in these external memories. The cost model of these new primitives and a formal semantics as persistent features have been investigated and some benchmarks have been done. This library is the follow-up to our work on imperative features of our functional data-parallel language [22].

There are several possible directions for future works. The first direction is the implementation of persistent primitives using special parallel I/O libraries as described in [29]. For example, low level libraries for shared RAID disks could be used for a fault tolerance implementation of the global I/O primitives.

A complementary direction is the implementation of BSP algorithms [13, 38, 45] and their transformations into EM^2-BSP algorithms as described in [16]. We will design a new library of classical programs as in the BSMLlib library to be used with large computational problems. We also have extended the model to include shared disks. To validate the cost model of these programs, we need a benchmark suite in order to automatically determine the EM parameters. This is ongoing work. We are also working on a result of simulation of a shared external memory as those of the main memory in the BS-PRAM of [48].

A semantic investigation of this framework is another direction of research. To ensure safety and a compositional cost model which allow cost analysis of the programs, two kinds of persistent primitives are needed, global and local ones. Such operators need occur in their context (local or global) and not in another one. We are currently working on a flow analysis [43] of BSML to avoid this problem statically and to forbid nesting of parallel vectors. Static cost analysis as in [51] is also another direction of research.

Acknowledgments The author wishes to thanks the anonymous referees of the Practical Aspects of High-Level Parallel Programming workshop (PAPP 2004), Frédéric Loulergue, Anne Benoï£¡ and Myztzu Modard for their comments.

REFERENCES

[1] *The Coq Proof Assistant (version 8.0)*. Web pages at coq.inria.fr, 2004.
[2] M. Bamha and M. Exbrayat, *Pipelining a Skew-Insensitive Parallel Join Algorithm*, Parallel Processing Letters, 13 (2003), pp. 317–328.
[3] J. Berthold and R. Loogen, *Analysing dynamic channels for topology skeletons in eden*, Tech. Rep. 0408, Institut für Informatik, Lübeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds.
[4] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*, Springer, 2004.
[5] R. Bisseling, *Parallel Scientific Computation. A structured approach using BSP and MPI*, Oxford University Press, 2004.
[6] G.-H. Botorog and H. Kuchen, *Efficient high-level parallel programming*, Theoretical Computer Science, 196 (1998), pp. 71–107.
[7] E. Caron, O. Cozette, D. Lazure, and G. Utard, *Virtual memory management in data parallel applications*, in HPCN Europe, 1999, pp. 1107–1116.
[8] E. Caron and G. Utard, *On the performance of parallel factorization of out-of-core matrices*, Parallel Computing, 30 (2004), pp. 357–375.
[9] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, D. E. Vengroff, and J. S. Vitter, *External-memory Graphs Algorithms*, in ACM-SIAM Symp on Discrete Algorithms, 1995, pp. 139–149.
[10] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner, *Performance issues of the parallel pam-crash code*, Supercomputer Applications and High Performance Computing, 11 (1997), pp. 3–11.
[11] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos, *Randomized External Memory Algorithms for Geometric Problems*, in ACM Annual Conf on Computational Geometry, 1998, pp. 259–268.
[12] C. Cérin and J. Hai, eds., *Parallel I/O for Cluster Computing (Hardback)*, Kojan Page Science, hermes spenton ed., 2002.
[13] F. Dehne, *Special issue on coarse-grained parallel algorithms*, Algorithmica, 14 (1999), pp. 173–421.

[14] F. Dehne, W. Dittrich, and D. Hutchinson, *Efficient external memory algorithms by simulating coarse-grained parallel algorithms*, Algorithmica, 36 (2003), pp. 97–122.
[15] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari, *Parallel virtual memory*, in 10th Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, 1999, pp. 889–890.
[16] ———, *Bulk synchronous parallel algorithms for the external memory model*, Theory of Computing Systems, 35 (2003), pp. 567–598.
[17] W. Dittrich and D. Hutchinson, *Blocking in Parallel Multisearch Problems*, Theory of Computing Systems, 34 (2001), pp. 145–189.
[18] M. Dowse and A. Butterfield, *A language for reasoning about concurrent functional I/O*, Tech. Rep. 0408, Institut für Informatik, Lübeck, September 2004. (IFL'04 workshop), C. Grelck and F. Huch eds.
[19] P. Ferragina and F. Luccio, *String search in coarse-grained parallel computers*, Algorithmica, 24 (1999), pp. 177–194.
[20] F. Gava, *Formal Proofs of Functional BSP Programs*, Parallel Processing Letters, 13 (2003), pp. 365–376.
[21] ———, *Parallel I/O in Bulk Synchronous Parallel ML*, in The International Conference on Computational Science (ICCS 2004), Part III, M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, eds., LNCS, Springer Verlag, 2004, pp. 339–346.
[22] F. Gava and F. Loulergue, *Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features*, in Parallel Computing: Software Technology, Algorithms, Architectures and Applications, Proceeding of the 10th ParCo Conference, G. Joubert, W. Nagel, F. Peters, and W. Walter, eds., Dresden, 2004, North Holland/Elsevier, pp. 95–102.
[23] ———, *A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting*, Future Generation Computer Systems, 21 (2005), pp. 665–671.
[24] A. Gordon and R. L. Crole, *A sound metalogical semantics for input/output effects*, Mathematical Structures in Computer Science, 9 (1999), pp. 125–188.
[25] C. Grelck and S.-B. Scholz, *Classes and objects as basis for I/O in SAC*, in Proceedings of IFL'95, Gothenburg, Sweden, 1995, pp. 30–44.
[26] J. Gustedt, *Towards realistic implementations of external memory algorithms using a coarse grained paradigm*, Tech. Rep. 4719, INRIA, 2003.
[27] G. Hains, *Parallel functional languages should be strict*, in Workshop on General Purpose Parallel Computing. World Computer Congress, B. Perhson and I. Simon, eds., vol. 1, IFIP, North-Holland, September 1994, pp. 527–532.
[28] J. Hill, W. McColl, and al., *BSPlib: The BSP Programming Library*, Parallel Computing, 24 (1998), pp. 1947–1980.
[29] H. Jin, T. Cortes, and R. Buyya, eds., *High Performance Mass Storage and Parallel I/O*, IEEE Press, wiley-interscience ed., 2002.
[30] P. T. K. Hammond and all, *Comparing parallel functional languages: Programming and performance*, Higher-order and Symbolic Computation, 15 (2003).
[31] U. Klusik, Y. Ortega, and R. Pena, *Implementing EDEN: Dreams becomes reality*, in Proceedings of IFL'98, K. Hammond, T. Davie, and C. Clack, eds., vol. 1595 of LNCS, Springer-Verlag, 1999, pp. 103–119.
[32] M. V. Kreveld, J. Nievergelt, T. Roos, and P. W. (editor), *Algorithms Foundations of Geographics Information Systems*, in International Symposium on High Performance Computing, no. 1340 in Lecture Notes in Computer Science, Springer, 1997.
[33] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon, *The Objective Caml System release 3.08*, 2004. web pages at www.ocaml.org.
[34] X. Leroy and M. Mauny, *Dynamics in ML*, Journal of Functional Programming, 3 (1993), pp. 431–463.
[35] Z. Li, P. H. Mills, and J. H. Reif, *Models and resource metrics for parallel and distributed computation*, Parallel Algorithms and Applications, 9 (1995), pp. 35–59.
[36] W. B. Ligon and R. B. Ross, *Beowulf Cluster Computing with Linux*, T. Sterling, mit press ed., November 2001, ch. PVFS: Parallel Virtual File System, pp. 391–430.
[37] F. Loulergue, G. Hains, and C. Foisy, *A Calculus of Functional BSP Programs*, Science of Computer Programming, 37 (2000), pp. 253–277.
[38] W. F. McColl, *Scalability, portability and predictability: The BSP approach to parallel programming*, Future Generation Computer Systems, 12 (1996), pp. 265–272.
[39] R. Milner, *A theory of type polymorphism in programming*, Journal of Computer and System Sciences, 17 (1978), pp. 348–375.
[40] K. Munagala and A. Ranade, *I/O Complexity of Graph Algorithms*, in ACM-SIAM Symposium on Discrete Algorithms, 1999, pp. 687–694.
[41] C. Okasaki, *Purely Functional Data-Structures*, Cambridge University Press, 1998.
[42] S. Pelagatti, *Structured Development of Parallel Programs*, Taylor & Francis, 1998.
[43] F. Pottier and V. Simonet, *Information Flow Inference for ML*, ACM Transactions on Programming Languages and Systems, 25 (2003), pp. 117–158.
[44] J. M. D. Rosario and A. Choudhary, *High performance I/O for massively parallel computers: Problems and prospects*, IEEE Computer, 27 (1994), pp. 59–68.
[45] J. F. Sibeyn and M. Kaufmann, *BSP-Like External-Memory Computation*, in Proc. 3rd Italian Conference on Algorithms and Complexity, vol. 1203 of LNCS, Springer-Verlag, 1997, pp. 229–240.
[46] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, *Questions and Answers about BSP*, Scientific Programming, 6 (1997), pp. 249–274.
[47] R. Thakur, E. Lusk, and W. Gropp, *I/O characterization of a portable astrophysics application on the ibm sp and intel paragon*, Tech. Rep. MCS-P534-0895, Argonne National Laboratory, October 1995.
[48] A. Tiskin, *The bulk-synchronous parallel random access machine*, Theoretical Computer Science, 196 (1998), pp. 109–130.
[49] P. Trinder and all., *GPH: An Architecture-independent Functional Language*, IEEE transactions on Software Engineerig, (1999).
[50] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), p. 103.

[51] P. B. Vasconcelos and K. Hammond, *Inferring cost equations for recursive, polymorphic and higher-order functional programs*, in IFL'02, LNCS, Springer Verlag, 2003, pp. 110–125.
[52] D. E. Vengroff and J. S. Vitter, *Supporting I/O-efficient Scientific Computation in TPIE*, in IEEE Symposium on Parallel and Distributed Computing, 1995.
[53] J. Vitter, *External memory algorithms*, in ACM Symp. Principles of Database Systems, 1998, pp. 119–128.
[54] J. Vitter and E. Shriver, *Algorithms for parallel memory, two -level memories*, Algorithmica, 12 (1994), pp. 110–147.
[55] J. S. Vitter, *External memory algorithms and data structures: Dealing with massive data*, ACM Computing Surveys, 33 (2001), pp. 209–271.
[56] P. Wadler, *Comprehending monads*, Mathematical Structures in Computer Science, 2 (1992), pp. 461–493.

Appendix A. Proof of the confluence.

LEMMA A.1. *If* $e/\{f_i\} \xrightarrow{i} e^1/\{f_i^1\}$ *and* $e/\{f_i\} \xrightarrow{i} e^2/\{f_i^2\}$ *then* $e^1 = e^2$ *and* $\{f_i^1\} = \{f_i^2\}$. *Proof.* By case of the rules of figures 5.1, 5.3 and by construction of rules (1), (3) and (5). □

LEMMA A.2. *If* $\{\mathcal{F}\}/e/\{f\} \overset{\bowtie}{\rightharpoonup} \{\mathcal{F}^1\}/e^1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \overset{\bowtie}{\rightharpoonup} \{\mathcal{F}^2\}/e^2/\{f^2\}$ *then* $e^1 = e^2$, $\{f\} = \{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By case of the rules of figures 5.2, 5.3 and by construction of rules (2), (4) and (6). □

LEMMA A.3. *If* $e = \Gamma_l^i(e_1)$ *then* $\nexists\ e_2$ *as* $e = \Gamma(e_2)$; *If* $e = \Gamma(e_1)$ *then* $\nexists\ e_2$ *as* $e = \Gamma_l^i(e_2)$. *Proof.* By definition, the hole [] is inside a parallel vector in the case of a Γ_l^i context and outside a parallel vector in the other case. By construction, contexts excluded each other. □

DEFINITION A.4. *We noted* $\underset{\bowtie}{\Rightarrow}$ *the reduction* $\rightharpoonup$ *only using the rule (8) and* $\underset{i}{\Rightarrow}$ *the reduction* $\rightharpoonup$ *only using the rule (7).*

LEMMA A.5. *If* $\{\mathcal{F}\}/\Gamma_l^i(e)/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}^1\}/\Gamma_l^i(e^1)/\{f^1\}$ *and* $\{\mathcal{F}\}/\Gamma_l^i(e)/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}^2\}\ /\Gamma_l^i(e^2)/\{f^2\}$ *then* $e^1 = e^2$, $\{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}\} = \{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By application of lemma A.1 and by definition of rule (7). □

LEMMA A.6. *If* $\{\mathcal{F}\}/\Gamma(e)/\{f\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}^1\}/\Gamma(e^1)/\{f^1\}$ *and* $\{\mathcal{F}\}/\Gamma(e)/\{f\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}^2\}/\ \Gamma(e^2)/\{f^2\}$ *then* $e^1 = e^2$, $\{f\} = \{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By application of lemma A.2 and by definition of rule (8). □

DEFINITION A.7. *We noted* $\{\mathcal{F}\}/e/\{f\} \underset{l}{\Rightarrow} \{\mathcal{F}\}/e_1/\{f'\}$ *the reduction* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\underset{i}{\Rightarrow}} \{\mathcal{F}\}/e_1/\{f'\}\ \forall\ i$ *and where* $\nexists\ e_2 \wedge \Gamma_l^i$ *as* $e_1 = \Gamma_l^i(e_2)$ *and where* e_2 *is not a value.*

LEMMA A.8. *If* $\Gamma_l^i(e_1) = \Gamma_l^j(e_2)$ *and* $\{\mathcal{F}\}/\Gamma_l^i(e_1)/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}\}/\Gamma_l^i(e_3)/\{f^3\}$ *and* $\{\mathcal{F}\}/\Gamma_l^j(e_2)/\{f\} \underset{j}{\Rightarrow} \{\mathcal{F}\}/\Gamma_l^j(e_4)/\{f^4\}$ *then* $\exists\ \Gamma'^j_l \wedge \Gamma'^i_l$ *as* $\Gamma_l^i(e_3) = \Gamma'^j_l(e_2)$ *and* $\Gamma_l^j(e_4) = \Gamma'^i_l(e_1)$ *where* $\{\mathcal{F}\}/\Gamma'^j_l(e^2)/\{f^3\} \underset{j}{\Rightarrow} \{\mathcal{F}\}/\Gamma'^j_l(e_5)/\{f^5\}$ *and* $\{\mathcal{F}\}/\Gamma'^i_l(e_1)/\{f^4\} \underset{i}{\Rightarrow} \{\mathcal{F}\}/\ \Gamma'^i_l(e_6)/\{f^6\}$ *and where* $\Gamma'^j_l(e_5) = \Gamma'^i_l(e_6)$ *and* $\{f^5\} = \{f^6\}$.
Proof. It is easy to see that a $\underset{i}{\Rightarrow}$ reduction only modify an expression of the i^{th} component of a parallel vector and the i^{th} file system. Such reduction is determinist by lemma A.5 and thus we have that if two reductions appear in two different components of a parallel vector then such reductions could be done in any order and give the same final result. □

LEMMA A.9. *If* $\{\mathcal{F}\}/e/\{f\} \underset{l}{\Rightarrow} \{\mathcal{F}\}/e_1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \underset{l}{\Rightarrow} \{\mathcal{F}\}/e_2/\{f^2\}$ *then* $e_1 = e_2$ *and* $\{f^1\} = \{f^2\}$.
Proof. By induction on the two reduction $\underset{l}{\Rightarrow}$ and using lemma A.8 to "re-stick" together different paths of the derivations: parallel reductions could be done in any order. □

DEFINITION A.10. $\Rightarrow = \underset{\bowtie}{\Rightarrow} \cup \underset{l}{\Rightarrow}$

LEMMA A.11. *If* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^1\}/v_1/\{f^1\}$ *and* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^2\}/v_2/\{f^2\}$ *then* $v_1 = v_2$, $\{f^1\} = \{f^2\}$ *and* $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$.
Proof. By induction of the derivation and by using lemma A.3: for the two inductive cases, we have the two following cases: if $\{\mathcal{F}'\}/e'/\{f'\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}''\}/e''/\{f'\}$ then the reduction is deterministic by lemma A.6 else $\{\mathcal{F}'\}/e'/\{f'\} \underset{l}{\Rightarrow} \{\mathcal{F}'\}/e''/\{f''\}$ and then the reduction is also deterministic by lemma A.9. □

LEMMA A.12. *If* $\{\mathcal{F}\}/e/\{f\} \underset{i}{\Rightarrow} \{\mathcal{F}\}/e_1/\{f^1\}$ *then* $\{\mathcal{F}\}/e_1/\{f^1\} \overset{*}{\underset{i}{\Rightarrow}} \{\mathcal{F}\}/e_2/\{f^2\}$ *and where* e_2 *is a value*

or $\{\mathcal{F}\}/e_2/\{f^2\} \underset{\bowtie}{\Rightarrow} \{\mathcal{F}'\}/e_3/\{f^2\}$.

Proof. By induction and using lemma A.3 for each steps of the derivation. □

LEMMA A.13. *if* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}'\}/v/\{f'\}$ *then* $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}'\}/v/\{f'\}$. *Proof.* By induction of the derivation. If the rule (8) is used, we are in the case of a global reduction and then we have a $\underset{\bowtie}{\Rightarrow}$ reduction. Else if the rule (7) is used, we are in the case of a local reduction and we have by lemma A.12 that we have a $\underset{l}{\Rightarrow}$ reduction. □

THEOREM A.14. ***confluence of the semantics***

Proof. if $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^1\}/v_1/\{f^1\}$ and $\{\mathcal{F}\}/e/\{f\} \overset{*}{\rightharpoonup} \{\mathcal{F}^2\}/v_2/\{f^2\}$ then $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^1\}/v_1/\{f^1\}$ and $\{\mathcal{F}\}/e/\{f\} \overset{*}{\Rightarrow} \{\mathcal{F}^2\}/v_2/\{f^2\}$ by lemma A.13 and then $v_1 = v_2$, $\{f^1\} = \{f^2\}$ and $\{\mathcal{F}^1\} = \{\mathcal{F}^2\}$ by lemma A.11. □

Edited by: Frédéric Loulergue
Received: June 3, 2004
Accepted: June 5, 2005

Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 71–81. http://www.scpe.org
ISSN 1895-1767

PETRI NETS AS EXECUTABLE SPECIFICATIONS OF HIGH-LEVEL TIMED PARALLEL SYSTEMS

FRANCK POMMEREAU*

Abstract. We propose to use *high-level Petri nets* as a model for the semantics of *high-level parallel systems*. This model is known to be useful for the purpose of verification and we show that it is also *executable* in a parallel way. Executing a Petri net is not difficult in general but more complicated in a *timed context*, which makes necessary to *synchronise* the *internal time* of the Petri net with the *real time* of its environment. Another problem is to relate the execution of a Petri net, which has its own semantics, to that of its environment; *i. e.*, to properly handle *input/output*.

This paper presents a parallel algorithm to execute Petri nets with time, enforcing the even progression of internal time with respect to that of the real time and allowing the exchange of information with the environment. We define a class of Petri nets suitable for a *parallel execution machine* which preserves the *step sequence semantics* of the nets and ensures time consistent executions while taking into account the solicitation of its environment. The question of the efficient verification of such nets has been addressed in a separate paper [14], the present one is more focused on the practical aspects involved in the execution of so modelled systems.

Key words. Petri nets, parallelism, real-time, execution machines.

1. Introduction. *Petri nets* are widely used as a model of *concurrency*, which allows to represent the occurrence of *independent* events. They can be as well a model of *parallelism*, where the *simultaneity* of the events is more important. Indeed, when we consider their *step sequence semantics*, an execution is represented by a sequence of *steps*, each of them being the simultaneous occurrences of some transitions. Within this semantics, the execution of a step may be replaced by that of any of its linearisation (total or partial). This can be viewed as possible executions of the same program on parallel machines with different numbers of processors. In this context, the choice of executing one step or another becomes a question of scheduling (this is usually solved non-deterministically by the Petri net semantics). Petri nets are thus suitable for *specifying* and *verifying* systems in models for which the portability is an important concern.

Our main goal in this paper is to show that Petri nets are also suitable for the *execution* of the modelled systems. We thus consider high-level Petri nets for modelling high-level parallel systems, with the aim to allow both verification and execution of the specification. The question of the efficient verification of such nets has been addressed in a separate paper [14], the present one is more focused on the practical aspects involved in the execution of so modelled systems.

There are at least two reasons for having executable specifications. First, it allows for prototyping and testing at early stage of the design: there may be no need to have an implementation in order to see how the program behaves when its model can already be executed. Second, if the execution of the specification can be made (or happens to be) efficient enough, there is no need to consider any further implementation. This completely saves from the risk of introducing errors on the way from specification to implementation: the verified model and the executed program are exactly the same object. It may be objected that Petri nets are suitable for modelling but really not for programming. This is true. However, Petri nets like those used in this paper are widely used has a semantical domain for parallel programming languages or process algebra with concurrent semantics. For instance, the semantics of the parallel language B(PN)^2 [3] is defined in terms of Petri nets similar to those used in this paper. It features most usually expected high-level constructs for programming languages, in particular: nested declaration of typed variables and FIFO communication channels; communication through shared variables or channels; atomic actions; control flow constructs including parallelism; procedures with parameters passed by value or by reference and allowing recursive and parallel calls [10]; exceptions whose propagation can carry arbitrary value [11]; or Ada-like tasking with suspend/resume or abort capability [12]. Moreover, it can be easily extended with real-time constructs using the same approach to timed system as presented in the following, see [13, § 7.3]. Another example is the *Causal Time Calculus* defined in [14] which is a process algebra with timing features having a step based semantics. Both these formalisms could be applied to massively parallel problems, allowing to leave Petri nets in the background while working with much more pleasant and convenient notations.

*LACL, université Paris 12 — 61, avenue du général de Gaulle — 94010 Créteil, France —pommereau@univ-paris12.fr

Executing a Petri net is not difficult when we consider it alone, *i. e.*, in a *closed world*. But as soon as the net is *embedded in an environment*, the question becomes more complicated. The first problem comes when the net is timed: we have to ensure that its time reference matches that of the environment. The second problem is to allow an exchange of information between the net and its environment. Both these questions are addressed in this paper.

The *causal time* approach is a way to introduce timing features in an otherwise untimed model [7], in particular Petri nets. The idea behind causal time is to *use the expressive power of the model* in order to give an *explicit representation of clocks* in the modelled systems. In the case of high-level Petri nets, it is possible to introduce *counters* and a distinguished *tick transition* whose role is to simultaneously increment them. These counters thus become the timing reference and can be used as clock-watches by the processes as in [15, 6, 13, 14]. It was shown in [6, 14] that the causal time approach is highly relevant since it is simple to put into practice and allows for *efficient verification* through model checking. This paper shows that this approach is also relevant when *concrete execution* are considered. For the purpose of verification, the hypothesis of the closed world is assumed: the Petri net which models a system is considered alone, without any reference to something external to it. The situation differs if we consider the execution of such a Petri net in an *environment* which has its *own time reference*. Indeed, the tick transition of a Petri net may causally depend on the progression of other transitions in the net, which results in the so called *deadline paradox* [7]: "tick is disabled until the system progresses". In a closed world, this statement is logically equivalent to "the system is forced to progress before the next tick", which solves the deadline paradox. But, in the case of an open world, one may wonder how even is the progression of the causal time with respect to that of the *real time*, which is the time imposed by the environment.

Moreover, if the Petri net has to communicate with its environment, one may ask how the net can receive information from the environment and send back appropriate responses. Producing output is rather simple since the net is not disturbed; but reading input (*i. e.*, changing the behaviour of the net in reaction to the changes in the environment) is more difficult and may not be always possible.

In this paper, we define a *parallel execution machine* whose role is to run a Petri net with a tick transition in such a way that the ticks occur evenly with respect to the real time. We show that this can be ensured under reasonable assumptions about the Petri net. The other role of the machine is to allow the communication between the Petri net and the environment and we will identify favourable situations, very easy to obtain in practice, in which the reaction to a message is ensured within a short delay. An important property of our execution machine will be that it will preserve the step sequence semantics of the Petri net: this machine can be seen as an implementation of the Petri net execution rule including additional constraints related to the environment (real time and communication).

In the perspective of direct execution of the modelled systems, it becomes natural to provide parallel executions of the model of a parallel system. So, our goal in proposing a parallel execution machine is more related to a question of consistency than to that of speedup. The question of the speed of our execution machine will thus be intentionally left out of the topics of this paper. However, our definitions will leave enough free space to investigate in this direction and we will come back to this discussion at the end of the paper.

1.1. Execution machines. Defining an execution machine is the usual way to show that an abstract model, defined under assumptions which may be considered as unrealistic, can be used for concrete executions. For instance, the family of synchronous languages (*e. g.*, Esterel [2]), relies on the *synchronous hypothesis* which states that the reaction to a signal is instantaneous. This leads to consider an infinitely fast computer in the abstract model. Several execution machines for these languages have been defined (see, *e. g.*, [1, 5]); in all cases, the solution to remove the synchronous hypothesis makes use of a compilation stage which produces finite automata in which a whole chain of action/reaction is collapsed on a single transition. This allows a correct implementation of the instantaneous reaction assuming a computer *fast enough* with respect to the delays that the environment can observe. However, this breaks the causality relation between events and leads to reject some systems which may be considered on the abstract level but are concretely impossible to implement.

Similar concerns arise in the case of Petri nets with causal time; in particular, we have to reject systems which allow runs of unbounded length between two consecutive ticks. (Such behaviours are often called *Zeno runs*.) Concerning the question of reacting to the solicitation of the environment, it is easy to introduce specific constructs in a Petri net in order to ensure that a signal will be always taken into account very efficiently,

provided that the environment is not "too demanding". This is to say that we will need a computer fast enough with respect to its environment, exactly like for synchronous languages.

1.2. Organisation of the paper. The sequel is organised as follows. The section 2 introduces the basic notions related to Petri nets and their semantics. The section 3 then defines the class of Petri nets we are interested in and gives the assumptions which must be considered in order to allow their real-time execution. The section 4 shows how such nets can be compiled into a form suitable for their execution. Then, the section 5 defines the execution machine itself. We finally conclude in the section 6, introducing discussions about the efficiency of an implementation.

2. Basic definitions about Petri nets. This section briefly introduces the class of Petri nets and the related notions that will be used in the following.

2.1. Multisets. A *multiset* over a set X is a function $\mu : X \to \mathbf{N}$. We denote by $\mathsf{mult}(X)$ the set of all finite multisets μ over X, *i. e.*, such that $\sum_{x\in X} \mu(x) < \infty$. We write $\mu \leq \mu'$ if the domain X of μ is included in that of μ', and if $\mu(x) \leq \mu'(x)$, for all $x \in X$. An element $x \in X$ belongs to μ, denoted $x \in \mu$, if $\mu(x) > 0$. The sum and difference of multisets, and the multiplication by a non-negative integer are respectively denoted by $+$, $-$ and $*$ (the difference is defined only when the second argument is smaller or equal to the first one). A subset of X may be treated as a multiset over X, by identifying it with its characteristic function, and a singleton set can be identified with its sole element. A finite multiset μ over X may be written as $\sum_{x\in X} \mu(x) * x$ or $\sum_{x\in X} \mu(x) * \{x\}$, as well as in extended set notation, *e. g.*, $\{a_1, a_1, a_2\}$ denotes a multiset μ such that $\mu(a_1) = 2$, $\mu(a_2) = 1$ and $\mu(x) = 0$ for all $x \in X \setminus \{a_1, a_2\}$.

2.2. Labelled Petri nets. Let $\mathbb{S}$ be a set of *actions symbols*, $\mathbb{D}$ a finite set of *data values* (or just *values*) and $\mathbb{V}$ a set of *variables*. For $A \subseteq \mathbb{S}$ and $X \subseteq \mathbb{D} \cup \mathbb{V}$, we denote by $A \otimes X$ the set $\{a(x) \mid a \in A, x \in X\}$. Then, we define $\mathbb{A} \stackrel{\text{df}}{=} \mathbb{S} \otimes (\mathbb{D} \cup \mathbb{V})$ as the set of *actions* (with parameters). These four sets are assumed pairwise disjoint.

Definition 2.1. *A* labelled marked Petri net *is a tuple* $N = (S, T, \ell, M)$ *where:*

- S *is a nonempty finite set of* places*;*
- T *is a nonempty finite set of* transitions*, disjoint from* S*;*
- ℓ *defines the* labelling *of places, transitions and* arcs*, i. e., elements of* $(S \times T) \cup (T \times S)$*, as follows:*
  - *for* $s \in S$*, the labelling is* $\ell(s) \subseteq \mathbb{D}$ *which defines the tokens that the place is allowed to carry (often called the* type *of* s*),*
  - *for* $t \in T$*, the labelling is* $\ell(t) \stackrel{\text{df}}{=} \alpha(t)\gamma(t)$ *where* $\alpha(t) \in \mathbb{A}$ *and* $\gamma(t)$ *is a boolean expression called the* guard of t*,*
  - *for* $(x, y) \in (S \times T) \cup (T \times S)$*, the labelling is* $\ell(x, y) \in \mathsf{mult}(\mathbb{D} \cup \mathbb{V})$ *which denotes the tokens flowing on the arc during the execution of the attached transition. The empty multiset* $\emptyset$ *denotes the absence of arc;*
- M *is a* marking *function which associates to each place* $s \in S$ *a multiset in* $\mathsf{mult}(\ell(s))$ *representing the tokens held by* s*.*

Notice that $\alpha(t)$ could be a finite multiset of actions. This would be a trivial extension but would lead to more complicated definitions; we choose to restrict ourselves to single actions in order to streamline the presentation.

We adopt the standard rules about representing Petri nets as directed graphs with the following simplifications: the names of some nodes (especially places) may not be given; the two components of transition labels are depicted separately; true guards are omitted as well as brackets around sets; arcs may be labelled by expressions as a shorthand (see the example given in the figure 2.1).

Fig. 2.1. *On the left, a Petri net which actually denotes that given on the right, with* $\eta \geq 0$*,* $\{0, \ldots, \eta\} \subseteq \mathbb{D}$*,* $\{x, y\} \subseteq \mathbb{V}$ *and* $\tau \in \mathbb{S}$*.*

2.3. Step sequence semantics. A *binding* is a function $\sigma : \mathbb{V} \to \mathbb{D}$ which associates concrete values to the variables appearing in a transition and its arcs. We denote by $\sigma(E)$ the evaluation of the expression E bound by σ.

Let (S, T, ℓ, M) be a Petri net, and $t \in T$ one of its transitions. A binding σ is *enabling* for t at M if the guard evaluates to true, *i. e.*, $\sigma(\gamma(t)) = \top$, and if the evaluation of the annotations on the adjacent arcs respects the types of the places, *i. e.*, for all $s \in S$, $\sigma(\ell(s,t)) \in \mathsf{mult}(\ell(s))$ and $\sigma(\ell(t,s)) \in \mathsf{mult}(\ell(s))$.

A *step* corresponds to the simultaneous execution of some transitions, it is a multiset

$$U = \{(t_1, \sigma_1), \ldots, (t_k, \sigma_k)\}$$

such that $t_i \in T$ and σ_i is an enabling binding of t_i, for $1 \leq i \leq k$. U is *enabled* if the marking is sufficient to allow the flow of tokens required by the execution of the step, *i. e.*, for all $s \in S$

$$M(s) \;\geq \sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\ell(s,t)).$$

It is worth noting that if a step U is enabled at a marking, then so is any sub-step $U' \leq U$. A step U enabled by M may be *executed*, leading to the new marking M' defined for all $s \in S$ by

$$M'(s) \stackrel{\text{df}}{=} M(s) \;- \sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\ell(s,t)) \;+ \sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\ell(t,s)).$$

This is denoted by $M[U\rangle M'$ and this notation naturally extends to sequences of steps. The empty step, denoted by $\emptyset$, is always enabled and we have $M[\emptyset\rangle M$. A marking M' is *reachable* from a marking M if their exists a sequence of steps ω such that $M[\omega\rangle M'$; we will say in this case that M enables ω. Notice that M is reachable from itself through a sequence of empty steps.

The *step sequence semantics* is defined as the set containing all the sequences of steps enabled by a net. This semantics is based on transitions identities but the relevant information is generally the labels of the executed transitions. The *labelled step* associated to a step U is defined as $\sum_{(t,\sigma)\in U} U((t,\sigma)) * \sigma(\alpha(t))$, which allows to naturally define the *labelled step sequence semantics* of a Petri net. In the sequel we will consider only this semantics and omit the word "labelled".

2.4. Safety. A Petri net (S, T, ℓ, M) is *safe* if any marking M' reachable from M is such that, for all $s \in S$ and all $d \in \ell(s)$, $M'(s)(d) \leq 1$, *i. e.*, any place holds at most one token of each value. The class of safe Petri nets (including models abbreviating them) is very interesting:

- from a theoretical point of view, safe Petri nets never have auto-concurrency of transitions, which allows for efficient verification techniques [8];
- from a pragmatical point of view, safe Petri nets corresponds to the class of finite state Petri nets (as shown in [4], bounded Petri nets can be reduced to safe Petri nets while preserving their concurrent semantics), which correspond to realistic systems, *i. e.*, those that can be implemented on a concrete computer;
- from a practical point of view, this class was shown expressive enough to model most interesting problems from the real world. For instance, the semantics procedures, exceptions or tasks preemption in the language B(PN)² do not require more than safe Petri net.

Another nice property of safe Petri nets, directly related to our purpose, is that they have finitely many reachable markings, each of which enabling finitely many steps whose sizes are bounded by the number of transitions in the net. For all these reasons, as in the previous works about causal time [15, 6, 13, 14], we restrict ourselves to safe Petri nets.

3. Petri nets with causal time: CT-nets. We are now in position to define the class of Petri nets we are actually interested in; it consists in safe Petri nets, with several restrictions, for which we will define some specific vocabulary related to the occurrence of ticks. We assume that there exists $\tau \in \mathbb{S}$, used in the labelling of the tick transition.

Definition 3.1. *A* Petri net with causal time *(*CT-net*) is a safe labelled Petri net* $N \stackrel{\text{df}}{=} (S, T, \ell, M)$ *in which there exists a unique* $t_\tau \in T$*, called the* tick transition *of* N*, such that:*

- $\alpha(t_\tau) \in \{\tau\} \otimes (\mathbb{D} \cup \mathbb{V})$*;*

- $\alpha(t) \notin \{\tau\} \otimes (\mathbb{D} \cup \mathbb{V})$ *for all* $t \in T \setminus \{t_\tau\}$;
- t_τ *has at least one incoming arc labelled by a singleton.*

A tick-step *is a step* U *of* N *which involves the tick transition, i. e., such that* $\tau(d) \in U$ *for a* $d \in \mathbb{D}$.

Thanks to the safety and the last restriction on t_τ, any tick-step contains exactly one occurrence of the tick transition. On the other hand, one may notice that this definition is very liberal and allows to define nets in which the tick transition is not tight to increment counters but may produce any other effect not related to time. Fortunately, we do not need a more restrictive definition, which lets us free to experiment different approaches in the future.

The figure 3.1 shows a toy CT-net that will be used as a running example. In this net, the role of the tick transition t_τ is to increment a counter located in the top-right place. When the transition t_1 is executed, it resets this counter and picks in the top-left place a value which is bound to the variable m. This value is transmitted to the transition t_2 which will be allowed to execute when at least m ticks will have occurred. Thus, m specifies the minimum number of ticks between the execution of t_1 and that of t_2. At any time, the transition t_3 may randomly change the value of this minimum while emitting a visible action $u(x)$ where x is the new value. Notice that the maximum number of ticks between the execution of t_1 and that of t_2 is enforced by the type of the place connected to t_τ which specifies that only tokens in $\{0, \ldots, \eta\}$ are allowed (given $\eta > 0$).

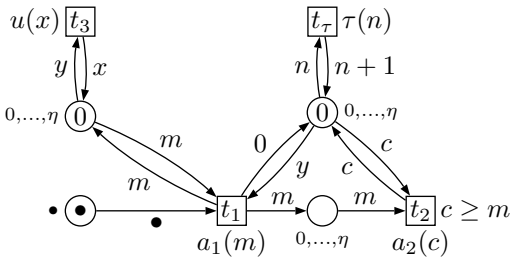


FIG. 3.1. *An example of a CT-net, where* $\eta > 0$, $\{a_1, a_2, u, \tau\} \subseteq \mathbb{S}$, $\{c, n, m, x, y\} \subseteq \mathbb{V}$ *and* $\{0, \ldots, \eta\} \cup \{\bullet\} \subseteq \mathbb{D}$.

Assuming $\eta \geq 5$, a possible execution of this CT-net is:

$$\{\tau(0)\}\ \{u(2)\}\ \{a_1(2)\}\ \{\tau(0), u(1)\}\ \{\tau(1)\}\ \{u(5)\}\ \{\tau(2)\}\ \{\tau(3)\}\ \{a_2(4), u(0)\}\ \{\tau(4)\}\ .$$

3.1. Tractability. A CT-net (S, T, ℓ, M) is *tractable* if there exists an integer $\delta \geq 2$ such that, for all marking M' reachable from M, any sequence of at least δ nonempty steps enabled by M' contains at least two tick-steps. In other words, the length of an execution between two consecutive ticks is bounded by δ whose smallest possible value is called the *maximal distance between ticks*.

This notion of tractable nets is important because it allows to distinguish those nets which can be executed on a realistic machine: indeed, an intractable net may have potentially infinite runs between two ticks (so called *Zeno runs*), which cannot be executed on a finitely fast computer without breaking the evenness of ticks occurrences.

For example, the CT-net of our running example is intractable because the transition t_3 can be executed infinitely often between two ticks: in the execution given above, the step $\{u(5)\}$ could be repeated an arbitrary number of times. In the rest of this paper, we restrict ourselves to tractable CT-nets.

3.2. Input and output. The communication between a CT-net and its environment is modelled using some of the actions in transitions labels. We distinguish for this purpose two finite disjoint subsets of $\mathbb{S}$: $\mathbb{S}_i$ is the set of *input action symbols* and $\mathbb{S}_o$ is that of *output actions symbols*. We assume that $\tau \notin \mathbb{S}_i \cup \mathbb{S}_o$. We also distinguish a nonempty set $\mathbb{D}_{io} \subseteq \mathbb{D}$ representing the values allowed for input and output. Intuitively, the distinguished symbols correspond to communication ports on which values from $\mathbb{D}_{io}$ may be exchanged between the execution machine and its environment. Thus the execution of a transition labelled by $a_o(d_o) \in \mathbb{S}_o \otimes \mathbb{D}_{io}$ is seen as the sending of the value d_o on the output port a_o. Conversely, if the environment sends a value $d_i \in \mathbb{D}_{io}$ on the input port $a_i \in \mathbb{S}_i$, the net is expected to execute a step containing the action $a_i(d_i)$. In general, we cannot ensure that such a step is enabled, in the worst case, it may happen that no transition has a_i in its label. Fortunately, we show now that a net can easily be designed in order to ensure that such an input message is always correctly handled.

A naive way to achieve this result is to use self-loops, like the transition t_3 in the figure 3.1. In this example, if we assume $u \in \mathbb{S}_i$ and $\{0, \ldots, \eta\} \supseteq \mathbb{D}_{io}$, any requested communication on u can always be handled. Unfortunately, self-loops lead to intractable nets since such transitions can always be arbitrarily repeated (remember the step $\{u(5)\}$ above). Actually, a self-loop indicates that the CT-net is expected to be able to respond instantaneously to all the messages that the environment would send on the corresponding port, which is not a realistic assumption. Indeed, if the number of such messages sent in a given amount of real time is not bounded, then a finitely fast computer cannot avoid to miss some of them. So, in the following, we assume that the environment may not produce more than one message on each input port between two ticks, which will lead to the notion of tick-reactiveness. This assumption is equivalent to say that we require the CT-net to be executed on a computer fast enough with respect to its environment; so, this is actually one of the classical conditions that must be assumed while defining an execution machine.

Let $A \subseteq \mathbb{S}_i$ be a nonempty set of input action symbols, we denote by $\mathsf{req}(A)$ the set of potential requests on A, which contains all the sets of the form $\{a_1(d_1), \ldots, a_k(d_k)\}$ where $\{a_1, \ldots, a_k\} \subseteq A$ and $(d_1, \ldots, d_k) \in {\mathbb{D}_{io}}^k$ for all $k \geq 1$. Each element of $\mathsf{req}(A)$ is potentially a step of a CT-net.

A CT-net (S, T, ℓ, M) is *once-reactive* to $A \subseteq \mathbb{S}_i$ iff: either, it enables only the empty step; or, there exists a step $U' \notin \mathsf{req}(A)$ such that $M[U'\rangle M''$ and, for all $U \in \mathsf{req}(A)$, we have $M[U\rangle M'$ and the CT-net (S, T, ℓ, M') is once-reactive to $A \setminus \{a \in A \mid \exists d \in \mathbb{D}_{io}, a(d) \in U\}$. Intuitively, this inductive definition states that, for all input port $a_i \in \mathbb{S}_i$, the CT-net can react to any request on a as soon as it comes, after what it may miss them. On the other hand, the CT-net is never forced to execute an action involving an input port in A (thanks to the step U'). At any time, the CT-net may terminate its execution with a deadlock.

A CT-net (S, T, ℓ, M) is *tick-reactive* to $A \subseteq \mathbb{S}_i$ iff it is once-reactive to A and, for all sequence of steps $U_1 \cdots U_k$ such that U_k is a tick-step and $M[U_1 \cdots U_k\rangle M'$, then the CT-net (S, T, ℓ, M') is tick-reactive to A. This definition is also inductive and states that a tick-reactive CT-net is almost like a once-reactive net except that its capability to react is fully restored after each tick. This guarantees that one message on a may always be handled between two ticks, which exactly matches our assumption. It turns out that it is easy to transform a reactive CT-net with self-loops into a tick-reactive one. It is enough to add one place for each self-loop with the type $\{\circ, \bullet\}$ and marked with $\bullet$, and arcs such that each occurrence of the self-loop consumes the $\bullet$ and replace it with a $\circ$, so it cannot occur twice; on the other hand, each occurrence of the tick-transition must reset to $\bullet$ the token in the added places. This way, self-loops cannot be repeated with at least one tick in between. As we can see, it is easy to construct a tick-reactive net; for instance, the figure 3.2 shows a modified version of our running example which is tick-reactive to $\{u\}$ and tractable (now, the step $\{u(5)\}$ could not be repeated at will).

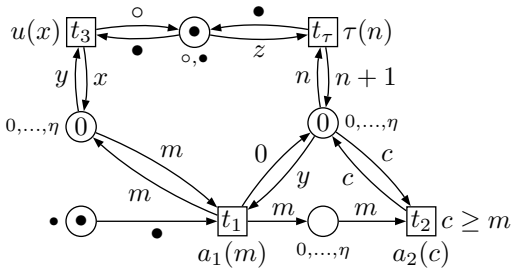


FIG. 3.2. *The tick-reactive version of the running example, where* $z \in \mathbb{V}$ *and* $\{\circ, \bullet\} \subset \mathbb{D}$.

3.3. Consistency. We denote by $U[a]$ the number of occurrences of the action symbol a in a step U, *i. e.*, $U[a] \stackrel{\text{df}}{=} \sum_{a(x) \in U} U(a(x))$. A step U is *consistent* if $U[a] \leq 1$ for all $a \in \mathbb{S}_i \cup \mathbb{S}_o$. A CT-net is *consistent* if its step sequence semantics only involve consistent steps. Inconsistent steps are those during the execution of which several communications take place on the same port. Since the transitions executed by a single step occur simultaneously, this means that several values may be sent or received on the same port at the same time. This is certainly something which is not realistic and so, we restrict ourselves to consistent CT-nets in the following.

The nets given in the figures 3.1 and 3.2 are both consistent. But, assuming $a_2 \in \mathbb{S}_i \cup \mathbb{S}_o$, it would not be the case if we would replace $u(x)$ by $a_2(x)$ in the label of the transition t_3 since we could have and execution with the step $\{a_2(4), a_2(0)\}$ which is not consistent.

4. Compilation of CT-nets: CT-automata. The aim of this section is to show how to transform a tractable and consistent CT-net into a form more suitable for the execution machine. This corresponds to a compilation producing an automaton (non-deterministic in general), called a *CT-automaton*, whose states are the reachable markings of the net and whose transitions correspond to the steps allowing to reach one marking from another. It should be remarked that this compilation is not strictly required but allows to simplify things a lot, in particular in an implementation of the machine: with respect to its corresponding CT-net, a CT-automaton has no notion of markings, bindings, enabling, etc., which results in a much simpler model. Another reason to introduce this compilation stage is that it can be used to check if the net of interest is really a safe, tractable and consistent CT-net; moreover, it is an almost necessary step to compute the value of δ (the maximal distance between ticks) which will be used during the execution. So, as we cannot avoid a computation at least equivalent to this compilation stage, we turn it into an advantage for the execution which can be made much simpler and more efficient.

In order to record only the input and output actions in a step U of a CT-net, we define the set of the *visible actions in* U by $\lfloor U \rfloor \stackrel{\text{df}}{=} U \cap (((\mathbb{S}_i \cup \mathbb{S}_o) \otimes \mathbb{D}_{io}) \cup (\{\tau\} \otimes \mathbb{D}))$. Because of the consistency, $\lfloor U \rfloor$ could not be a multiset.

DEFINITION 4.1. *Let $N = (S, T, \ell, M)$ be a tractable and consistent CT-net, the* CT-automaton of N *is the finite automaton $\mathcal{A}(N) \stackrel{\text{df}}{=} (S_{\mathcal{A}}, T_{\mathcal{A}}, s_{\mathcal{A}})$ where:*

- *$S_{\mathcal{A}}$ is the set of states defined as the set of all the reachable markings of N;*
- *the set of transitions is $T_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times L_{\mathcal{A}} \times S_{\mathcal{A}}$, where $L_{\mathcal{A}} \stackrel{\text{df}}{=} \{A \subseteq ((\mathbb{S}_i \cup \mathbb{S}_o) \otimes \mathbb{D}_{io}) \cup (\{\tau\} \otimes \mathbb{D})\}$, and is defined as the set of all the triples (M', A, M'') such that $M', M'' \in S_{\mathcal{A}}$ and there exists a nonempty step U of N such that $M[U\rangle M'$ and $A = \lfloor U \rfloor$;*
- *$s_{\mathcal{A}} \stackrel{\text{df}}{=} M \in S_{\mathcal{A}}$ is the initial state of $\mathcal{A}(N)$, i. e., the initial marking of N.*

The following holds by definition but should be stressed since it states that a CT-net and the corresponding CT-automaton have exactly the same executions.

PROPOSITION 4.2. *Let $N \stackrel{\text{df}}{=} (S, T, \ell, M)$ be a tractable and consistent CT-net, M' be a reachable marking of N and $(S_{\mathcal{A}}, T_{\mathcal{A}}, M) \stackrel{\text{df}}{=} \mathcal{A}(N)$.*

1. *If $M'[U\rangle M''$ for a nonempty step U then $(M', \lfloor U \rfloor, M'') \in T_{\mathcal{A}}$.*
2. *Conversely, if $(M'', A, M''') \in T_{\mathcal{A}}$ then there exists a nonempty step U such that $M''[U\rangle M'''$ and $\lfloor U \rfloor = A$.*

As an example, the figure 4.1 shows the CT-automaton which corresponds to the tractable version of our running example (given in the figure 3.2). For the sake of compactness, we assumed $\eta \stackrel{\text{df}}{=} 1$ (the automaton for $\eta = 2$ has 105 states and this number grows to 277 for $\eta = 3$). Moreover, we assumed $\{a_1, a_2, u\} \subseteq \mathbb{S}_i \cup \mathbb{S}_o$.

5. The execution machine. We now describe the execution machine. In order to communicate with the environment, a symbol $a_o \in \mathbb{S}_o$ is considered as a port on which a value $d \in \mathbb{D}_{io}$ may be written, which is denoted by $a \leftarrow d$ (more generally, this is used for any assignment). Similarly, a symbol $a_i \in \mathbb{S}_i$ is considered as a port on which such a value, denoted by $a_i?$, may be read; we assume that $a_i? = \varnothing \notin \mathbb{D}_{io}$ when no communication is requested on a_i. Moreover, in order to indicate to the environment if a communication have been properly handled, we also assume that each $a \in \mathbb{S}_i$ may be marked "accepted" (denoting that the communication has been correctly handled), "refused" (denoting that the communication could not been handled), "erroneous" (denoting that a communication on this port was possible but with another value, or that a communication was expected but not requested) or not marked, which is represented by "no mark". We also use the notation $a_i \leftarrow \mathit{mark}$ when an input port is being marked.

Let $(S_{\mathcal{A}}, T_{\mathcal{A}}, s_{\mathcal{A}})$ be a CT-automaton and let Δ be a constant amount of time; we will see later on how Δ is defined since it depends on the definition of the execution machine. We will use three variables:

- Θ is a time corresponding to the occurrences of ticks;
- $s \in S_{\mathcal{A}}$ is the current state;
- $I \subseteq \mathbb{S}_i$ is the set of ports on which the environment asks a communication.

The behaviour of the machine is described by the algorithm given on the left of the figure 4.2 where the execution of a step (line 13) is detailed on the right of the figure. Several aspects of this algorithm should be commented:

- the statement "**now**" evaluates to the current time when it is executed;
- the "**for all**" loops are parallel loops;
- the execution of the line 8 can be parallelised also (see below);

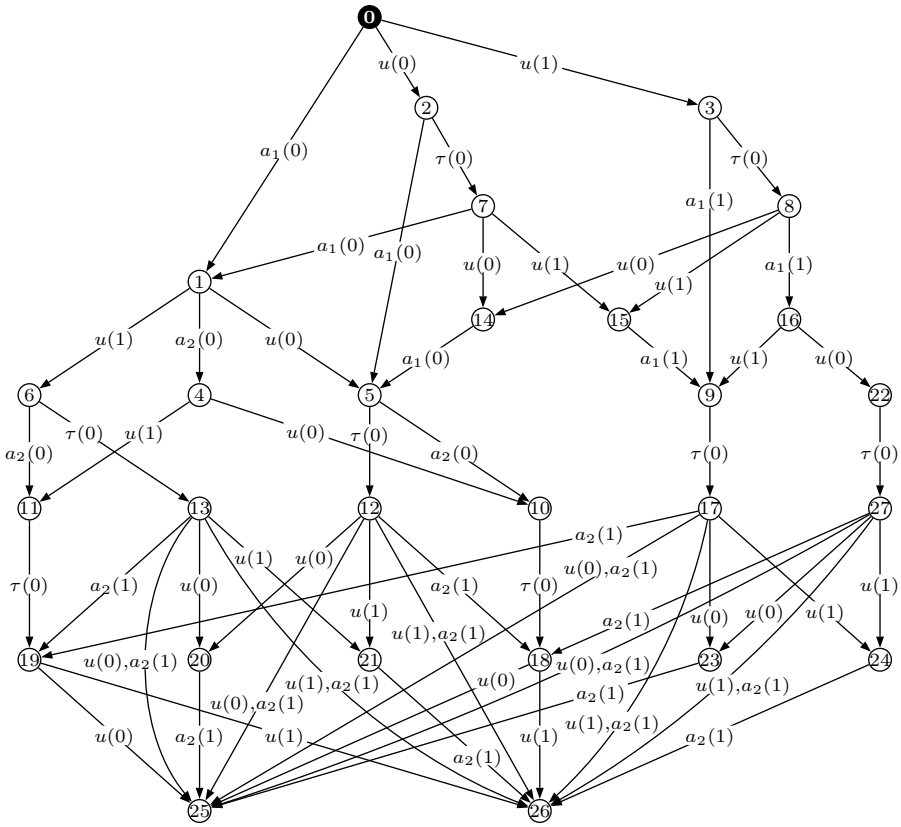


FIG. 4.1. *The CT-automaton of the CT-net given in the figure 3.2 (with* $\eta \stackrel{\text{df}}{=} 1$*), the initial state is numbered 0 and filled in black.*

```
1:  s ← s_A
2:  Θ ← now
3:  while s has successors do
4:     for all a ∈ 𝕊_i do
5:        a ← "no mark"
6:     end for
7:     I ← {a ∈ 𝕊_i | a? ≠ ∅}
8:     choose a transition (s, A, s')
9:     if A is a tick-step then
10:       wait until now = Θ + Δ
11:       Θ ← now
12:    end if
13:    execute(A, I)
14:    s ← s'
15: end while
```

```
procedure execute(A, I) :
17: for all a(d) ∈ A (a ≠ τ) do
18:    if a ∈ 𝕊_o then
19:       a ← d
20:    else if a ∈ 𝕊_i and a? = d then
21:       a ← "accepted"
22:    else
23:       a ← "erroneous"
24:    end if
25:    I ← I \ {a}
26: end for
27: for all a ∈ I do
28:    a ← "refused"
29: end for
```

FIG. 4.2. *The main loop of the execution machine (on the left) and the execution of a step A with respect to requested inputs given by I (on the right).*

- each execution of the "**while**" loop performs a bounded amount of work, in particular the following numbers are bounded: the number of ports; the number of transitions outgoing from a state; the number of actions in each step. Assuming that choosing a transition requires a fixed amount of time (see below), Δ is the maximum amount of time required to execute the "**while**" loop $\delta - 1$ times;
- no tick is explicitly executed but its occurrence actually corresponds to the execution of the line 11.

PROPOSITION 5.1. *The algorithm presented in the figure 4.2 ensures an even occurrence of the ticks.*

Proof. Let θ be the value assigned to Θ when the line 2 is executed. A number of transitions (at most $\delta - 2$) is executed until a tick transition is chosen. All together, the duration of these executions requires is $D \leq \Delta$ so the line 10 waits during $\Delta - D$. Thus, the line 11, which corresponds to the tick, is executed at time $\theta + D + (\Delta - D) = \theta + \Delta$. By induction, we obtain that ticks are executed at times $\theta + k\Delta$ for $k \geq 1$. □

5.1. Choosing a transition. We still have to define how a transition may be chosen, in a fixed amount of time, in order to mark "accepted" as much as possible input ports in the set I of requested communications. In order to define a criterion of maximality, we assume that there exists a total order on $\mathbb{S}_i$. This corresponds to a priority between the ports: when several communications are requested but not all are possible, we first choose to serve those on the ports with the highest priorities. Then, given I, we define a partial order $\prec$ on the transitions outgoing from a state and the machine chooses one of the smallest transitions according to $\prec$. This choice may be random or driven by a scheduler. For instance, we may choose to execute steps as large as possible, or steps no larger than the number of processors, etc. The definition of a scheduling strategy is out of the scope of this paper; we just need to assume that the time needed to choose a transition is bounded (which should hold in the reasonable cases).

For each step A appearing on a transition outgoing from the current state, we define a vector $V_A \in \{0,1,2\}^{\mathbb{S}_i}$ which represents the marks on the input ports after A would be executed: the value 0 stands for "accepted" or "no mark", the value 1 for "refused" and the value 2 for "erroneous". Thus, the value of $V_A(a)$ can be found using the following table, where d and d' are distinct values in $\mathbb{D}_{io}$:

	$A[a] = 0$	$a(d) \in A$	$a(d') \in A$
$a? = d$	1	0	2
$a? = \varnothing$	0	2	2

Then, $A_1 \prec A_2$ if $V_{A_1} < V_{A_2}$ according to the lexicographic order on these vectors.

Again, it is clear that building these vectors and choosing the smallest one is feasible in a fixed amount of time since the number of transitions outgoing from a state is bounded. This is also feasible in parallel: all the V_A's can be computed in parallel (as well as all their components) and the selection of the smallest one is a logarithmic reduction.

Notice that if $\prec$ allows to define a total order on steps, it is not the case for the transitions since several transitions may be labelled by the same step. For instance, assuming $u \notin \mathbb{S}_i \cup \mathbb{S}_o$, the running example would give a CT-automaton similar to that of the figure 4.1 but in which all the actions $u(0)$ or $u(1)$ would have been deleted. In this case, the state 13 would have two outgoing transitions labelled by $\emptyset$ and three labelled by $a_2(1)$.

PROPOSITION 5.2. *Let $a \in \mathbb{S}_i$ be an input action symbol and N be a CT-net which is tick-reactive to $R \ni a$. Then, the execution of $\mathcal{A}(N)$ will never mark a as "erroneous" nor "refused".*

Proof. Let s be the current state of $\mathcal{A}(N)$ and (s, A, s') be the transition chosen by the execution machine. There are three cases.

(1) If $a? = \varnothing$, then it may be marked "erroneous" or not marked. In the former case, this means that $a(d) \in A$ for a $d \in \mathbb{D}_{io}$. Then, if $A = \{a(d)\}$, because of the tick-reactiveness, there must exist a transition (s, U', s'') which does not involve a (tick-reactiveness never forces the occurrence of an input action), otherwise, the transition (s, A', s'') with $A' \stackrel{\text{df}}{=} A \setminus \{a(d)\}$ must exists (since it corresponds to a sub-step). In both cases, we have $(s, U', s'') \prec (s, A, s')$ or $(s, A', s'') \prec (s, A, s')$ hence a contradiction with the fact that (s, A, s') was chosen. So, a must be not marked in this case.

(2) If $a? = d \neq \varnothing$ and the communication on a is marked "refused", this means that $A[a] = 0$. The tick-reactiveness ensures that there must exist a transition $(s, A \cup \{a(d)\}, s'')$ (by assumption, a cannot have been requested before since the previous tick), hence again a contradiction. So, a must be marked "accepted" in this case.

(3) If $a? = d \neq \varnothing$ and the communication on a is marked "erroneous", this means that $a(d') \in A$ for a $d' \in \mathbb{D}_{io} \setminus \{d\}$. But there must exist a transition $(s, (A \cup \{a(d)\}) \setminus \{a(d')\}, s'')$ (tick-reactiveness allows the occurrence for any value in $\mathbb{D}_{io}$), hence again a contradiction. So, a is also marked "accepted" here. □

Then, the next result shows that a communication requested on a port to which the CT-net is tick-reactive is always correctly handled (*i. e.*, accepted) within the current "**while**" loop, which is the best response time that one can expect from the presented algorithm.

PROPOSITION 5.3. *Let $a \in \mathbb{S}_i$ be an input action symbol and N be a CT-net which is tick-reactive to $R \ni a$. If $a? = d \neq \varnothing$ before the execution of the line 7 in the figure 4.2, then a is marked* "accepted" *after the line 13 has executed.*

Proof. Directly follows from how the machine chooses a transition and from the proposition 5.2. □

6. Concluding remarks. We defined a parallel execution machine which shows the adequacy of causal and real time by allowing time-consistent executions of causally timed Petri nets (CT-nets) in a real-time environment. We also shown that it was possible to ensure that the machine efficiently reacts to the solicitation of its environment by designing CT-nets having the property of tick-reactiveness, which is easy to construct. In order to obtain these results, several restrictions have been adopted:

- only *safe* Petri nets are considered;
- the nets must be *tractable*, *i. e.*, they are not allowed to have unbounded runs between two ticks;
- the nets must be *consistent*, *i. e.*, they cannot perform several simultaneous communications on the same port;
- the execution machine must be run on a computer *fast enough* to ensure that the environment cannot attempt more than one communication on a given port between two ticks.

We do not consider the tractability and consistency requirements as true restrictions since they actually correspond to what can be performed on a realistic machine. The last restriction is actually a prescription: in order to ensure a correct communication, one has to run the execution machine on a computer fast enough to execute ticks more often than the environment can produce input. Moreover, it should be noticed that the frequency of ticks is arbitrary. So, if the ticks of a CT-net are too much sparse with respect to the requested inputs, it is easy to multiply by a constant k all its timing constraints in the net so ticks will occur k times more often. Using non-safe Petri nets may be considered in the future, however, this would lead to the class of infinite state systems which does not seem realistic for the purpose of execution.

6.1. Future work. Petri nets like CT-nets have been used for a long time as a semantical domain for high-level programming languages and process algebras with step based semantics (see, *e. g.*, [3, 14]) and these techniques could be directly applied to massively parallel languages or formalisms. In this direction, we envisage to combine a n-ary parallel composition operation with symmetry reductions [9] allowing to the verification of very large systems while giving modelling support for kinds of SPMD systems.

6.2. Implementation issues. A preliminary version of this work proposed a sequential execution machine and a prototype has been successfully implemented in Ada; this allowed to show that the evenness of ticks was not only possible in the theory but also easy to achieve in an implementation. (The only "difficulty" was to obtains Δ using test runs at the starting of the machine.) A parallel implementation of the version presented here had been started but had to be delayed since it turned out that there were still need for a ground study. Indeed, several open questions are actually critical ones. Notice that if our goal is to perform testing or simulation, an implementation can be naive and may even be sequential. But in the perspective of direct execution of the modelled systems, the speedup becomes crucial and actually depends on the interaction between several parameters: the model of computation, the family of parallel machine targeted and the scheduling strategy (as discussed in the section 5.1). All these questions were left out of the current paper; we thus envisage further research on this subject with the goal to identify good combinations allowing to produce high-quality implementations of our execution machine. In particular: how to exploit the parallelism in the presented algorithm strongly depends on the computational model envisaged (which may itself depend on the target architecture); the question of storing the CT-automaton is also important if one targets a distributed memory architecture. Taking all these parameters into account may lead to several very different refinements of the algorithm proposed above, each specially dedicated to a particular class of parallel computer and parallel programming language or model.

Related to the goal of efficient executions, another interesting problem is to connect the input/output of the machine to the concrete computer in order to delegate some computation. Indeed, output actions may be considered as calls to computational primitives, while input actions could correspond to the receiving of the computed values. This introduces delays, externals to the model, which must be taken into account. This can be made by introducing further timing constraints in the model in order to reflect the execution times obtained from benchmarks or from real-time guarantees in the case of calls to real-time primitives. In this perspective, considering Petri nets with time becomes necessary.

6.3. Conclusion. We believe that the framework proposed in this paper can be used to build concrete parallel applications in which the control flow could be ensured by Petri nets while a large part of the computation would be delegated to dedicated primitives with known performances. Using Petri nets for both the modelling and the execution allows to verify and run the same object, saving from the risk to introduce errors on the way from a model to its implementation, while allowing executions even during the early stages of the design.

REFERENCES

[1] C. André F. Boulanger, A. Girault, *Software Implementation of Synchronous Programs*, ICACSD'2001, IEEE Computer Society, 2001.
[2] G. Berry, *The foundations of Esterel*, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, 1998.
[3] E. Best and R. P. Hopkins, *B(PN)^2 — A basic Petri net programming notation.* PARLE'93. LNCS 694, Springer, 1993.
[4] E. Best and H. Wimmel, *Reducing k-safe Petri nets to pomset-equivalent 1-safe Petri nets*, ICATPN'00. LNCS 1825, Springer, 2000.
[5] E. Boufaïd, *Machines d'ex'ecution pour langages synchrones*, PhD Thesis, University of Nice-Sophia Antipolis, 1998.
[6] C. Bui Thanh, H. Klaudel and F. Pommereau, *Petri nets with causal time for system verification*, MTCS'02. ENTCS, Elsevier, 2002.
[7] R. Durchholz, *Causality, time, and deadlines*, Data & Knowledge Engineering, 6. North-Holland, 1991.
[8] J. Esparza, *Model checking using net unfoldings*, Science of Computer Programming, Elsevier, 1994.
[9] T. Junttila, *On the Symmetry Reduction Method for Petri Nets and Similar Formalisms*, PhD Thesis, Helsinki University of Technology, 2003
[10] H. Klaudel, *Compositional High-Level Petri nets Semantics of a Parallel Programming Language with Procedures*, Sciences of Computer Programming 41, Elsevier, 2001.
[11] H. Klaudel and F. Pommereau, *A concurrent semantics of static exceptions in a parallel programming language*, ICATPN'01. LNCS 2075, Springer, 2001.
[12] H. Klaudel and F. Pommereau, *A class of composable and preemptible high-level Petri nets with an application to multi-tasking systems*, Fundamenta Informaticae, 50(1):33–55. IOS Press, 2002.
[13] F. Pommereau, *Modèles composables et concurrents pour le temps-re'el*, PhD. Thesis, University Paris 12, France, 2002.
[14] F. Pommereau, *Causal Time Calculus*, FORMATS'03. LNCS 2791, Springer, 2004.
[15] G. Richter, *Counting interfaces for discrete time modeling*, Technical report 26, GMD. September 1998.

Edited by: Frédéric Loulergue
Received: June 8, 2004
Accepted: June 9, 2005

Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 83–94. http://www.scpe.org
ISSN 1895-1767

AGENT BASED SEMANTIC GRIDS: RESEARCH ISSUES AND CHALLENGES

OMER F. RANA* AND LINE POUCHARD†

Abstract. The use of agent based services in a Computational Grid is outlined—along with particular roles that these agents undertake. Reasons why agents provide the most natural abstraction for managing and supporting Grid services is also discussed. Agent services are divided into two broad categories: (1) infrastructure services, and (2) application services. Infrastructure services are provided by existing Grid management systems, such as Globus and Legion, and application services by intelligent agents. Usage scenarios are provided to demonstrate the concepts involved.

1. Introduction and Related Work. There has been an increase in interest recently within the Grid community [11] towards "Service Oriented" Computing. Services are often seen as a natural progression from component based software development [6], and as a means to integrate different component development frameworks. A service in this context may be defined as a behaviour that is provided by a component for use by any other component based on a network-addressable interface contract (generally identifying some capability provided by the service). A service stresses interoperability and may be dynamically discovered and used. According to [7], the service abstraction may be used to specify access to computational resources, storage resources, and networks in a unified way. How the actual service is implemented is hidden from the user through the service interface. Hence, a compute service may be implemented on a single or multi-processor machine—however, these details may not be directly exposed in the service contract. The granularity of a service can vary—and a service can be hosted on a single machine, or it may be distributed. The "TeraGrid" project [9] provides an example of the use of services for managing access to computational and data resources. In this project, a computational cluster of IA-64 machines may be viewed as a compute service, for instance—hiding details of the underlying operating system and network. A developer would interact with such a system using the GT4.0 [26] system—via a collection of services and software libraries.

Web Services provide an important instantiation of the Services paradigm, and comprise infrastructure for specifying service properties (in XML—via the Web Services Description Language (WSDL) for instance), interaction between services (via SOAP), mechanisms for service invocation through a variety of protocols and messaging systems (via the Web Services Invocation Framework), support for a services registry (via UDDI), tunnelling through firewalls (via a Web Services Gateway), and scheduling (via the Web Services Choreography Language). A variety of languages and support infrastructure for Web Services has appeared in recent months—although some of these are still specifications at this stage with no supporting implementation. Web Services play an important role in the Semantic Web [17] vision, aiming to add "machine-processable information to the largely human-language content currently on the Web" [12]. A list of publicly accessible Web Services (defined in WSDL) can be found at [21]. By providing metadata to enable machine processing of information, the Semantic Web provides a useful mechanism to enable automatic interaction between software—thereby also providing a useful environment for agent systems to interact [8]. The adoption of more complex representation schemes for metadata, such as WebONT [13], suggest that the software using this information can be more adaptive, and support updates when new information becomes available. The agent paradigm therefore provides a useful mechanism for managing and mediating access to Web Services. Various extensions of Web services through the agents paradigm have been discussed by Huhns [8]—the most significant in the context of Grid computing include self-awareness and learning capability, the ability to support a number of ontologies, and the formation of groups or teams of agents. Conversely, a key advantage of using agents is to support semantic interoperability (i. e. interaction between software systems based on pre-agreed, semantically grounded, definitions). Support of technologies such as WebONT in the context of Web Services are likely to provide the necessary core infrastructure for agents to work more effectively in dynamic environments such as Computational Grids.

2. Role of Agents in Grids. Grid computing currently focuses on sharing resources at regional and national centres. Generally, these include large computational engines or data repositories, often requiring the user to accept "usage policy" statements from the centre managers and owners. Similarly, resource owners are

*School of Computer Science, Cardiff University, UKo.f.rana@cs.cf.ac.uk
†Computer Science and Mathematics Division, Oak Ridge National Laboratory, PO Box 2008, Oak Ridge TN 37831-6367, USA pouchardlc@ornl.gov

obliged under the policy to guarantee access once an external user has been approved. Access rights to the resources are supported through X.509 certificates—whereby a user requiring access must posses a certificate. The `grid-proxy-init` function in Globus provides a mechanism for delegation—however, it is limited in scope, and protected mainly by standard Unix access rights. In this model, a trust-chain must be established before a proxy request can be accepted. Furthermore, system administrators responsible for particular resource domains are accountable—and operate based on the policy of the site. As Grid systems embrace service-oriented computing, more open and flexible mechanisms are necessary to support service provision and service usage, as a user providing a service may not belong to a particular centre. Hence, multiple providers may offer a similar service, and the service user now has to select between them. The more "open" perspective on Grids—whereby service providers can be a collection of centres or individuals—would necessitate a user evaluating service providers based on a number of different criteria, such as: choosing services which are best value for money, choosing the most "reputable" services, choosing the most secure services, or services which have the highest response (execution) time, or which have been around the longest. These criteria are therefore more diverse in scope, and can support service choice based on dynamic, run-time attributes of a service. We assume two kinds of services to exist within a Grid: (1) core services—which are provided by the infrastructure and by trusted users, and (2) user services—which can be provided by any participant utilising common Grid software—such as OGSA. Two such core services—responsible for managing access to user services—include:

- Certificate Authority (Security Service): The certificate authority is externally managed, and used to authenticate services—based on the identity of a service provider. Only a few of these services are likely to exist across a Grid—and aimed at ensuring that service providers can be verified. The Certificate Authority services is also used to support the development of service contracts between a service user and provider. A simple mechanism based on X.509 certificates already exists, and additional work is necessary to extend this to include users who require temporary certificates, or may change their identity over time. A criteria to be associated with such a service includes the "risk" of accessing a service which does not posses a certificate. In this context, the service user must now decide whether to not accept any service at all, or to choose one which is non-trustable. Such risk evaluation must be undertaken with other decisions being made by the service user—and within a limited time. The decision making capability needed to undertake such an evaluation can be supported through agent systems—and has been a subject of extensive research as "trust models" [31]. The concept of risk can be defined in a number of different ways—for instance, a high risk service may be one that is likely to give low-accuracy results (for a numeric service), or one that is provided by an unknown vendor. It is therefore important to qualify what is meant by risk in a particular instance.
- Reputation Service: Each service can have an associated "Reputation" index, which is used to classify how often the provider has fulfilled its Service Level Agreement (contract) in the past, and to what degree of confidence. It is possible for a particular service user to subscribe to multiple such Reputation Services—and indeed for a client service to look up the reputation of the providing service from multiple Reputation providers. The concept of Reputation Services have been developed in the Peer-2-Peer computing community [14], and aimed at increasing accountability within a system of anonymous peers. Another concept of reputation (in the FreeHaven project [15]) requires service owners to provide "receipts" (feedback) to verify the correctness of results obtained from other services they interact with. These receipts are coupled with services that act as "witnesses" to ensure that receipts have been generated, and thereby can judge node misbehaviour. In the context of Grid services, witnesses can be external nodes which monitor that a given node has met its Service Level Agreement, and can verify that the feedback provided by the user on the service provider is accurate.

A Reputation or Certificate can be used by a client service to identify whether to use a particular service provider. This confidence in a given service is important in the context of service-oriented Grids—as it allows requesting services to select between multiple providers with a greater degree of accuracy. Agents provide the most suitable mechanism for offering and managing Grid services. Each agent can be a service provider or user, or can interact with an existing information service.

We therefore assume that services within a Grid environment are managed and executed via agents. It is also possible for each agent to support one or more "service types" (see section 4.2). We assume three kinds of agents to be present: (1) Service Providers, (2) Service Consumers, and (3) Community Managers (see section 4.1). Each agent must therefore provide support for managing a community description, managing and sustaining interactions with other agents, and provide a policy interpreter. The policy interpreter is used by a service

provider and a community manager to ensure that a service provider conforms to its service provision contract. Particularly important in Grid systems is the role played by middle agents—primarily service providers which do not offer an application service, but act as brokers to discover other services of interest. The criteria for service discovery used by a broker may range from service type to service reputation—and a service consumer may simultaneously invoke a number of different service providers (brokers) to undertake this search.

The particular challenges therefore include the ability to assess the risk associated with using a service, and provide feedback to potential users to evaluate this risk. Middle agents can support the management of risk within an agent community—enabling agents to combine the use of trusted services along with newer ones.

3. Service Lifecycle. Each agent is responsible for managing one or more services—and each agent may utilise a number of different infrastructure services to achieve this. An agent exists within a particular community, and utilises infrastructure services (such as a security or registration service) within its community first. A service lifecyle identifies the stages in creating, managing, and terminating a service. A new service may either be created by an agent, or a service may be associated with an agent by a user—where access to the service is subsequently mediated by the agent. A new service may also be created by combining services offered by different agents—whereby an agent manages a service aggregate. The agent is now responsible for invoking services in the order specified in the composition process (specified in a service enactment contract). Once a new service has been created, it must be registered with its "community manager" by the agent. A service is initialised and invoked by sending a request to the agent managing the service, which may either agree to the request immediately, or offer a commitment to perform the service at a later time. Service termination involves an agent unregistering a service via the community manager, and removing all data corresponding to the service state. When an agent needs to execute an aggregate service, it will involve interactions with agents within multiple communities. The manager within each community is responsible for ensuring that service contracts are being adhered to by agents within its community. The ability to create a service aggregate leads to the formation of "dynamic workflow"—whereby an agent decides at run time which other agent it needs to interact with to achieve a particular goal. Consequently, the exact invocation sequence between services is not pre-defined, and may vary based on the operating environment of the agent undertaking the aggregation. The following technical challenges are significant in the context of Service Lifecycles:

- Service Creation: Creating a service description using a standard format is an important requirement—to enable the service to be subsequently discovered. The creation of a service also necessitates associating the service with an agent. An agent would receive a request for an application service and create a new instance of it using the Factory Interface [7]. Each agent therefore provides a persistent place holder for an application service. An important challenge in this context is determining the number and types of services that should be managed by a single agent.
- Service advertising and discovery: Registering a service with the local community manager may restrict access—unless there is also some mechanism to allow community managers to interact. Discovering a service across multiple network based registries becomes an important concern—and efficiency of the referral and query propagation mechanisms between community managers become significant. The greater the number of participants that need to be contacted to search for a service, the more time consuming and complex the search process will be. The number of registries searched to find a service of interest becomes an important criteria, as does the mechanism used to formulate and constrain the query. The ability to divide a query into sub-parts which can be simultaneously sent to multiple registries is useful in this context—although it restricts the specification of a query.
- Contract enforcement: The community manager is responsible for ensuring that a request for service provision is being honoured by an agent within the community. There is a need for monitoring tools to verify that a contract is being adhered to—although this requires an agent to reveal its internal state to the monitoring service. Enforcement of a contract also requires the community manager to de-register the service or to restrict access to it if it does not meet its contract. As previously discussed, it is also possible for a community manager to change the risk or reputation index of such a service—and utilise monitoring tools to periodically update this. Contract enforcement must be undertaken based on a community specific policy.

A service may also register interest in one or more event types via its agent or the community manager. Certain event types may be common for all services within a community, and handlers for these provided at service creation time. Such an event mechanism may also provide support for service leasing—whereby a service is

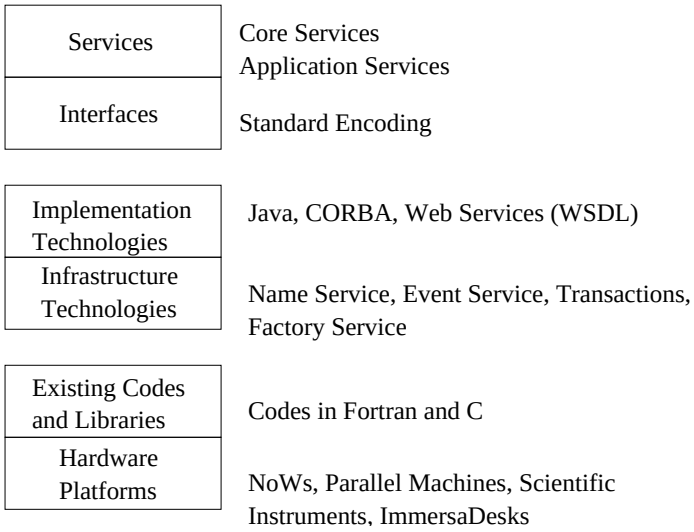


FIG. 4.1. *The Services Stack*

only made available to a community (or to external agents) for a lease duration—the lease is monitored by the community manager. When the lease period expires, the service agent must either renew the lease or delete the service.

4. Service Types and Instances. Figure 4.1 illustrates the layers within service oriented Grids—starting from the services themselves (which can be infrastructure or user services) and interfaces to these services encoded in some agreed upon format. At present no standard exists within the Grid community, although there are working groups in the GGF [11] exploring standard interfaces for services within a particular application domain. Existing work on the Common Component Architecture (CCA) [10] provides a useful precedence for developing common interface standards. Some of the services may also wrap existing executable codes, developed in C or Fortran—requiring the users of these legacy codes to publish interfaces to their code.

Services may subsequently be implemented using a number of different technologies—and interface definitions using WSDL may bind to a number of different implementations. Service interaction is then supported through an infrastructure that provides support for service registration and discovery, distributed event delivery between services, and support for transactions between services. Currently, this is provided by systems such as Globus, although the need for integrating such infrastructure services from other platforms, such as Enterprise JavaBeans or CORBA becomes significant.

Services are assumed to be of two categories: (1) infrastructure services provided via Globus/OGSA (for instance), and (2) application services provided by agents. Examples of infrastructure services include a Security Service, an Accounting Service, a Data Transfer service etc. Examples of application services include Matrix solvers, PDE Solvers, and complete scientific applications. Agents utilise infrastructure services on-demand, and may use type information made available by infrastructure services. Agents can also interact with each other based on a goal they are aiming to satisfy.

A minimal set of service metadata should be agreed upon by all agents within a community, regardless of the application domain—referred to as a "Services Ontology". Such an ontology would be used by agents to discover other service providers and service consumers, and the types of services they offer—and based on the Grid Services Specification (GSS) [22]. Terms within such an ontology can include the concept of file/service title, authors/service manager, locations, dates, and metadata about file content—such as quality, provenance etc. Each agent responsible for a service must also decide how to process requests being made to a given service that it manages. These criteria may be enforced by the community manager, or based on the attributes of the services being managed by the agent.

4.1. Service Interactions and Communities. Interactions between services form an essential part of Grid systems, with interactions ranging from simple requests for information (such as extracting data from the Grid Information Index Service (GIIS) in Globus), to more complex negotiation mechanisms for arranging

common operations between services (such as co-scheduling operations on multiple machines). Interactions between agents are constrained by the paradigm used—such as the concept of a "virtual market"—whereby agents can trade services based on a computational economy [30]. An important aspect of such an interaction paradigm is that agents need to make decisions in an environment over which they have limited control, restricted information about other agents, and often a limited understanding of the global objectives of the environment they inhabit. The concept of "communities" becomes important to limit the complexity of decisions each agent needs to make, by limiting interactions to a restricted set of other agents. In the community context, agents must be able to first establish which communities to join, and subsequently to decide upon mechanisms for making their local state visible to others. Each community must have a manager entity, responsible for admitting other agents, and for ensuring that agents adhere to some common obligations within the community. Interaction between agents may also be mediated via such a manager—whereby the manager also acts as a protocol translator. The community manager is also responsible for advertising the properties of a community to others, and for eventually disbanding a community if it is non-persistent.

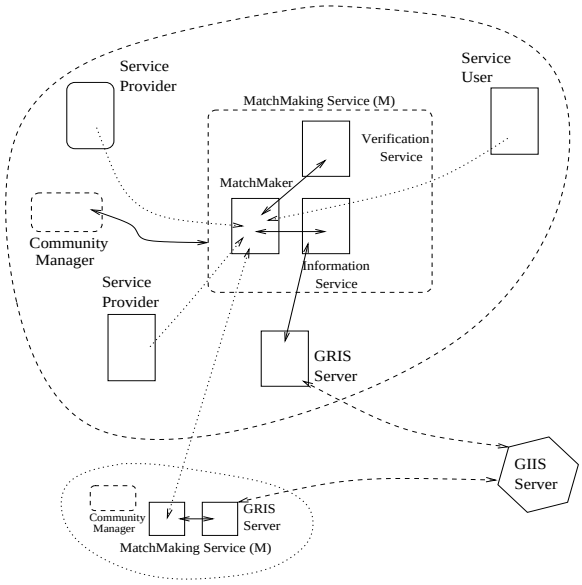


FIG. 4.2. *Service Community*

Figure 4.2 illustrates the core services provided within each community, and consists of service users/providers, a MatchMaker (M)—which is supported via a verification and information service, and a community manager. The MatchMaker provides an example of a middle agent, facilitating interaction between other service users and providers within the community. The information service can interact with the GRIS/GIIS server and locate other computational resources of interest—using the Globus system. Interaction between the service user and provider is undertaken based on a common data representation—which enables the state of a given service to be queried at a given time 't' (an example of this data model for computational services can be found in [23]). We assume that there is a single M within a community, although the request for match may utilise different criteria. The availability of a service over time extends from $t < t_{\text{current}}$ (usage history) to $t > t_{\text{current}}$ (projected usage) and includes $t = t_{\text{current}}$ (current usage). Availability over time is just one of the parameters that must be supported in the system, for instance, we also consider availability over the set of service users. The matchmaking service works as follows:

- Each Service Provider sends an asynchronous message to a pre-defined matchmaking service 'M' (running on a given host) to indicate its availability within the local community. Each message may be tagged with the service type that is being supported. The message contains no other information, and is sent to the local 'M'. The identity of M may be pre-built into each service when it is created, or may be obtained from the community manager agent (via a multicast request within the community).
- On receiving the message, the local 'M' responds by sending a document specifying the required information to be completed by the service provider agent. This information is encoded in an XML

document (see [23]), and contains specialised keywords that correspond to dynamic information that must be recorded for every service managed by the agent. The document also contains a time stamp indicating when it was issued, and an address for 'M'.

- The service provider agent completes the document—obtaining the necessary information via the GIIS server (if necessary), and sends back the form to 'M', maintaining a local copy. The document contains the original time stamp of 'M', and a new time stamp generated by the service manager. Some parts of the document are static, while others can be dynamically updated. The new service is now registered with the community manager, and can be invoked by a service user, until it de-registers with 'M'. If the service is terminated or crashes, 'M' will automatically de-register it when it tries to retrieve a new copy of the document. An alternative technique would involve a 'push' model whereby each service updates M with its state on a change. Typically, the update would be to describe changes in availability, for example after a reservation has been made by a service user. However, the update could also involve a change in capability, for example an extra service being added to the local system. If a push mechanism is used from the service to M then repeated polling of the resources is not necessary. It is useful to note that the community manager does not directly maintain any service information or content itself, and interacts with M to obtain the necessary service details.

Agents within a community may need to undertake multiple interactions to reach consensus. For instance, an agent trying to discover suitable services may need to issue multiple discovery requests before it is able to find a suitable service. Interaction mechanisms between agents therefore may be more complex, and utilise auction and negotiation mechanisms, or interaction rules. The community manager may provide mediation in this process, by restricting the maximum number of message exchanges between agents. The main objective being to enable service providers to enable their services to be more effectively used.

A particular challenge in this context is the ability to agree on a common data model for exchange service capability documents. There must be some agreement based on GSS [22], but also the ability of a service provider to identify additional properties if available in the service interface. Another important challenge is to identify the complexity of the match process (from a syntax based match to a semantic match—for instance)—and to enable a user to limit the complexity of the match in their request to 'M'.

4.2. Service Semantics. Service interactions require definitions of common terms—the definition of common units when exchanging engineering data for instance (where one service may reports its results in miles, while the service user undertakes its processing in kilometres). Service semantics are generally assumed in distributed systems—where checks on the results can be made by a user. However, when services interact directly, it is important to ensure that the results they produce follow some predefined types.

Service types may be "abstract" types—directly supported by a service, or "derived" types which are obtained by extending or combining abstract types. An agent therefore also publishes type information associated with the services it supports—enabling service users (other agents) to undertake the necessary type conversions. Service types can be based on data types supported within the service implementation—such as `float`, `string`, etc, or they may be application related—such as a `distance` type or a `co-ordinate` type. The service type mechanism may be extended into an ontology—which may also identify additional attributes, such as particular instances of types, axioms for transforming between types, and constraints on types.

The type mechanism is also used for discovering other services, and for launching specialist services which provide a particular output type. The semantics associated with a particular type must also be defined by a service—hence, a service which uses a derived type `distance`, must prefix it with its service identity. Consequently, services with similar types but different semantics may co-exist, and can publish this information as part of their interface descriptions. One example of semantic services include mathematical libraries (such as in the MONET project [20]) with predefined categorisation of these numeric libraries. In this context therefore, a search for a numeric solver service by a user in a particular application domain would proceed by contacting one more more broker agents and perform matching based on problem domain, along with various non-mathematical issues such as the user's preferences for particular kinds or brands of software. The motivation stems from the observation that many scientists prefer to use services from particular developers, a decision often determined by the application domain of the scientist. This subjective criteria should therefore be utilised when searching for suitable numeric services—and used along with the operational interface the service offers.

In a typical Grid environment, multiple domain specific ontologies are likely to co-exist. Work being undertaken in the Gene Ontology Consortium [24] provides one example of a vocabulary being developed to

support software interoperability. There are therefore likely to be a number of common services (based on a generic services ontology), and a number of specialist services (such as mathematical libraries, gene clustering software etc), which can only be invoked in a limited way, and by a restricted set of other services. An important challenge in this context is to identify the granularity at which these domain specific services should be described, and whether advertising of services should be restricted. Also important is to identify how services across domains can be defined in common ways—for instance, the use of clustering and data analysis services may be common in a number of different domains. However, the particular description schemes used may vary. Many of the concerns related to the definition of ontologies needs to be undertaken within the particular scientific community involved—although ways of identifying common services used by a number of different communities would be a useful undertaking.

5. Scenario. We illustrate the concepts outlined in this paper via a project which uses agents for managing user access to scientific instruments at Oak Ridge National Laboratory (ORNL). It was mainly aimed at automating an existing manual process of approving user requests to obtain time on a microscope and other scientific instruments. The project was undertaken as part of the Materials Microcharacterization Collaboratory (MMC) [16] project, involving ORNL and various other participants. The purpose of collaboration within the MMC is to characterise the microstructure of material samples using techniques such as electronic microscopy, and X-ray and neutron diffraction. Observation, data acquisition, and analysis are performed using instruments such as transmission and scanning electronic microscopes, and a neutron beam line. An important aspect of the MMC project is the computer co-ordination and control of remote instrumentation, data repositories, visualisation platforms, computational resources, and expertise, all of which are distributed at various sites across the US. The role of ORNL in this collaboratory was to provide access to, and management of experiments within the High Temperature Materials Laboratory [18]. A scientists is required to complete a pre-formatted proposal document (a part of this is illustrated in figure 5.1), and pass this to a central facility. Based on the type of experiment, and the instrument identified, the facility selects one or more experts to evaluate the proposal. The selection criteria involves economic factors (such as industrial impact the experiment is likely to have), technical factors (such as types of materials to be analysed in the experiment), safety factors (such as whether the user has had radiation or general training on the instrument), and credibility factors (such as what publications the user already has in the field, why the experiment is being requested etc). These factors are weighed by the expert, and a decision is made on whether the proposal to undertake the experiment should be granted. The project was conceived to automate some of the processing involved in reaching a decision on the initial proposal. It was decided that replacing the expert was not a viable option, as this would involve a detailed knowledge elicitation from existing experts, and the effort and time involved in such an undertaking would be significant. Instead, the approach adopted was to support the decision making process of the expert, and to automate as much analysis of the proposal as possible, prior to delivery of the proposal to the expert.

The automation of the current system was achieved using Web based forms, CGI scripts and an agent development tool. An agent is used to represent every entity involved in the system, and includes a "User" agent, an "Expert" agent, an "Instrument" agent, an "Experiment" agent, and two utility/middle agents, a "Scheduling" agent and a "Facilitator" agent. Each of these agents perform a pre-defined set of services, which must interact to complete the overall request. Message exchanges between agents can relate to requests for proposal to be verified, confirmation or denial of a proposal, and a verification of scheduling request. Each agent operates as an autonomous entity, in that it manages and makes requests for information to other agents, in order to achieve a given goal. The goals are specified by the physical entities which are being represented by the agent—such as a human user (for a User agent), or an instrument expert (for an Expert agent). Each agent then tries to find a set of services to be undertaken to reach the goal it has been set. Goal completion is based on each agent choosing an initial action that will lead it closer to its goal, and determined by the pre-conditions for a given action to be taken, and post-conditions (or effects) identifying the outcome of a given action on the agent itself, and its environment. The agent based approach provides the best option for modelling scenarios where a large number of users, instruments and experts can co-exist, with each entity controlling and managing its own requirements and goals.

`MatML` for Materials Property Data [25] is used for specifying intrinsic characteristics of materials. In the DeepView system developed for the MMC [27], an instrument schema has been designed for instrument properties permitting the remote, on-line operation of microscopes [28]. These schemas were examined to form the basis of a local ontology for our system. However, re-use of existing schemas raises questions concerning

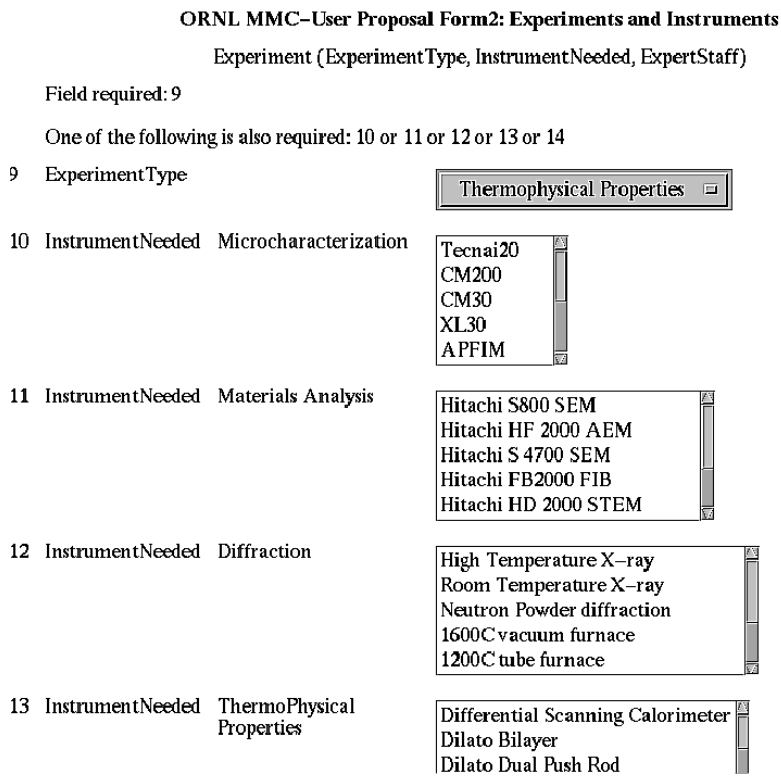


FIG. 5.1. *Form completed by the user*

the purpose and scope of an ontology within the context of an agent-based system—as our objective was to enable a user to access an instrument and performance of the system was of issue [29]. With these constraints in mind, it was decided that the concepts in the ontology must focus on use of instruments and characteristics of (human) users rather than on properties of materials such as chemical composition and geometry (MatML), and instrument characteristics such as vendor and resolution (DeepView). For these and other reasons, a domain ontology for our system was created that did not re-use concepts in the schemas mentioned above. The domain ontology is divided into four categories: Users, Experts, Experiments and Instruments—figure 5.2 illustrates the "Experiment ontology". Terms used within the ontology can take on a number of different content types—such as integers, reals, strings—and constraints are defined as ranges on these basic types. An important concern was to identify mechanisms to translate existing types supported in the form, into types that could be directly interpreted by the agents. Some attributes in the ontologies utilised by the agents required an appropriate representation of "Phase" (in the Instrument ontology), the concept of "Impact" (in the Experiment ontology), and common ways to encode time and date information. It was also necessary to constrain parameters associated with ontologies maintained by different agents—to enable interaction between agent roles.

Each agent in the system undertakes a particular set of actions to achieve its "role". A role is defined as a set of goals that need to be completed by an agent, in a given context. Hence, a User agent plays the role of an external user. In the context of the MMC, this involves "Creating a Proposal" and "Accepting a Proposal". A role is defined at a higher level of abstraction than method calls on objects, or sub-routine calls in source code. In an agent based system, a given entity (or agent) can only undertake pre-defined roles which determine its function in a given society of other agents. Hence, a User agent in this particular context cannot schedule operations on a given instrument, because it does not possess this as a role. It can make a request to a Scheduling agent to undertake such an operation, or alternatively, to communicate with an Expert agent to request a given schedule to be validated. Agents can therefore posses roles and relationships with each other based on their particular function in the agent society. It is assumed in this project that agents cannot change or modify their roles or services, although they can update the information content of their local repositories based on interactions with other agents.

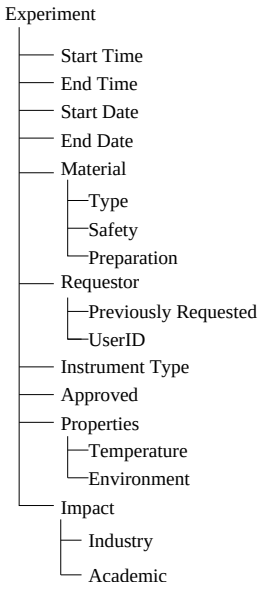


FIG. 5.2. *The "Experiment" ontology*

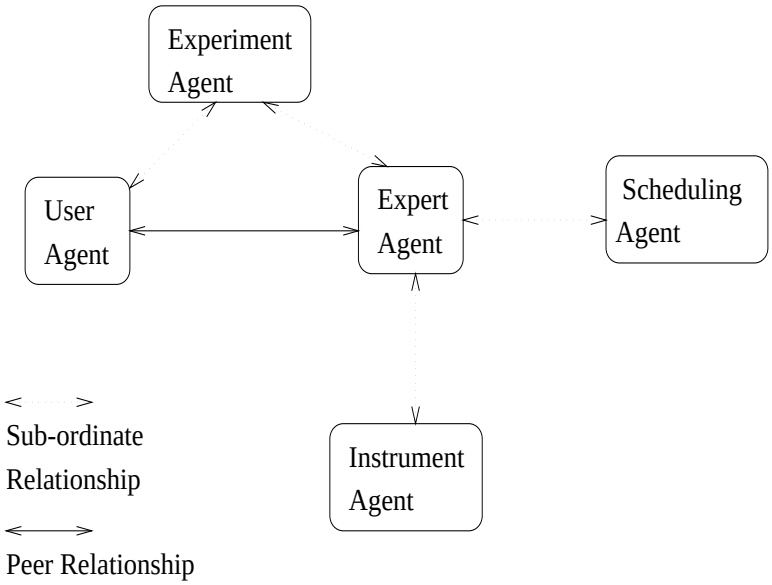


FIG. 5.3. *Co-ordination mechanism and role interaction between collaborating agents for MMC resource allocation*

A User agent and an Expert agent have a peer-to-peer relationship, as each can initiate a request to the other one. An Instrument agent is a sub-ordinate to an Expert agent, as an Expert agent can request information from an Instrument agent, but not vice versa. Roles between agents in the MMC system are illustrated in figure 5.3. Each agent in the system, and the particular services undertakes are as below:

- *User Agent:* This agent undertakes two basic services: `CreateProposal` and `AcceptProposal`. The CreateProposal task involves reading a file from disk, based on a given User ID, and initiating a proposal request to an Expert agent. The AcceptProposal task involves verifying that the schedule given by the Expert agent is acceptable—the acceptance criteria is based on checking constraints defined in the proposal with the initial request made by the User agent.
- *Expert Agent:* This agent is the most complex of all, and acts as the co-ordinator. The Expert agent can undertake one of five different services: `ReceiveProposal`, `RequestInstrument`, `CheckSchedule`, `ConfirmSchedule` and `ValidateRequest`. ReceiveProposal involves accepting a User generated request

to undertake a given experiment. RequestInstrument involves verifying constraints via the Instrument agent, based on availability of the instrument, and whether the parameters for the requested experiment are valid for the given instrument. Only two such parameters were identified as being relevant for this prototype—the "Operating Temperature" of the instrument, and the "Phase ID". Both of these are compared with the initial request from the User agent to confirm that a given instrument can support these ranges or absolute values. The CheckSchedule and ConfirmSchedule involve checking constraints on the availability of the instrument, with the availability of the expert. For the MMC, it is identified as a requirement that an instrument and an expert are available over the same time period, and that this falls within the duration of the requested experiment. The CheckSchedule task validates that such an overlap exists, and the ConfirmSchedule task generates a message to the Scheduling agent confirming the Schedule is valid. The ValidateRequest task is used by the Expert agent to confirm that a given request from a User agent does not violate any existing schedules that have already been decided. The Expert agent achieves this by interacting with the Scheduler agent, and checking the stored schedules.

- *Instrument Agent:* This agent acts as a wrapper for a microscope, and is used to identify particular access parameters required to request it for an experiment.
- *Experiment Agent:* This agent can interact with a User agent or an Expert agent to help them prepare an experiment. It supports the generation of proposals by a User agent, and the verification or checking of these by an Expert agent. Its primary purpose is to act as a support agent for helping formulate proposals, and help the User and Expert agents negotiate over parameters identified in a proposal. The Experiment agent undertakes three services: `PrepareProposal`, `CheckProposalRequest` and `ValidateProposalRequest`. The PrepareProposal task is activated by a User agent, and involves the Experiment agent helping to complete missing parameters in the proposal being sent to it. The CheckProposalRequest is used by an Expert agent to ensure that the parameters requested in a proposal are valid. The ValidateProposalRequest is used by the Experiment agent to undertake the above two services based on its local database of facts. The database is an external program that must be provided by the developer of the system.
- *Scheduling Agent:* This agent maintains a list of all valid schedules at any time, and can undertake three services: `ReceiveRequest`, `ConfirmRequest` and `ValidateSchedule`. The ReceiveRequest task involves accepting a request to verifying a proposal from an Expert agent. The Scheduling agent acts as a sub-ordinate of the Expert agent, and provides support to the Expert agent to reach a particular goal. The ValidateSchedule task involves verifying the requested schedule against its database to ensure that the requested schedule does not conflict any already assigned. The ConfirmRequest task is then used to send a message to the given Expert agent to confirm or deny the request.
- *Facilitator and Name Server Agents:* These agents acts as utility agents, mapping an agent location to its IP address (for the Name Server agent), and identifying services that a given agent can undertake, in some respects similar to a yellow page service (for the Facilitator agent).
- *Globus Gateway Agent:* The Globus/OGSA gateway agent enables an Experiment agent to launch jobs on remote instruments. Job management can be supported via the MatML data model. The gateway agent also makes use of the Facilitator and Name Server to locate and communicate with other agents.

A prototype system was implemented using the Zeus agent development tools [32].

5.1. Barriers and Discussion. Services supported by agents need to interact with infrastructure services provided through tools such as Globus/OGSA—although this is only necessary to support execution of scientific codes. Agents must therefore interact with existing Grid services via one or more gateways. Performance issues become significant when deploying agents to manage services—as no direct interaction between services exist. Existing Web services technologies—such as the use of SOAP—can have significant overheads, primarily due to the HTTP transport used and the parsing of XML based messages—especially when encoding data types along wit the content (a useful study on SOAP performance can be found in [33]). Standards such as DIME [19] may provide some performance improvement. Therefore, although the use of Web Services infrastructure may provide an important route for a wider use of Grid infrastructure, the performance implications introduced by such technologies still need to be overcome (the scientific codes currently deployed via Grid middleware have performance as a key requirement). Although many scientists may be willing to relinquish this requirement in the prototyping phase of their work—deploying production codes in this way may not be possible. Many Web Services standards are also at an early stage of development at the present time, and most experimentation

is still being undertaken behind firewalls. It is also not apparent how the UDDI (service registries) are to be managed, and by whom. Should there be a few "root" UDDI registries (like current Domain Name Servers), or should the registration mechanism be more distributed? Some of these concerns need to be evaluated in the context of Grid registration services (currently utilising Globus/OGSA), to enable more effective sharing of Grid Services across applications. We also see a number of similarities between the Peer-2-Peer (P2P) approach [1] and agent systems—as both focus on service provision through a decentralised model of cycle sharing or file sharing. Whereas agent systems focus on the semantics of the shared services, the focus in P2P systems is on the efficiency of the routing mechanism used.

The use of the service oriented approach for deploying scientific codes also requires the delegation of control to a remote service. This is especially true when service aggregation is being undertaken by an agent. It is therefore important to identify how ownership is delegated in the context of such a composition process, and how a service contract must be defined and enforced for the aggregate service. One incentive for supporting such an aggregation of services may be based on the concept of a "virtual economy" [30]—whereby services can have associated costs of access and deployment. Although a useful model (and one which closely resembles the current usage of computational resources at national centres)—it is unclear how services are priced, and what roles are necessary within such an economy. Should these roles be centrally assigned and managed in the same way as index services are being used today, or can they be distributed across multiple sites? Another closely related issue is the types of relationships that must exist between services within such an economy—for instance, should we be able to support the myriad different financial trading schemes that exist in our markets, and more importantly, what enforcement mechanisms need to be provided to ensure that these trading schemes are being observed.

6. Conclusion. Issues in developing service oriented Grids are outlined. We indicate why agents provide a useful abstraction for managing services in this context, and research challenges that need to be addressed to make more effective use of agents. The need to agree upon common data models/ontologies is significant, and we view this as a significant future undertaking to make Grids more widely deployable. The need for particular application communities to agree and implement common service representations is therefore important—as is the need to agree upon a common ontology for defining generic services. A system for managing user access to scientific instruments is outlined—identifying the services supported and interactions between agents.

REFERENCES

[1] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen Xu, *Peer-to-Peer Computing,* HP Labs, Technical Report HPL-2002-57, 2002.

[2] S. J. Poslad, P. Buckle, and R. Hadingham, *The FIPA-OS Agent Platform: Open Source for Open Standards,* Proceedings of Workshop on Scalability in MAS (Ed: T.Wagner and O.F.Rana), at Autonomous Agents 2000, Barcelona, Spain, 2000.

[3] David De Roure, Nick Jennings, and Nigel Shadbolt, *Semantic Grids,* `http://www.semanticgrid.org/`. Last visited: September 2002.

[4] Luc Moreau, *Agents for the Grid: a Comparison with Web Services (Part I: Transport Layer),* `http://citeseer.nj.nec.com/moreau02agents.html`.

[5] A. Avila-Rosas, L. Moreau, V. Dialani, S. Miles, and X. Liu, *Agents for the Grid: A comparison with Web Services (Part II: Service Discovery),* AgentCities Workshop at AAMAS, Bologna, 2002.

[6] M. Stevens, *Service-Oriented Architecture Introduction, Part 1,* `http://softwaredev.earthweb.com/microsoft/article/0,,10720_1010451_1,00.html`

[7] Ian Foster, Carl Kesselman, Jeffrey M. Nick, Steven Tuecke, *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration,* `http://www.globus.org/research/papers/ogsa.pdf`, 2002.

[8] Michael N. Huhns, *Agents as Web Services,* IEEE Internet Computing, pp 93–95, July/August 2002.

[9] *The TeraGrid Project,* `http://www.teragrid.org/`. Last visited: September 2002.

[10] *The Common Component Architecture Forum,* `http://www.cca-forum.org/`. Last visited: September 2002.

[11] *The Global Grid Forum,* `http://www.gridforum.org/`. Last visited: September 2002.

[12] A. Swartz, *MusicBrainz: A Semantic Web Service,* IEEE Intelligent Systems, pp 76–77, January/February 2002.

[13] *The W3C Web Ontologies Working Group,* `http://www.w3.org/2001/sw/WebOnt/`. Last visited: September 2002.

[14] E. Turcan and R. L. Graham, *Getting the Most from Accountability in P2P,* Proceedings of First International Conference on Peer-to-Peer Computing, IEEE Computer Society Press.

[15] R. Dingledine, M. Freedman, D. Molnar, *The FreeHaven Project,* Massachussets Institute of Technology. `http://www.freehaven.net/`. Last visited: October 2002.

[16] *MMC Virtual Lab: The Materials Microcharacterization Collaboratory Project,* `http://www.ornl.gov/doe2k/mmc/`

[17] W3C, *Semantic Web,* `http://www.w3.org/2001/sw/`. Last visited: September 2002.

[18] *Oak Ridge National Laboratory, The High Temperature Materials Laboratory—User Proposal Package,* http://www.ms.ornl.gov/htmlhome/
[19] H. F. Nielsen, H. Sanders, R. Butek, and S. Nash, *Direct Internet Message Encapsulation (DIME),* June 2002. http://msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-02.txt
[20] *MONET:Mathematics on the Net,* http://monet.nag.co.uk/. Last visited: September 2002.
[21] *Web Services in WSDL,* http://www.xmethods.net/. Last visited: October 2002.
[22] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, *Grid Service Specification,* Open grid Service Infrastructure WG, Global Grid Forum, Draft 2, 7/17/2002. http://www.globus.org/research/papers.html
[23] O. F. Rana, D. Bunford-Jones, D.W. Walker, M. Addis, M. Surridge, and K. Hawick, *Resource Discovery for Dynamic Clusters in Computational Grids,* In *Proceedings of Heterogeneous Computing Workshop*, at IPPS/SPDP, San Francisco, California, April 2001, IEEE Computer Society Press.
[24] *The Gene Ontology Consortium,* http://www.geneontology.org/. Last visited: October 2002.
[25] *MatML Home,* http://www.ceramics.nist.gov/matml/matml.htm. Last visited: October 2002.
[26] Argonne National Laboratory, *The Globus Project,* See Web site at: http://www.globus.org/. Last visited: October 2002.
[27] B. Parvin, J. Taylor, G. Cong, M. O'Keefe, M. Barcellos-Hof, *DeepView: A Channel for Distributed Microscopy and Informatics,* Proceedings of the ACM/IEEE SC99 Conference, Portland, OR, November 1999.
[28] http://vision.lbl.gov/Projects/DeepView/Instruments/Instrument.dtd. Last visited: October 2002.
[29] L. Pouchard, D. Walker, *A Community of Agents for User Support in a Problem-Solving Environment* in Tom Wagner, Omer Rana (Eds.), *Infrastructure for Agents, Multi-Agents Systems, and Scalable Multi-Agent Systems,* Lecture Notes in Computer Science 1887, Springer Verlag, 2001.
[30] R. Buyya, *Economic Paradigm for Resource Management and Scheduling for Service-Oriented Grid Computing,* http://www.buyya.com/ecogrid/. Last visited: October 2002.
[31] Various papers—5^{th} International workshop on *Deception, Fraud and Trust in Agent Societies,* at AAMAS 2002, Bologna, Italy. http://wwwistc.ip.rm.cnr.it/news/wstrust.htm
[32] *The Zeus Project,* See Web site at: http://193.113.209.147/projects/agents.htm, 2000.
[33] D. Davis and M. Parashar, *Latency Performance of SOAP Implementations,* Proceedings of the 2^{nd} International Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems, IEEE International Symposium on Cluster Computing and the Grid, Berlin, Germany, May 2002.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: August 12, 2002
Accepted: December 13, 2002

Scalable Computing: Practice and Experience
Volume 6, Number 4, pp. 95–107. http://www.scpe.org
ISSN 1895-1767

A FEEDBACK CONTROL MECHANISM FOR BALANCING I/O- AND MEMORY-INTENSIVE APPLICATIONS ON CLUSTERS

XIAO QIN* , HONG JIANG† , YIFENG ZHU† , AND DAVID R. SWANSON†

Abstract. One common assumption of existing models of load balancing is that the weights of resources and I/O buffer size are statically configured and cannot be adjusted based on a dynamic workload. Though the static configuration of these parameters performs well in a cluster where the workload can be modeled and predicted, its performance is poor in dynamic systems in which the workload is unknown. In this paper, a new feedback control mechanism is proposed to improve overall performance of a cluster with a general and practical workload including I/O-intensive and memory-intensive load. This mechanism is also shown to be effective in complementing and enhancing the performance of a number of existing dynamic load-balancing schemes. To capture the current and past workload characteristics, the primary objectives of the feedback mechanism are: (1) dynamically adjusting the resource weights, which indicate the significance of the resources, and (2) minimizing the number of page faults for memory-intensive jobs while increasing the utilization of the I/O buffers for I/O-intensive jobs by manipulating the I/O buffer size. Results from extensive trace-driven simulation experiments show that compared with a number of schemes with fixed resource weights and buffer sizes, the feedback control mechanism delivers a performance improvement in terms of the mean slowdown by up to 282% (with an average of 125%).

Key words. Feedback control, I/O-intensive applications, cluster, load balancing

1. Introduction. Scheduling [16, 19] and load balancing [1, 10] techniques in parallel and distributed systems have been investigated to improve system performance with respect to throughput and/or individual response time. Scheduling schemes assign work to machines to achieve better resource utilization, whereas load-balancing policies can migrate a newly arrived job or a running job preemptively to another machines if needed.

Since clusters-a type of loosely coupled parallel system-have become widely used for scientific and commercial applications, several distributed load-balancing schemes in clusters have been presented in the literature, primarily considering CPU [9, 10], memory [1, 23], or a combination of CPU and memory [26, 27]. Although these load-balancing policies have been very effective in increasing the utilization of resources in distributed systems (and thus improving system performance), they have ignored one type of resource, namely disk (and disk I/O). The impact of disk I/O on overall system performance is becoming significant as more and more jobs with high I/O demand are running on clusters. This makes storage devices a likely performance bottleneck. Therefore, we believe that for any dynamic load balancing scheme to be effective in this new application environment, it must be made I/O-aware.

Typical examples of I/O-intensive applications include long running simulations of time-dependent phenomena that periodically generate snapshots of their state [22], archiving of raw and processed remote sensing data [4], multimedia and web-based applications. These applications share a common feature in that their storage and computational requirements are extremely high. Therefore, the high performance of I/O-intensive applications heavily depends on the effective usage of storage, in addition to that of CPU and memory. Compounding the performance impact of I/O in general, and disk I/O in particular, the steady widening gap between CPU and I/O speed makes load imbalance in I/O increasingly more crucial to overall system performance. To bridge this gap, I/O buffers allocated in the main memory have been successfully used to reduce disk I/O costs, thus improving the throughput of I/O systems.

This paper proposes a feedback control mechanism to dynamically configure resource weights and I/O buffers in such a way that the weights are capable of reflecting the significance of system resources, and the memory utilization is improved for I/O- and memory-intensive workload.

The rest of the paper is organized as follows. Related work in the literature is reviewed in Section 2. Section 3 describes system model, and Section 4 proposes the feedback control mechanism. Section 5 evaluates the performance of the mechanism. Finally, Section 6 concludes the paper by summarizing the main contributions and commenting on future directions of this work.

*Department of Computer Science, New Mexico Institute of Mining and Technology, Socorro, New Mexico 87801. http://www.cs.nmt.edu/∼xqin (xqin@cs.nmt.edu). Questions, comments, or corrections to this document may be directed to that email address.

†Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115.

2. Related Work. There exists a large base of excellent research related to distributed load balancing models, and to name just a few: sender or receiver-initiated diffusion [5, 24], the gradient model [6, 13, 14], and the hierarchical balancing model Pollak [24]. Eager et al. studied both receiver and sender initiated diffusion, and the results of their study showed that receiver-initiated policies are preferable at high system loads if the overheads of task transfer under the two policies are comparable [5]. The gradient model makes use of a gradient proximity map of underloaded processors to guide the migration of tasks from overloaded to underloaded processors [6, 13, 14]. Underloaded nodes dynamically update the gradient proximity map, whereas overloaded nodes initiate task migrations. Pollark proposed a scalable approach for dynamic load balancing in large parallel and distributed systems on a multi-level control hierarchy [15]. The hierarchical scheme achieve significant performance gain due to the parallelism in the low level of the hierarchy and the possibility to aggregate information in the higher level of the control tree [15].

The issue of distributed load balancing for CPU and memory resources has been extensively studied and reported in the literature. For example, Harchol-Balter et al. [9] proposed a CPU-based preemptive migration policy that was more effective than non-preemptive migration policies. Zhang et al. [27] focused on load sharing policies that consider both CPU and memory services among the nodes of a cluster. Throughout this paper, the CPU-memory-based load balancing policy presented in [27] will be referred to as CM. The simulation results show that the CM policy not only improves performance of memory-intensive jobs, but also maintains the same load sharing quality of the CPU-based policies for CPU-intensive jobs [27].

A large body of work can be found in the literature that addresses the issue of balancing the load of disk systems [11, 18]. Scheuermann et al. [18] studied two issues in parallel disk systems, namely striping and load balancing, and showed their relationship to response time and throughput. Lee et al. [11] proposed two file assignment algorithms that minimize the variance of the service time at each disk, in addition to balancing the load across all disks. Since the problem of balancing the utilizations across all disks is isomorphic to the multiprocessor scheduling problem [7], a greedy multiprocessor-scheduling algorithm, called LPT [8], can be applied to disk load balancing [11]. Thus, LPT greedily assigns a process to the processor with the lightest I/O load [11]. Throughout this paper, we refer to the approaches that directly apply LPT to I/O load balancing as the IO policy. The I/O load balancing policies in these studies have been shown to be effective in improving overall system performance by fully utilizing the available hard drives.

Very recently, three load balancing models, which consider I/O, CPU and memory resources simultaneously, were presented [21, 26]. In [21], a dynamic load-balancing scheme, tailored for the specific requirements of the Question/Answer application, was proposed along with a performance analysis of the approach. Xiao et al. proposed effective load sharing strategies by minimizing both CPU idle time and the number of page faults in clusters [26].

However, the load-balancing models presented in [21, 26] are similar in the sense that the weights of system resources and buffer size are statically configured with a dynamical workload. In contrast, the new feedback control mechanism proposed in this study judiciously configures these parameters in accordance with the workload of the cluster. Trace-driven simulations show that, compared with the CM and IO policies, the proposed scheme with a feedback control mechanism significantly enhances the overall performance of a cluster system under both memory-intensive and I/O-intensive workload.

Some work has been done to make use of feedback control mechanisms in operating systems and distributed environments [12, 20]. For example, Steere et al. proposed a scheduling scheme that dynamically adjusts CPU allocation and period of threads using the feedback of an application's rates of progress with respect to its inputs and/or outputs [20]. Li and Nahrstedt studied a feedback control algorithm to support end-to-end QoS in a distributed environment [12]. However, the feedback controls of resource weights and buffer sizes have not been addressed in these works. In contrast, this paper has presented the experimental results that verify the benefits of the proposed feedback control mechanism for both resource weights and buffer sizes in a highly dynamic environment.

3. System Model. We consider the issue of feedback control method to improve the performance of load balancing schemes in a cluster connected by a high-speed network, where each node not only maintains its individual job queue that holds jobs until they finish execution, but also perceives reasonably up-to-date global load information by periodically exchanging load status with other nodes. Jobs arrive at each node dynamically and independently, and share three main resources, namely, CPU, main memory, and disk I/O. It is assumed that a round-robin scheduling (time-sharing) is employed as the CPU scheduling policy [9, 27], and the disk of

each node is modeled as a single M/G/1 queue [11]. Since jobs may be delayed because of waiting in queues (to share resources with other jobs) or being migrated to remote nodes, the slowdown imposed on a job u is defined as below,

$$slowdown(u) = \frac{t_f(u) - t_a(u)}{t_{CPU}(u) + t_{IO}(u)} \tag{3.1}$$

where $t_f(u)$ and $t_a(u)$ are the finish and arrival times of the job, and $t_{CPU}(u)$ and $time_{IO}(u)$ are the times spent by job u on CPU and I/O, respectively, without any resource sharing.

In expression 3.1, the numerator corresponds to the total time the job spends running, accessing I/O, waiting, or migrating, and the denominator corresponds to the execution time for job u in a dedicated setting. The definition of slowdown is an extension of the one used in [9, 26, 27], where I/O access time is not considered.

For simplicity, we assume that all nodes are homogeneous, having identical computing power, memory capacity, and disk I/O performance characteristics. This simplifying assumption should not restrict the generality of the proposed model, because if a cluster is heterogeneous, the relative load of a given job imposed on a node with high processing capability is less than that imposed on a node with low performance. The proposed scheme may be extended to handle heterogeneous system by incorporating a simple conversion mechanism for relative load [16].

We also assume the network in our model is fully connected and homogenous in the sense that communication delay between any pair of nodes is the same. This simplification of the network is commonly used in many load-balancing models [9, 26, 27]. Additionally, we assume that the input data of each job has been stored on the local disk of the node to which the job is submitted. This assumption is conservative in nature, since we conducted an experiment to show that, under I/O-intensive workload, the performance of the proposed schemes with such assumption is approximately 10% less effective than that of the schemes without it.

For a newly arrived job u at a node i, load balancing schemes attempt to ship it to a remote node with the lightest load if node i is heavily loaded, otherwise job u is admitted into node i and executed locally. To avoid useless migration that may potentially degrade the system performance, the load balancing schemes consider transferring a job only if the load discrepancy between the source node and the destination node is greater than the load of the newly arrived job plus the migration cost, therefore guaranteeing that each migration improves the expected slowdown of the job. If an appropriate candidate remote node is not available or the migration is evaluated to be useless, the load balancing schemes will not initiate the job migration.

4. Adaptive Load Balancing Scheme.

4.1. Weighted Average Load-balancing Scheme. In this section, we present WAL, a weighted average load-balancing scheme. Each job is described by its requirements for CPU, memory, and I/O, which are measured by the number of jobs running in the nodes, Mbytes, and number of disk accesses per ms, respectively. For a newly arrived job u at a node i, the WAL-FC scheme balances the system load in the following five steps.

1. First, the load of node i is updated by adding job u's load, assigning the newborn job to the local node.
2. Second, a migration is to be initiated if node i's load is overloaded. Node i is overloaded, if: (1) its load is the highest; and (2) the ratio between its load and the average load across the system is greater than a threshold, which is set to 1.25 in our experiments. This optimal value, which is consistent with the result reported in [25], is obtained from an experiment where the threshold is varied from 1.0 to 2.0.
3. Third, a candidate node j with the lowest load is chosen. In the case where there are more than two nodes with the lowest load, we randomly select one node to break the tie. If a candidate node is not available, WAL-FC will be terminated and no migration will be carried out.
4. Fourth, WAL-FC determines if job u is eligible for migration. A job is eligible for migration if its migration is able to potentially reduce the job's slowdown.
5. Finally, job u is migrated to the remote node j, and the load of nodes i and j is updated in accordance with job u's load.

WAL-FC calculates the weighted average load index in the first step. The load index of each node i is defined as the weighted average of CPU and I/O load, thus:

$$load(i) = W_{CPU} \times load_{CPU}(i) + W_{IO} \times load_{IO}(i), \tag{4.1}$$

where $load_{CPU}(i)$ is CPU load defined as the number of running jobs and $load_{IO}(i)$ is the I/O load defined as the summation of the individual implicit and explicit I/O load contributed by jobs assigned to

node i. W_{CPU} and W_{IO} are resource weights used to indicate the significance of the corresponding resource.

It is noted that the memory load is expressed by the implicit I/O load imposed by page faults. Let $l_{page}(i,u)$ and $l_{IO}(i,u)$ denote the implicit and explicit I/O load of job u assigned to node i, respectively. $load_{IO}(i)$ can be defined by equation 4.2, where M_i is a set of jobs running on node i:

$$load_{IO}(i) = \sum_{u \in M_i} l_{page}(i,u) + \sum_{u \in M_i} l_{IO}(i,u). \tag{4.2}$$

Let $r_{MEM}(u)$ denote the memory space requested by job u, and $n_{MEM}(i)$ represent the memory space in bytes that is available to all jobs running on node i. It is to be noted that the memory space, $n_{MEM}(i)$, can be configured in accordance with the buffer size that is adaptively tuned by the feedback control mechanism proposed in Section 4.2. When the node's available memory space is larger than or equal to the memory demand, there is no implicit I/O load imposed on the disk. Conversely, when the memory space of a node is unable to meet the memory requirements of the jobs, the node encounters a large number of page faults, leading to a high implicit I/O load. Implicit I/O load depends on three factors, namely, the available user memory space, the page fault rate, and the memory space requested by the jobs assigned to node i. More precisely, $l_{page}(i,u)$ can be defined as follows, where μ_i denotes the page fault rate of the node, and $load_{MEM}(i)$ is the memory load denoted as the sum of the memory requirements of the jobs running on node i.

$$l_{page}(i,u) = \begin{cases} 0 & \text{if } load_{MEM}(i) \le n_{MEM}(i), \\ \frac{\mu_i \times \sum_{v \in M_i} r_{MEM}(v)}{n_{MEM}(i)} & \text{otherwise.} \end{cases} \tag{4.3}$$

$l_{IO}(i,u)$ in Equation 4.2 is a function of I/O access rate, denoted λ_u), and I/O buffer hit rate $h(i,u)$ that will be discussed in Section 4.1. Thus, $l_{IO}(i,u)$ is approximated by the following expression:

$$l_{IO}(i,u) = \lambda_u \times (1 - h(i,u)). \tag{4.4}$$

In what follows, we quantitatively determine whether a job is eligible for migration. When a job u is assigned to node i, its expected response time $r(i,u)$ can be computed in Equation 4.5.

$$r(i,u) = t_u \times E(L_i) + t_u \times \lambda_u \times E(s^i_{disk} + \frac{\Lambda^i_{disk} \times E((s^i_{disk})^2)}{2(1-\rho^i_{disk})}), \tag{4.5}$$

where t_u and λ_u are the computation time and I/O access rate of job u, respectively. $E(s^i_{disk})$ and $E((s^i_{disk})^2)$ are the mean and mean-square I/O service time in node i, and ρ^i_{disk} is the utilization of the disk in node i. $E(L_i)$ represents the mean CPU queue length L_i, and Λ^i_{disk} denotes the aggregate I/O access rate in node i. Since the expected response time of an eligible migrant on the source node has to be greater than the sum of its expected response time on the destination node and the migration cost, job u is eligible for migration if:

$$r(i,u) > r(j,u) + c_u, \tag{4.6}$$

where j represents a destination node, and c_u is the migration cost (time) modeled as follows,

$$c_u = e + d_u \times (\frac{1}{b^{ij}_{net}} + \frac{1}{b^i_{disk}} + \frac{1}{b^j_{disk}}), \tag{4.7}$$

where e is the fixed cost of migrating the job and loading it into the memory on another node, b^{ij}_{net} denotes the available bandwidth of the network link between node i and j, b^i_{disk} is the available disk bandwidth in node i. In practice, b^{ij}_{net} and b^j_{disk} can be measured by a performance monitor [3]. Accordingly, the simulator discussed in Section 5 estimates b^{ij}_{net} and b^j_{disk} by storing the most recent values of the disk and network bandwidth. d_u represents the amount of data initially stored on disk to be processed by job u. Thus, the second term on the right hand side of Equation 4.7 represents the migration time spent on transmitting data over the network and on accessing source and destination disks.

4.2. Problem Description and Examples. The feedback control mechanism that aims at minimizing the mean slowdown focuses on adjusting the resource weights and the buffer sizes. To help describe the problem of fixed resource weights and I/O buffer sizes, we first present the following examples that motivate the proposed solution to improve the system performance.

Assume a cluster with six identical nodes [9, 17, 26, 27], to which the IO load-balancing policy is applied. The average page-fault rate and I/O access rate are chosen to be 2.0 No./ms (Number/Millisecond) and 2.8 No./ms, respectively. The total memory size for each node is 640 Mbyte, and other parameters of the cluster are given in Section 5.1. We modified the traces used in [9, 27], adding a randomly generated I/O access rate to each job. The traces used in [9] have been collected from one workstation on six different time intervals. In the traces used in our experiments, the CPU and memory demands remain unchanged, and the memory demand of each job is chosen based on a Pareto distribution with the mean size of 4 Mbytes [27].

To evaluate the impact of resource weights (see Equation 4.1) on the system performance, we conducted a simulation experiment where the resource weights were statically set. Figure 4.1 plots the relationship between the resource weight of I/O and the mean slowdown experienced by all the jobs in the trace. The result indicates that the mean slowdown consistently decreases as the I/O resource weight increases from 0 to 1 with increments of 0.2. We attributed this observation to the fact that, under I/O-intensive workload conditions, the I/O resource weight with a high value is able to accurately reflect the significance of the disk I/O resources in the system.

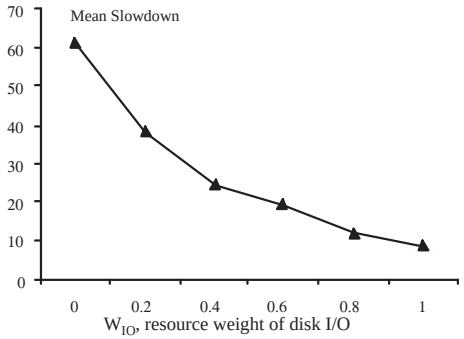


FIG. 4.1. *Mean slowdowns as a function of the I/O resource weight. Average page-fault rate* = 2.0*No./ms, average I/O access rate* = 2.8 *No./ms.*

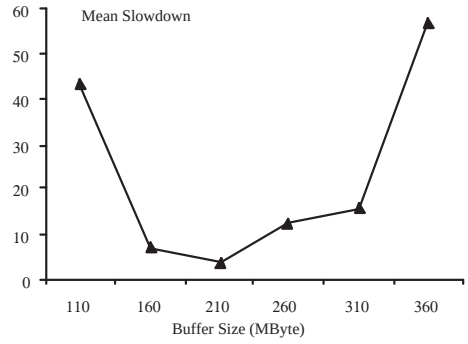


FIG. 4.2. *Mean slowdowns as a function of the buffer size. Average page-fault rate is* 5.0 *No./ms, average I/O access rate is* 2.3*No./ms*

The memory of each node is divided into two portions, with one serving as I/O buffer and the other being used to store working sets of running jobs. Without loss of generality, we assume that the buffer sizes of six nodes are identical. We conducted a second experiment, in which the buffer sizes were statically configured. Figure 4.2 shows the buffer size chosen in the experiment and the corresponding mean slowdowns obtained from the simulator.

The curve in Figure 4.2 reveals that the buffer size has a large effect on the mean slowdowns of the IO-aware policy. When buffer size is smaller than 210 MByte, the slowdown decreases with the increasing value of the buffer size. In contrast, the slowdown increases as the buffer size increases if the buffer size is greater than 210 MByte. Optimally, the mean slowdown of this given workload reaches the minimum value when buffer size is 210 MByte. A large buffer size results in a high buffer hit rate and reduces I/O processing time, thereby causing a positive effect on the performance. On the other hand, given a fixed value of the total available main memory size, a larger buffer size implies a smaller the amount of memory used to store the working sets of running jobs, which in turn leads to a larger number of page faults. In general, a large buffer size may introduce both positive and negative effect on the mean slowdown at the same time, and the overall performance depends on the resultant effect.

Although the static configuration of resource weights and buffer sizes is an approach to tuning the performance of clusters where workload conditions can be modeled and predicted, this approach performs poorly and inefficiently for highly dynamic environments where workloads are unknown at compile time. Therefore, a feedback control algorithm is developed in this study to adaptively configure resource weights and buffer sizes.

4.3. A Feedback Control Mechanism. The high level view of the architecture for the feedback control mechanism is presented in Figure 4.3, where the architecture comprises a load-balancing scheme, a resources-sharing controller, and a feedback controller. The resource-sharing controller consists of a CPU scheduler, a memory allocator and an I/O controller. The slowdown of a newly completed job and the history slowdowns are fed back to the feedback controller, which then determines the required control action ΔW_{IO} and $\Delta bufsize$. $\Delta W_{IO} > 0$ means the IO-weight needs to be increased, and otherwise the IO-weight should be decreased. Since the sum of W_{CPU} and W_{IO} is 1, the control action ΔW_{CPU} can be obtained accordingly. Similarly, $\Delta bufsize > 0$ means the buffer size needs to be increased, and otherwise the buffer size is to be decreased.

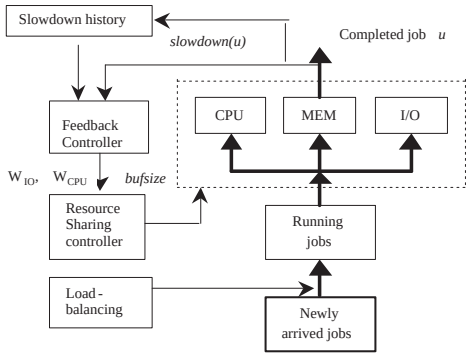


FIG. 4.3. *Architecture of the feedback control mechanism*

The first goal of the feedback controller is to manipulate the resource weights in a way that makes it possible to minimize the mean slowdown of jobs. The system model for an open loop balancer is approximately given by the following equation,

$$slowdown(z) = -wg(L)W_{IO}(z) + wd(L), \tag{4.8}$$

where $wg(L)$ and $wd(L)$ are the gain factor and disturbance factor of the I/O resource weight under workload L, respectively. The values of wg and wd largely depend on workload conditions and the applied load-balancing policy. Thus, wg and wd can be obtained based on simulation models for open-loop load balancers. The control rule for the resource weight is formally modeled below,

$$\Delta W_{IO,u} = G_w(1 - \frac{\overline{S_u}}{\overline{S_{u-1}}})\frac{\Delta W_{IO,u-1}}{|\Delta W_{IO,u-1}|}, \tag{4.9}$$

$$W_{IO,u} = W_{IO,u-1} + \Delta W_{IO}, \tag{4.10}$$

where $\Delta W_{IO,u}$ is the control action, $\overline{S_u}$ denotes the average slowdown, $\frac{\Delta W_{IO,u-1}}{|\Delta W_{IO,u-1}|}$ indicates whether the previous control action has increased or decreased the resource weight, and G_w denotes the controller gain for the I/O resource weight. In the experiments presented shortly in the next section, G_w is tuned to be 0.5 for better performance. Let $W_{IO,u}$ be the resource weight upon the arrival of job at the system, the resource weight will be updated to $W_{IO,u-1} + \Delta W_{IO}$. Without loss of generality, we make use of a linear model to capture the characteristics of varying workload conditions. The model is given by the following equation,

$$slowdown(z) = -wg_0(L)W_{IO}(z) + wd_0 + \Delta wd, \tag{4.11}$$

The feedback controller attempts to manipulate the resource weights in the following three steps. First, when a job u is accomplished, the controller calculates the slowdown s_u of this newly completed job, Second, s_u is stored in the slowdown history table, and the average slowdown $\overline{S_u}$ is computed accordingly. Note that $\overline{S_u}$ reflects a specific pattern of the recent slowdowns in the dynamic workload. The table size is a tunable parameter, and the oldest slowdown will be replaced by the latest one if the history table overflows. In our simulation model presented in Section 5.1, the history table size is fixed to 50. Finally, the controller generates

control actions $\Delta W_{IO,u}$ and $\Delta W_{CPU,u}$, which are based on the previous control action along with the comparison between $\overline{S_u}$ and $\overline{S_{u-1}}$. More precisely, the performance is regarded to be improved by the previous control action if $\overline{S_{u-1}} > \overline{S_u}$, therefore the controller continues increasing W_{IO} if it has been increased by the previous control action, otherwise W_{IO} is decreased. Similarly, $\overline{S_{u-1}} < \overline{S_u}$ means that the performance has been worsened since the latest control action, suggesting that W_{IO} has to be increased if the previous control action has reduced W_{IO}, and vice versa.

Besides configuring the weights, the second goal of the feedback control mechanism is to dynamically set the buffer size of each node based on the unpredictable workload. The mechanism is aiming at improving buffer utilizations and reducing the number of page faults by maintaining an effective usage of memory space for running jobs and their data.

We can derive the slowdown based on a model that captures the correlation between the buffer size and the slowdown. For simplicity, the model can be constructed as follows,

$$slowdown(z) = -bg(L)bufsize(z) + bd(L), \tag{4.12}$$

where $bg(L)$ and $bd(L)$ are the buffer size gain factor and disturbance factor under workload L, respectively. The control rule for buffer sizes is formulated as,

$$\Delta bufsize_u = G_b(\overline{S_{u-1}} - \overline{S_u})\frac{\Delta bufsize_{u-1}}{|\Delta bufsize_{u-1}|}, \tag{4.13}$$

where $\Delta bufsize_u$ is the control action, $\frac{\Delta bufsize_{u-1}}{|\Delta bufsize_{u-1}|}$ indicates whether the previous control action has increased or decreased the resource weight, and C_b denotes the controller gain. G_w is tuned to be 0.5 in order to deliver better performance. Let $bufsize_{u-1}$ be the current buffer size, the buffer size is calculated as $bufsize_u = bufsize_{u-1} + \Delta bufsize_u$.

As can be seen from 4.3, the feedback control generates control action $\Delta bufsize$ in addition to ΔW_{CPU} and ΔW_{IO}. The adaptive buffer size makes noticeable impacts on both the memory allocator and I/O controller, which in turn affect the overall performance (See Figure 4.2). The feedback controller generates a control action $\Delta bufsize$ based on the previous control action along with the comparison between $\overline{S_u}$ and $\overline{S_{u-1}}$. Specifically, $\overline{S_{u-1}} > \overline{S_u}$, means the performance is improved by the previous control action, thereby increasing the buffer size if it has been increased by the previous control action, otherwise the buffer size is reduced. Likewise, $\overline{S_{u-1}} < \overline{S_u}$, indicates that the latest buffer control action leads to a worse performance, implying that the buffer size has to be increased if the previous control action has reduced the buffer size, otherwise the controller decreases the buffer size.

The extra time spent in performing feedback control is negligible and, therefore, the overhead introduced by the feedback control mechanism is ignored in our simulation experiments. The reason is because the complexity of the mechanism is low, and it takes a constant time to make a feedback control decision.

5. Experiments and Results. To evaluate the performance of the proposed load-balancing scheme with a feedback control mechanism, we have conducted a trace-driven simulation, in which the performance metric used is slowdown that is defined earlier in section 3. We have evaluated the performance of the following load-balancing policies:

1. CM: the CPU-memory-based load-balancing policy [27] without using buffer feedback controller. If the memory is imbalanced, CM assigns the newly arrived job to the node that has the least accumulated memory load. When CPU load is imbalance and memory load is well balanced, CM attempts to balance CPU load.
2. IO: the IO-based policy [11] without using the feedback control mechanism. The IO policy uses a load index that represents only the I/O load. For a job arriving in node i, the IO scheme greedily assigns the job to the node that has the least accumulated I/O load.
3. WAL: the Weighted Average Load-balancing scheme without the feedback controller [21].
4. WAL-FC: the Weighted Average Load-balancing scheme with the feedback control mechanism.
5. NLB: The non-load-balancing policy without using the feedback controller.

5.1. Simulation Model. Before presenting the empirical results, the trace-driven simulation model and the workload are presented.

To study dynamic load balancing, Harchol-Balter and Downey [9] implemented a trace-driven simulator for a distributed system with 6 nodes in which round-robin scheduling is employed. The load balancing policy studied in that simulator is CPU-based. Zhang et. al [27] extended the simulator, incorporating memory recourses into the simulation system. Based on the simulator presented in [27], our simulator incorporates the following new features: (1) The above polices are implemented in the simulator. (2) The interconnect is assumed to be a fully connected network. (3) A simple disk model is added into the simulator. (4) An I/O buffer model, which will be presented shortly in this section, is implemented on top of the disk model. The traces used in the simulation are modified from [9][27], and it is assumed that the I/O access rate is randomly chosen in accordance with a uniform distribution. We assume that the I/O access rate of each job is independent of the job's memory space requirement and CPU service time. Although this simplification deflates any correlations between I/O requirement and other job characteristics, we can examine the impact of I/O requirement on system performance by configuring the mean I/O access rate as a workload parameter.

The simulated system is configured with parameters listed in Table 5.1. The parameters for CPU, memory, disks, and network are chosen in such a way that they resemble a typical cluster of the current day.

TABLE 5.1
Data Characteristics

Parameters	**Value**	**Parameters**	**Value**
CPU Speed	800 MIPS	Page Fault Service Time	8.1 ms
RAM Size	640 MByte	Seek and Rotation time	8.0 ms
Initial Buffer Size	160 MByte	Disk Transfer Rate	40MB/Sec.
Context switch time	0.1 ms	Network Bandwidth	1Gbps

Disk accesses of each job are modeled as a Poisson process. Data sizes d_u^{RW} of the I/O requests in job u are randomly generated based on a Gamma distribution with the mean size of 250 KByte and the standard deviation of 50 Kbyte. The sizes chosen in this way reflect typical data characteristics for MPEG-1 data [2], which is retrieved by many multimedia applications.

Since buffer can be used to reduce the disk I/O access frequency (See Equation 4.4), we approximately model the buffer hit probability of I/O access for job u running on node i by the following formula:

$$h(i,u) = \begin{cases} \frac{r_u}{r_u+1} & \text{if } d_{buf}(i,u) \geq d_{data}(u), \\ \frac{r_u}{r_u+1} \times \frac{d_{buf}(i,u)}{d_{data}(u)} & \text{otherwise,} \end{cases} \tag{5.1}$$

where r_u is the data re-access rate, $d_{buf}(i,u)$ is the buffer size allocated to job u, and $d_{data}(u)$ is the amount of data job u retrieves from or stored to the disk, given a buffer with infinite size. I/O buffer in a node is a resource shared by multiple jobs in the node, and the buffer size a job can obtain in node i at run time heavily depends on the jobs' access patterns, characterized by I/O access rate and average data size of I/O accesses. $d_{data}(u)$ linearly depends on access rate, computation time and average data size of I/O accesses d_u^{RW}, and $d_{data}(u)$ is inversely proportional to I/O re-access rate. $d_{buf}(i,u)$ and $d_{data}(u)$ are estimated using the following two equations:

$$d_{buf}(i,u) = \frac{\lambda_u d_u^{RW} d_{buf}(i)}{\sum_{k \in M_i} \lambda_k d_u^{RW}}, \tag{5.2}$$

$$d_{data}(u) = \frac{\lambda_u t_u d_u^{RW}}{r_u + 1}. \tag{5.3}$$

From Equations 5.1, 5.2 and 5.3, hit rate $h(i,u)$ becomes:

$$h(i,u) = \begin{cases} \frac{r_u}{r_u+1} & \text{if } d_{buf}(i,u) \geq d_{data}(u), \\ \frac{r_u d_{buf}(i)}{t_u \sum_{j \in M_i} \lambda_j d_j^{RW}} & \text{otherwise.} \end{cases} \tag{5.4}$$

Figure 5.1 shows the effects of buffer size on the buffer hit probabilities of the NLB, CM and IO policies. When buffer size is smaller than 150 Mbyte, the buffer hit probability increases almost linearly with the buffer

size. The increasing rate of the buffer hit probability drops when the buffer size is greater than 150 Mbyte, suggesting that further increasing the buffer size can not significantly improve the buffer hit probability when the buffer size approaches to a level at which a large portion of the I/O data can be accommodated in the buffer.

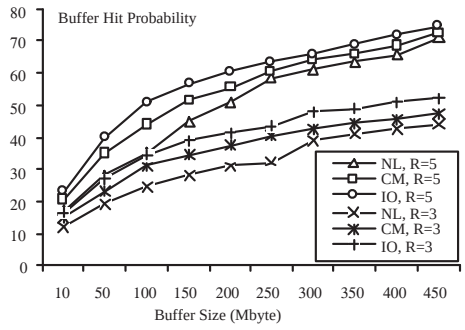


FIG. 5.1. *Buffer Hit Probability as a function of the Buffer Size, page-fault rate is 4.0 No./ms, I/O access rate is 2.2No./ms.*

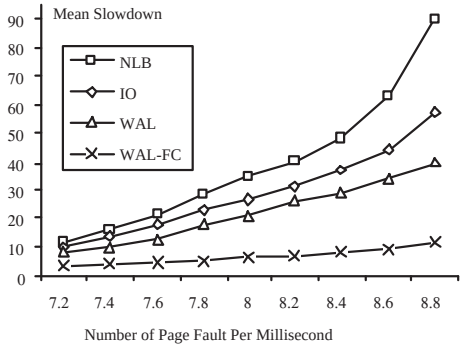


FIG. 5.2. *Mean slowdowns as a function of the page-fault rate, I/O access rate of 0.1 No./ms.*

5.2. Memory Intensive Workload. To simulate a memory intensive workload, the I/O access rate is fixed to a comparatively low level of 0.1 No./ms. The page-fault rate is set from 7.2 No./ms to 8.8 No./ms with increments of 0.2 No./ms. The performance of CM is omitted, since it is very close to that of WAL.

Figure 5.2 reveals that the mean slowdowns of all the policies increase with the page-fault rate. This is because as I/O demands are fixed, high page-fault rate leads to a high utilization of disks, causing longer waiting time on I/O processing.

When the page-fault rate is high, WAL outperforms IO and NLB, and the WAL-FC has better performance than both WAL and IO. For example, the WAL policy reduces slowdowns over the IO policy by up to 37.2% (with an average of 31.5%), and the WAL-FC policy improves the performance in terms of mean slowdown over IO by up to a factor of 4 (400%). The reason is that the IO policy only attempts to balance explicit I/O load, ignoring the implicit I/O load that resulted from page faults. When the explicit I/O load is low, balancing explicit I/O load does not make a significant contribution to balancing the overall system load. In addition, NLB is consistently the worst among the six policies, since NLB leaves three shared resources extremely imbalanced and does not improve the buffer utilization by the adaptive configuration of buffer sizes.

More interestingly, the policies that use the feedback control mechanism algorithm considerably improve the performance over those without employing the feedback controller. For example, WAL-FC improves the system performance over WAL by up to 274% (with an average of 220%). Consequently, the slowdowns of NLB, WAL, and IO are more sensitive to the page-fault rate than WAL-FC.

5.3. I/O-Intensive Workload. To stress the I/O-intensive workload in this experiment, the I/O access rate is fixed at a high value of 2.8 No./ms, and the page-fault rate is chosen from 1.6 No./ms to 2.1 No./ms with increments of 0.1No./ms. The low page-fault rates imply that, even when the requested memory space is larger than the allocated memory space, page faults do not occur frequently. This workload reflects a scenario where memory-intensive jobs exhibit high temporal and spatial locality of access. Figure 5.3 plots slowdown as a function of the page-fault rate. The results of IO are omitted from Figure 5.3, since they are nearly identical to those of WAL.

First, the results show that the WAL scheme significantly outperforms the NLB and CM policies, suggesting that NLB and CM are not suitable for I/O intensive workload. For example, as shown in Figure 5.3, WAL improves the performance of CM in terms of the mean slowdown by up to a factor of 9 (with an average of 476%). This is because the CM policies only balance CPU and memory load, ignoring the imbalanced I/O load of clusters under the I/O intensive workload.

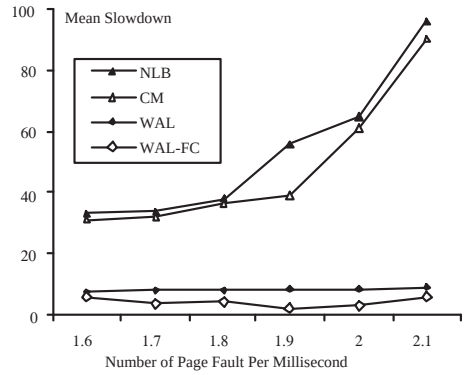


FIG. 5.3. *Mean slowdown as a function of the page-fault rate, I/O access rate is 2.8 No./ms.*

Second, Figure 5.3 shows that WAL-FC significantly outperforms WAL. For example, WAL-FC delivers a performance improvement over WAL by up to 282% (with an average of 125%). Again, this is because the WAL-FC scheme applies the feedback controller to meet the high I/O demands by changing the weights and the I/O buffer sizes to achieve a high buffer hit probability. This result suggests that improving the I/O buffer utilization by using the feedback control mechanism can potentially alleviate the performance degradation resulted from the imbalanced I/O load.

Third, the results further show the slowdowns of NLB and CM are very sensitive to the page-fault rate. In other words, the mean slowdowns of NLB and CM all increase noticeably with the increasing value of I/O load. One reason is, as I/O load are fixed, a high page-fault rate leads to high disk utilization, causing longer waiting time on I/O processing. A second reason is, when the I/O load is imbalanced, the explicit I/O load imposed on some node will be very high, leading to a longer paging fault processing time. Conversely, the page-fault rate makes insignificant impact on the performance of WAL, and WAL-FC. Since the high I/O load imposed on the disks is diminished either by balancing the I/O load or by improving the buffer utilization. This observation suggests that the feedback control mechanism is capable of boosting the performance of clusters under I/O-intensive workload even in the absence of any dynamic load-balancing schemes.

5.4. Memory and I/O intensive Workload. The two previous sections presented the best cases for the proposed scheme since the workload was either highly memory-intensive or I/O-intensive but not both. In these extreme scenarios, the feedback control mechanism provides more benefits to clusters than load-balancing policies do. This section attempts to show another interesting case in which the cluster has a workload with both high memory and I/O intensive jobs. The I/O access rate is set to 1.5 No./ms. The page fault rate is from 7.2 No./ms to 8.4 No./ms with increments of 0.2 No./ms.

Figure 5.4 shows that the performances of CM, IO, and WAL are close to one another. This is because the trace, used in this experiment, comprises a good mixture of memory-intensive and I/O-intensive jobs. Hence, while CM takes advantage of balancing CPU-memory load, IO can enjoy benefits of balancing I/O load. Interestingly, under this specific memory and I/O intensive workload, the resultant effect of balancing CPU-memory load is almost identical to that of balancing I/O load.

A second observation is that, under the memory and I/O intensive workload, load-balancing schemes achieve higher level of improvements over NLB. The reason is that when both memory and I/O demands are high, the buffer sizes in a cluster are unlikely to be changed, as there is a memory contention among memory-intensive and I/O-intensive jobs. Thus, instead of fluctuating widely to optimize the performance, the buffer sizes finally converge to a value that minimizes the mean slowdown.

Third, incorporating the feedback control mechanism in the existing load-balancing schemes is able to further boost the performance. For example, compared with WAL, WAL-FC further decreases the slowdown by up to 54.5% (with an average of 30.3%). This result suggests that, to sustain a high performance in clusters, compounding a feedback controller with an appropriate load-balancing policy is desirable and strongly recommend.

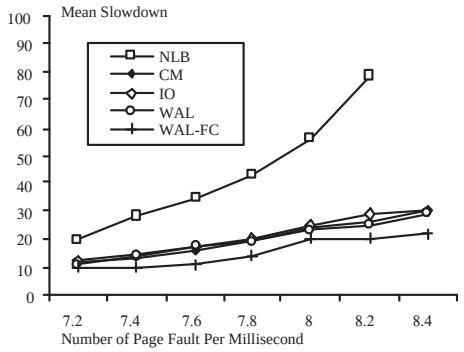


FIG. 5.4. *Mean slowdowns as a function of the page-fault rate, I/O access rate of 1.5 No./ms.*

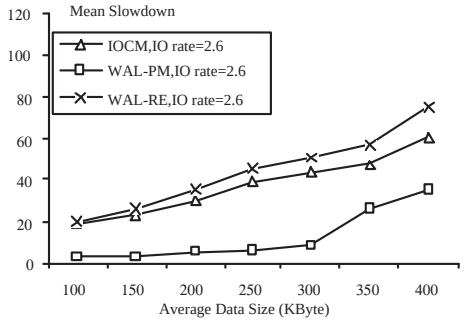


FIG. 5.5. *Mean slowdown as a function of the size of average data size. Page fault rate is 0.5 No./ms, and I/O rate is 2.6 No./ms.*

5.5. Average Data Size. In the previous experiments, the data sizes are chosen based on typical multimedia applications. It is noted that I/O load depends on I/O access rate and the average data size of I/O requests, which in turn rely on the I/O access patterns of applications. The purpose of this experiment is to study the performance improvements achieved by the feedback control mechanism for other types of applications if they exhibit different characteristics. Specifically, Figure 5.5 shows the impact of average data size on the performance of the feedback control mechanism. The page fault rate and the I/O access rate are set to 0.5 No./ms and 2.6 No./ms., respectively. The average data size is chosen from 100 KByte to 400 KByte with increments of 50 KByte.

Figure 5.5 indicates that, for three examined load-balancing policies, the mean slowdown increases as the average data size increases. This is because, under circumstance that both page fault rate and I/O access rate are fixed, a large average data size yields a high utilization of disks, causing longer waiting times on I/O processing. More importantly, Figure 5.5 shows that the performance improvement gained by the feedback control mechanism becomes more noticeable when the average data size is large. This result suggests that the proposed approach is not only beneficial for multimedia applications, but also turns out to be useful for a variety of applications that are data intensive in nature.

6. Conclusions. In this paper, we have proposed a feedback control mechanism to dynamically adjust the weights of recourses and the buffer sizes in a cluster with a general and practical workload that includes memory and I/O intensive workload conditions. The primary objective of the proposed approach is to minimize the number of page faults for memory-intensive jobs while improving the buffer utilization of I/O-intensive jobs. The feedback controller judiciously configures the weights to achieve an optimal performance. Meanwhile, under a workload where the memory demand is high, the buffer sizes are decreased to allocate more memory for memory-intensive jobs, thereby leading to a low page-fault rate.

To evaluate the performance of the mechanism, we compared the proposed WAL-FC scheme with WAL, CM, and IO. For comparison purposes, the NLB policy that does not consider load balancing is also simulated. A trace-driven simulation provides extensive empirical results demonstrating that WAL-FC is effective in enhancing performance of existing dynamic load-balancing policies under memory-intensive or I/O-intensive workload. In particular, when the workload is memory-intensive, WAL-FC reduces the mean slowdown over the CM and IO policies by up to a factor of 9. Further, we have made the following observations:

1. When the page-fault rate is higher and the I/O rate is very low, WAL and CM outperform IO and NLB, and WAL-FC has better performance than WAL;
2. When I/O demands are high, WAL and IO are significantly superior to CM and NLB. And WAL-FC has noticeably better performance than that of IO;
3. Under an I/O intensive workload, the mean slowdowns of NLB and CM all increase noticeably with I/O load. Conversely, the page-fault rate makes insignificant impact on the performance of IO, WAL, and WAL-FC.
4. Under the workload with a good mixture of memory and I/O intensive jobs, WAL-FC achieves high level of improvements over NLB.

5. The performance improvement gained by the feedback control mechanism becomes pronounced when the average data size is relatively large. Future studies in this area may be performed in several directions. First, the feedback control mechanism will be implemented in a cluster system. Second, we will study the stability of the proposed feedback controller. Finally, it will be interesting to study how quickly the feedback controller converges to the optimal value in clusters.

7. Acknowledgements. This work was partially supported by an NSF grant (EPS-0091900), a start-up research fund (103295) from the research and economic development office of the New Mexico Institute of Mining and Technology, a Nebraska University Foundation grant (26-0511-0019), a UNL Academic Program Priorities Grant, and a Chinese NSF 973 project grant (2004cb318201). We are grateful to the anonymous referees for their insightful suggestions and comments.

REFERENCES

[1] A. Acharya and S. Setia, *Availability and Utility of Idle Memory in Workstation Clusters*, Proc. ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, May 1999.
[2] E. Balafoutis, G. Nerjes, P. Muth, M. Paterakis, P. Triantafillou, and G. Weikum, *Clustered Scheduling Algorithms for Mixed-Media Disk Workloads*, Proc. Int'l Conf. on Cluster Computing, 2002.
[3] J. Basney and M. Livny, *Managing Network Resources in Condor*, Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing, Pittsburgh, Pennsylvania, August 2000, pp 298-299.
[4] C.Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, J. Saltz, *Titan: A High-Performance Remote-sensing Database*, Proc. of International Conference on Data Engineering, 1997.
[5] D. Eager, E. Lazowaska, and J. Zahorjan, *A comparison of receiver-initiated and sender-initiated adaptive load sharing*, Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems, Austin, Texas, 1985.
[6] D. J. Evans and Wunbutt, *Load balancing with network partitioning using host groups*, Parallel computing, 20:325-345, March 1994.
[7] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the theory of NP-Completeness*, W.H. Freeman, 1979.
[8] R. L. Graham, *Bounds on Multiprocessing Timing Anomalies*, SIAM J. Applied Math., Vol.17, No.2, pp.416-429, 1969.
[9] M. Harchol-Balter and A. Downey, *Exploiting Process Lifetime Distributions for Load Balacing*, ACM transaction on Computer Systems, vol. 3, no. 31, 1997.
[10] C. Hui and S. Chanson, *Improved Strategies for Dynamic Load Sharing*, IEEE Concurrency, vol.7, no.3, 1999.
[11] L. Lee, P. Scheauermann, and R. Vingralek, *File Assignment in Parallel I/O Systems with Minimal Variance of Service time*, IEEE Trans. on Computers, Vol. 49, No.2, pp.127-140, 2000.
[12] B. Li and K. Nahrstedt, *A Control Theoretical Model for Quality of Service Adaptations*, in IEEE International Workshop on Quality of Service, May 1998.
[13] F.C.H. Lin and R.M. Keller, *The Gradient Model Load Balancing Method*, IEEE Trans. Software Engineering, vol. 13, no. 1, pp. 32-38, Jan. 1987.
[14] F. Muniz and E.J. Zaluska, *Parallel Load Balancing: An Extension to the Gradient Model*, Parallel Computing, vol. 21, pp. 287-301, 1995.
[15] R. Pollak, *A Hierarchical Load Balancing Environment for Parallel and Distributed Supercomputer*, Proc. of the International Symposium on Parallel and Distributed Supercomputing, Fukuoka, Japan, September 1995.
[16] X. Qin, H. Jiang, and D. R. Swanson, *An Efficient Fault-tolerant Scheduling Algorithm for Real-time Tasks with Precedence Constraints in Heterogeneous Systems*, Proc. the 31st Int'l Conf. on Parallel Processing (ICPP 2002), Vancouver, Canada, Aug 2002, pp. 360-368.
[17] X. Qin, H. Jiang, Y. Zhu, and D. Swanson, *Dynamic Load Balancing for I/O-Intensive Tasks on Heterogeneous Clusters*, Proc. of the 10th International Conference on High Performance Computing (HiPC 2003), Dec.17-20, 2003, Hyderabad, India.
[18] P. Scheuermann, G. Weikum, P. Zabback, *Data Partitioning and Load Balancing in Parallel Disk Systems*, The VLDB Journal, pp. 48-66, July, 1998.
[19] H. Shen, S. Lor, and P. Maheshwari, *An architecture-independent graphical tool for automatic contention-free process-to-processor mapping*, Journal of Supercomputing, Vol. 18, No. 2, 2001, p. 115-139.
[20] D. C. Steere, A. Goel, J. Gruenberg, et. al., *A Feedback-driven Proportion Allocator for Real-Rate Scheduling*, Operating Systems Designand Implementation, New Orleans, Louisiana, Feb 1999.
[21] M. Surdeanu, D. Modovan, and S. Harabagiu, *Performance Analysis of a Distributed Question/ Answering System*, IEEE Trans. on Parallel and Distributed Systems, Vol. 13, No. 6, pp. 579-596, 2002.
[22] T. Tanaka, *Configurations of the Solar Wind Flow and Magnetic Field around the Planets with no Magnetic field: Calculation by a new MHD*, Journal of Geophysical Research, pp. 17251-17262, Oct. 1993.
[23] G. Voelker, *Managing Server Load in Global Memory Systems*, Proc. ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, May 1997.
[24] M. Willebeck-LeMair and A. Reeves, *Strategies for Dynamic Load Balancing on Highly Parallel Computers*, IEEE Trans. Parallel and Distributed Systems, vol. 4, no. 9, pp. 979-993, Sept. 1993.
[25] X. Wu, V. Taylor, and R. Stevens, *Design and Implementation of Prophesy Automatic Instrumentation and Data Entry System*, Proc. of the 13th International Conference on Parallel and Distributed Computing and Systems, Anaheim, CA, August 2001.

[26] L. Xiao, S. Chen, and X. Zhang, *Dynamic Cluster Resource Allocations for Jobs with Known and Unknown Memory Demands*, IEEE Transactions on Parallel and Distributed Systems, vol.13, no.3, 2002.
[27] X. Zhang, Y. Qu, and L. Xiao, *Improving Distributed Wrokload Performance by Sharing both CPU and Memory Resources*, Proc. 20th Int'l Conf. Distributed Computing Systems (ICDCS 2000), Apr. 2000.

Edited by: Hong Shen
Received: February 27, 2004
Accepted: June 6, 2004

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_ε 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.