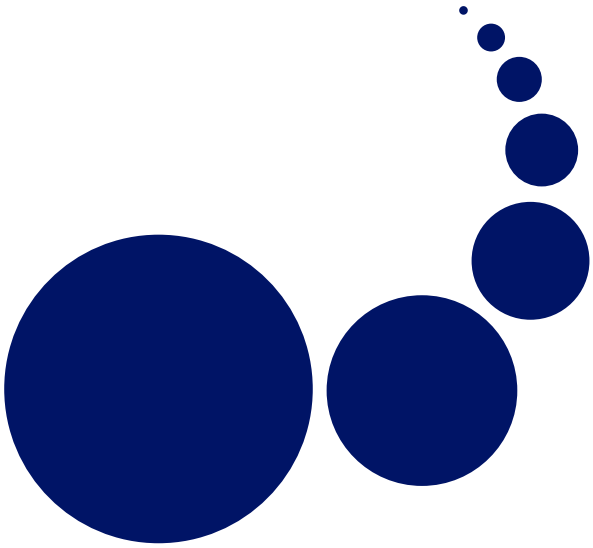
SCALABLE COMPUTING
Practice and Experience

Special Issue: Large Scale Computations on Grids

Editor: Przemysław Stpiczyński



Volume 7, Number 2, June 2006

ISSN 1895-1767

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

Scalable Computing: Practice and Experience

Volume 7, Number 2, June 2006

TABLE OF CONTENTS


SCALABLE COMPUTING: PRACTICE AND EXPERIENCE ISSN 1895-1767
Volume 7, Number 2, p. i. http://www.scpe.org

SPECIAL ISSUE INTRODUCTION: LARGE SCALE COMPUTATIONS ON GRIDS

The first Workshop on Large Scale Computations on Grids (LaSCoG'05) took place in Poznan, Poland, in conjunction with Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM 2005). It was devoted to various aspects of developing, analyzing and executing large-scale applications on computational grids including large-scale algorithms, symbolic and numeric computations, data models for large-scale applications, science portals, data visualization, performance analysis, evaluation and prediction.

The LaSCoG'05 Organizing Committee received nine submissions and after refereeing six of them were accepted for presentation during the Workshop and published in the Proceedings of PPAM 2005 (Lecture Notes in Computer Science vol. 3911). However, these papers were limited to eight pages, thus we decided to publish extended versions of the contributions. Finally, four papers were selected for publication in this special issue of the journal *Scalable Computing: Practice and Experience*:

- A Web Computing Environment for Parallel Algorithms in Java, by O. Bonorden, J. Gehweiler and F. Meyer auf der Heide;
- Mathematical Service Discovery: Architecture, Implementation and Performance, by S. A. Ludwig, O. F. Rana, W. Naylor and J. Padget;
- Benchmarking of a Joint IRISGrid/EGEE Testbed with a Bioinformatics Application, by J. Herrera, E. Huedo, R. S. Montero and I. M. Llorente;
- Heuristic Load Balancing for CFD Codes Executed in Heterogeneous Computing Environments, by D. Petcu, D. Vizman and M. Paprzycki.

Sometimes one can hear (or read) that grid computing is "not enough of a scientific discipline" and practical developments should be widely supported by more theoretical research. The extended versions of the LaSCoG papers selected for publications in this special issue can be considered as fine examples of such theoretical studies. The authors applied some formal methods as research techniques and obtained very interesting results from both practical and theoretical point of view.

Przemysław Stpiczyński
Maria Curie-Sklodowska University
Poland

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 1–14. http://www.scpe.org
ISSN 1895-1767

A WEB COMPUTING ENVIRONMENT FOR PARALLEL ALGORITHMS IN JAVA

OLAF BONORDEN* , JOACHIM GEHWEILER* , AND FRIEDHELM MEYER AUF DER HEIDE*

Abstract. We present a web computing library (PUBWCL) in Java that allows to execute tightly coupled, massively parallel algorithms in the bulk-synchronous (BSP) style on PCs distributed over the internet whose owners are willing to donate their unused computation power. PUBWCL is realized as a peer-to-peer system and features migration and restoration of BSP processes executed on it. The use of Java guarantees a high level of security and makes PUBWCL platform-independent. In order to estimate the loss of efficiency inherent in such a Java-based system, we have compared it to our C-based PUB-Library.

As the unused computation power of the participating PCs is unpredictable, we need novel strategies for load balancing that have no access to future changes of the computation power available for the application. We develop, analyze, and compare different load balancing strategies for PUBWCL. In order to handle the influence of the fluctuating available computation power, we classify the external work load.

During our evaluation of the load balancing algorithms we simulated the external work load in order to have repeatable testing conditions. With the best performing load balancing strategy we could save 39% of the execution time on average and even up to 50% in particular cases, in our test environment.

Key words. Bulk-Synchronous Parallel (BSP) Model, Web Computing, Volunteer-Based Computing, Scheduling, Load Balancing, Fault Tolerance, Java, Thread Migration

1. Introduction. Bearing in mind how many PCs do exist distributed all over the world, one can easily imagine that all their idle times together represent a huge amount of unused computation power. There are already several approaches geared to utilize this unused computation power, for example:

- *distributed.net* [3]
- *Great Internet Mersenne Prime Search (GIMPS)* [7]
- *Search for Extraterrestrial Intelligence (SETI@home)* [18]

A common characteristic of most of these approaches is that the computational problem to be solved has to be divided into many small subproblems by a central server; clients on all the participating PCs download a subproblem, solve it, send the results back to the server, and continue with the next subproblem. Since there is no direct communication between the clients, only independent subproblems can be solved by the clients in parallel.

Our contribution. We have developed a web computing library (PUBWCL) that removes this restriction; in particular, it allows to execute tightly coupled, massively parallel algorithms in the bulk-synchronous (BSP) style on PCs distributed over the internet. PUBWCL is written in Java to guarantee a high level of security and to be platform independent.

When utilizing the unused computation power in a web computing environment, one has to deal with unpredictable fluctuations of the available computation power on the particular computers. Especially in a set of tightly coupled parallel processes, one single process receiving little computation power can slow down the whole application. A way to balance the load is to migrate these "slow" processes. PUBWCL therefore features migration of the BSP processes executed on it. In particular, we have implemented and analyzed four different load balancing strategies. PUBWCL furthermore features restoration of the BSP processes in order to increase fault tolerance.

Related work. Like PUBWCL, *Oxford BSPlib* [8] and *Paderborn University BSP Library (PUB)* [2, 13] are systems to execute tightly coupled, massively parallel algorithms according to the BSP model (see Section 2). They are written in C and are available for several platforms. These BSP libraries are optimized for application on monolithic parallel computers and clusters of workstations. These systems have to be centrally administered, whereas PUBWCL runs on the internet, taking advantage of Java's security model and portability.

The *Bayanihan* BSP implementation [16] follows the master-worker-paradigm: The master decomposes the BSP program to be executed into pieces of work, each consisting of one superstep in one BSP process. The workers download a packet consisting of the process state and the incoming messages, execute the superstep, and send the resulting state together with the outgoing messages back to the master. When the master has received the results of the current superstep for all BSP processes, it moves the messages to their destination

*Heinz Nixdorf Institute, Computer Science Departement, Paderborn University, 33095 Paderborn, Germany, {bono, joge, fmadh}@uni-paderborn.de

packets. Then the workers continue with the next superstep. With this approach all communication between the BSP processes passes though the server, whereas the BSP processes communicate directly in PUBWCL.

In [11], the problem of scheduling BSP processes on idle times of the processors is formalized as an online problem. It is shown that in the worst case, the competitive ratio, i. e., the factor by which the BSP algorithm is executed slower in the online setting, compared to an optimal offline setting, is arbitrarily large. The main contribution are two models that restrict the way how the unused computation power of the system changes over time, and algorithms with very small competitive ratio for these models. Our formalization of external work load in Section 6 is inspired by these results.

Organization of paper. The rest of the paper is organized as follows: In Sections 2 and 3, we give an overview of the used parallel computing model and the Java thread migration mechanism. In Section 4, we describe our web computing library. In Sections 5 and 6, we describe the implemented load balancing strategies and analyze the external work load. In Section 7, we evaluate the performance of our Java-based implementation and discuss the results obtained from experiments with our load balancing strategies. Section 8 concludes this paper.

2. The BSP Model. In order to simplify the development of parallel algorithms, Leslie G. Valiant has introduced the *Bulk-Synchronous Parallel (BSP)* model [20] which forms a bridge between the hardware to use and the software to develop. It gives the developer an abstract view of the technical structure and the communication features of the hardware to use (e. g. a parallel computer, a cluster of workstations or a set of PCs interconnected by the internet).

A *BSP computer* is defined as a set of processors with local memory, interconnected by a communication mechanism (e. g. a network or shared memory) capable of point-to-point communication, and a barrier synchronization mechanism (cf. Fig. 2.1).

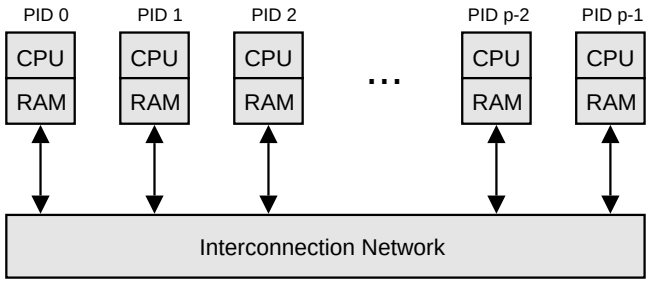


FIG. 2.1. *BSP computer*

A *BSP program* consists of a set of *BSP processes* and a sequence of *supersteps*—time intervals bounded by the barrier synchronization. Within a superstep each process performs local computations and sends messages to other processes; afterwards it indicates by calling the sync method that it is ready for the barrier synchronization. When all processes have invoked the sync method and all messages are delivered, the next superstep begins. Then the messages sent during the previous superstep can be accessed by its recipients. Fig. 3.1 illustrates this.

3. Thread Migration in Java. Complex algorithms often require much computation power, which means that they run for quite a long time, even on a parallel computer. Thus, we have the following problem in a web computing environment: If the owner of one of the PCs, whose unused computation power is donated, needs all his resources himself again, the BSP process running on that machine will take much more time to complete the current superstep. As shown in Fig. 3.1, this will delay the execution of the whole BSP program due to the barrier synchronization.

Thus, the execution time of a parallel program can be significantly improved if it is possible to migrate its processes at run-time to other hosts with currently more available computation power. Different load balancing strategies, that are used to decide when to migrate which BSP processes to which PUBWCL client, are presented in Section 5.

From the operating system's point of view, BSP processes are threads, so we actually need to migrate Java threads. There are three ways how this can be accomplished:

- modification of the Java Virtual Machine (VM) [12],
- bytecode transformations [15, 19],
- sourcecode transformations [17, 4].

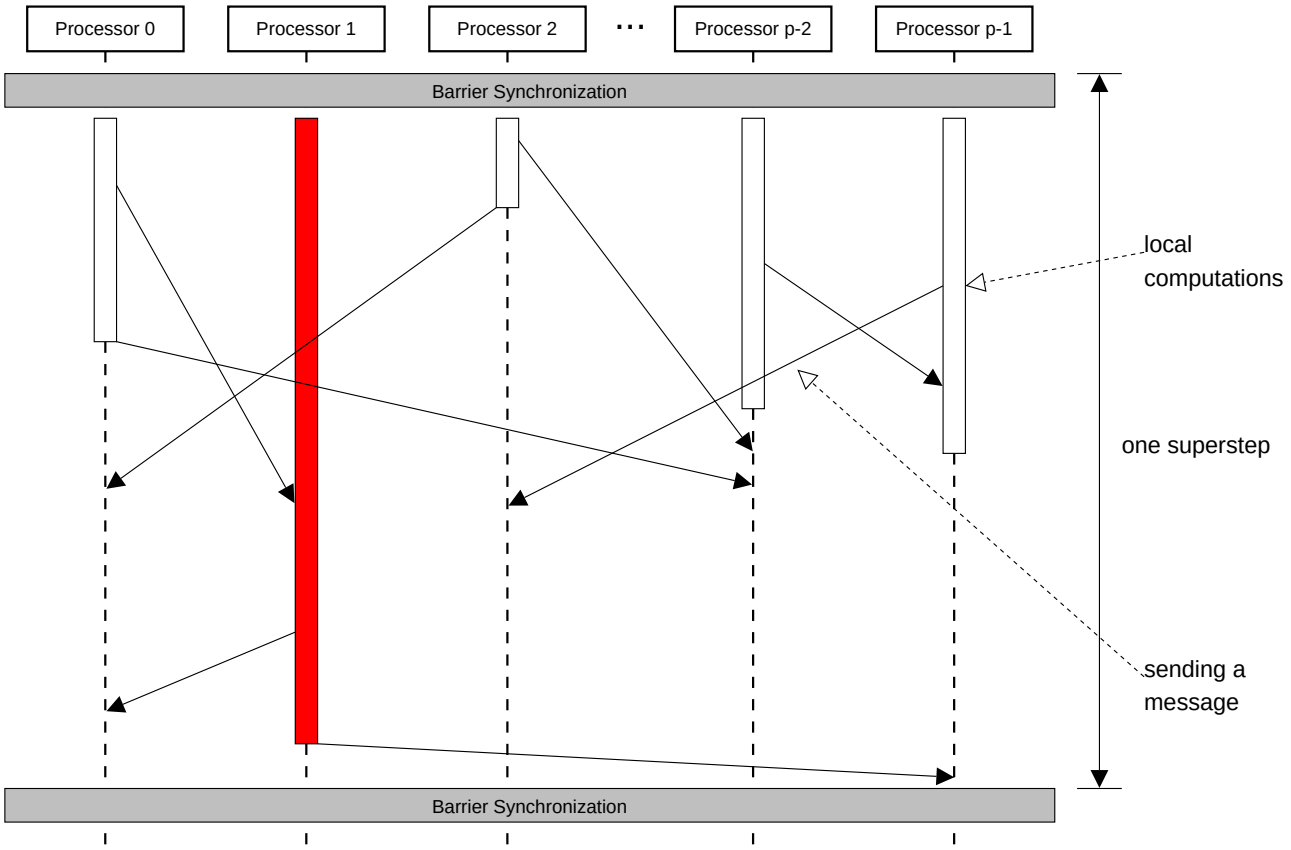


FIG. 3.1. *Delayed synchronization*

Modifying the Java VM is not advisable because everybody would have to replace his installation of the original Java VM with one from a third party, just to run a migratable Java program.

Inside PUBWCL, we use *JavaGo RMI* [17, 10] which is an implementation of the sourcecode transformation approach. It extends the Java programming language with three features:

- Migrations are performed using the keyword `go` (passing a filename instead of a hostname as parameter creates a backup copy of the execution state).
- All methods, inside which a migration may take place, have to be declared `migratory`.
- The depth, up to which the stack will be migrated, can be bounded using the `undock` statement.

The JavaGo compiler *jgoc* translates this extended language into Java sourcecode, using the unfolding technique described in [17]. Migratable programs are executed by dint of the wrapper `javago.Run`. In order to continue the execution of a migratable program, an instance of `javago.BasicServer` has to run on the destination host.

Since the original implementation of JavaGo is not fully compatible with the Java RMI standard, we use our own adapted version JavaGo RMI.

4. The Web Computing Library. People willing to join the *Paderborn University BSP-based Web Computing Library (PUBWCL)* system, have to install a PUBWCL client. With this client, they can donate their unused computation power and also run their own parallel programs.

Architecture of the system. PUBWCL is a hybrid peer-to-peer system: The execution of parallel programs is carried out on peer-to-peer basis, i. e., among the clients assigned to a task. Administrative tasks (e. g. user management) and the scheduling (i. e. assignment of clients and selection of appropriate migration targets), however, are performed on client-server-basis. Clients in private subnets connect to the PUBWCL system via the *proxy* component. The interaction of the components is illustrated in Fig. 4.1.

Since the clients may join or leave the PUBWCL system at any time, the login mechanism is lease-based, i. e., a login session expires after some timeout if the client does not regularly report back at the server.

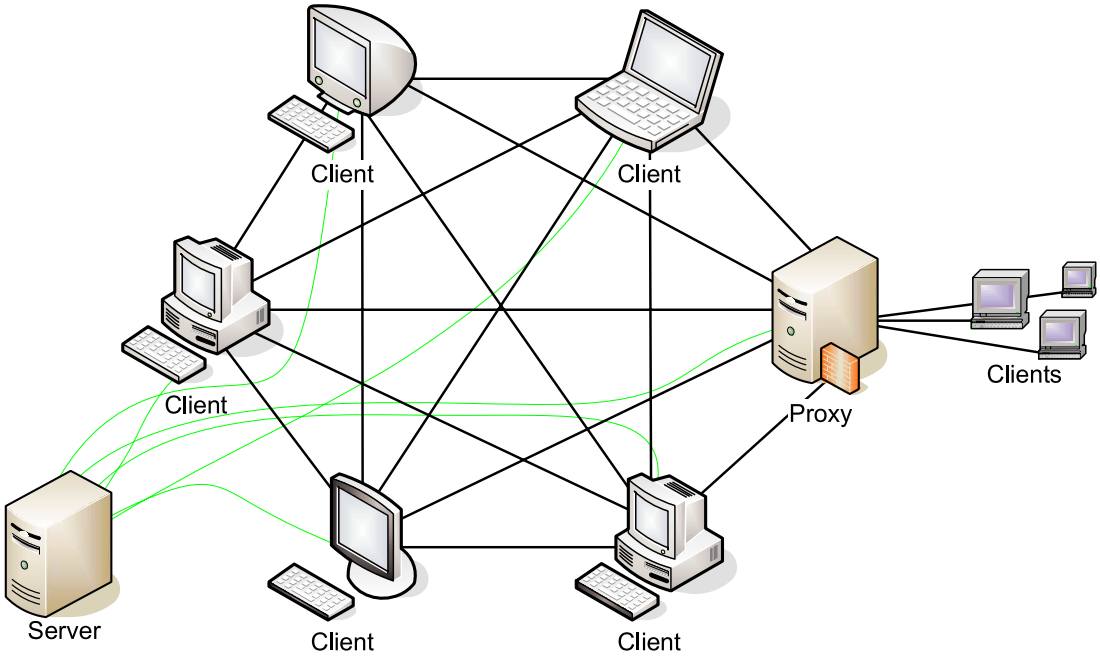


FIG. 4.1. *The architecture of PUBWCL.*

Though a permanent internet connection is required, changes of dynamically assigned IP addresses can be handled. This is accomplished by using *Global Unique Identifiers (GUIDs)* to unambiguously identify the clients: when logging in, each client is assigned a GUID by the server. This GUID can be resolved into the client's current IP address and port.

Executing a parallel program. If users want to execute their own parallel programs, they must be registered PUBWCL users (otherwise, if they only want to donate their unused computation power, it is sufficient to use the guest login). To run a BSP program, it simply has to be copied into a special directory specified in the configuration file. Then one just needs to enter the name of the program and the requested number of parallel processes into a dialog form. Optionally, one may pass command line arguments to the program or choose a certain load balancing algorithm.

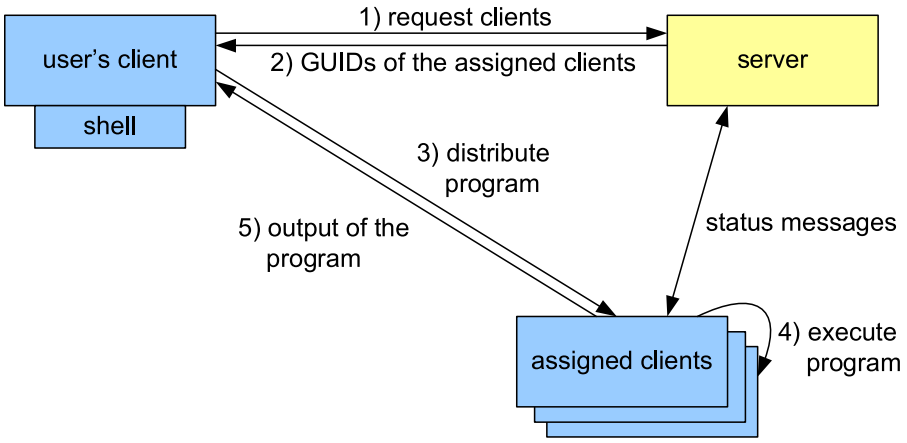


FIG. 4.2. *Executing a BSP program in PUBWCL.*

The server then assigns the parallel program to a set of clients and sends a list of these clients to the user's client (cf. Fig. 4.2). From now on, the execution of the parallel program is supervised by the user's client. On each of the assigned clients a PUBWCL runtime environment is started and the user's parallel program

is obtained via dynamic code downloading. The output of the parallel program and, possibly, error messages including stack traces are forwarded to the user's client.

All processes of parallel programs are executed in an own PUBWCL runtime environment in a separate process, so that it is possible to cleanly abort single parallel processes (e. g. in case of an error in a user program).

Security aspects. The *Java Sandbox* allows to grant code specific permissions depending on its origin. For example, access to (part of) the file system or network can be denied. In order to guarantee a high level of security, we grant user programs only read access to a few Java properties which are needed to write completely platform independent code (e. g. `line.separator` etc.).

Details on the internals of the library as well as a guide how to configure it can be found in [14] and [5].

4.1. The Programming Interface. User programs intended to run on PUBWCL have to be BSP programs ([1] is an excellent guide to parallel scientific compuation using BSP). Thereto the interface `BSPProgram` must be implemented, i. e., the program must have a method with this signature:

```
public void bspMain(BSP bspLib, String[] args)
    throws AbortedException
```

Its first parameter is a reference to the PUBWCL runtime environment which supports the BSP interface; the second parameter is an array containing the command line parameters passed to the BSP program.

In order to write a migratable program, the interface `BSPMigratableProgram` has to be implemented instead, which means that the main method has this different signature:

```
public migratory void bspMain(BSPMigratable bspLib,
    String[] args) throws AbortedException, NotifyGone
```

The following BSP library functions can be accessed via the `BSP` resp. `BSPMigratable` interface which is implemented by the PUBWCL runtime environment:

In non-migratable programs, the barrier synchronization is entered by calling:

```
public void sync()
```

The migratable version additionally creates backup copies of the execution state and performs migrations if suggested by the load balancing strategy:

```
public migratory void syncMig() throws NotifyGone
```

A message, wich can be any serializable Java object, can be sent with these methods; thereby the latter two methods are for broadcasting a message to an interval resp. an arbitrary subset of the BSP processes:

```
public void send(int to, Serializable msg)
    throws IntegrityException
public void send(int pidLow, int pidHigh, Serializable msg)
    throws IntegrityException
public void send(int[] pids, Serializable msg)
    throws IntegrityException
```

Messages sent in the previous superstep can be accessed with these methods, where the `find*` methods are for accessing messages of a specific sender:

```
public int getNumberOfMessages()
public Message getMessage(int index)
    throws IntegrityException
public Message[] getAllMessages()
public Message findMessage(int src, int index)
    throws IntegrityException
public Message[] findAllMessages(int src)
    throws IntegrityException
```

When receiving a message, it is encapsulated in a `Message` object. The message itself as well as the sender ID can get obtained with these methods:

```
public Serializable getContent()
public int getSource()
```

In order to terminate all the processes of a BSP program, e. g. in case of an error, the following method has to be called; the `Throwable` parameter will be transmitted to the PUBWCL user who has started the program:

```
public void abort(Throwable cause)
```

Any output to *stdout* or *stderr* should be printed using the following methods as they display it on the PUBWCL client of the user who has started the program rather than on the computer where the process is actually running:

```
public void printStdOut(String line)
public void printStdErr(String line)
```

To access data from files, the following method should be used. In particular, any file in the BSP program folder of the user's client can be read with it:

```
public InputStream getResourceAsStream(String name)
    throws ChainedException
```

In migratable programs, there is also a method available which may be called to mark additional points inside long supersteps where a migration is safe (i. e. no open files etc.):

```
public migratory boolean mayMigrate() throws NotifyGone
```

Furthermore, there are some service functions to obtain the number of processes of the BSP program, the own process ID, and so on.

Listing 1 shows an example program which demonstrates the basic BSP features, especially how to send and receive messages.

Listing 1
Example program demonstrating message passing.

```
1  import de.upb.sfb376.a1.*;
2  import de.upb.sfb376.a1.bsp.*;
3
4  public class MessagePassing implements BSPProgram
5  {
6    public void bspMain(BSP bsp, String[] args) {
7      int i, left, right;
8      Message msg;
9      Message[] msgs;
10
11     // calculate neighbours
12     left = (bsp.getPID() + bsp.getNumberOfProcessors() - 1) % bsp.getNumberOfProcessors()
         ;
13     right = (bsp.getPID() + 1) % bsp.getNumberOfProcessors();
14
15     try {
16       bsp.send(left, new Integer(1));
17       bsp.send(right, new Integer(2));
18     } catch(IntegrityException ie) {
19       bsp.printStdErr("an error occurred during 'send': `` + ie.getMessage());
20     }
21     bsp.sync();
22
23     // get all messages, method 1
24     for(i=0; i<bsp.getNumberOfMessages(); i++) {
25       try {
26         msg = bsp.getMessage(i);
27         bsp.printStdOut("recieved `` + msg.getContent() + `` from pid `` + msg.getSource
             () + `` in superstep `` + msg.getSuperstep());
28       } catch(IntegrityException ie) {
29         bsp.printStdErr("an error occurred during 'getMessage': `` + ie.getMessage());
30       }
31     }
32
33     // get all messages, method 2
34     msgs = bsp.getAllMessages();
35     bsp.printStdOut("recieved in total `` + msgs.length + `` messages");
36
37     // get messages from some specified pid, method 1
38     try {
39       i = 0;
40       while((msg = bsp.findMessage(0, i++)) != null)
41         bsp.printStdOut("recieved `` + msg.getContent() + `` from pid 0 in superstep `` +
              msg.getSuperstep());
42     } catch(IntegrityException ie) {
43       bsp.printStdErr("an error occurred during 'findMessage': `` + ie.getMessage());
44     }
```

```
45
46      // get messages from some specified pid, method 2
47      try {
48        msgs = bsp.findAllMessages(0);
49        bsp.printStdOut("recieved `` + msgs.length + `` messages from pid 0");
50      } catch(IntegrityException ie) {
51        bsp.printStdErr("an error occurred during 'findAllMessages': `` + ie.getMessage());
52      }
53    }
54  }
```

5. Load Balancing. As already pointed out in Section 3, one single process receiving little computation power can slow down the execution of the whole BSP program due to the barrier synchronization. Migrating these "slow" BSP processes can therefore significantly improve the execution time of the BSP program. Before we present our load balancing algorithms, we need some preliminaries for scheduling.

First of all, we can derive the following constraint from the properties of a BSP algorithm. Since all the BSP processes are synchronized at the end of each superstep, we can reduce the scheduling problem for a BSP algorithm with n supersteps to n subproblems, namely scheduling within a superstep.

Second, we assume that we only have to deal with "good" BSP programs, i. e., all of the p BSP processes require approximately the same amount of computational work. Thus the scheduler has to assign p equally heavy pieces of work.

Finally, we have another restriction. Due to privacy reasons we cannot access the breakdown of the CPU usage, i. e., we especially do not know how much computation power is consumed by the user and how much computation power is currently assigned to our BSP processes.

Parameters for scheduling. Since we cannot directly access the breakdown of the CPU usage, we have to estimate (1) how much computation power the BSP processes currently assigned to a client do receive, and (2) how much computation power a BSP process would receive when (additionally) assigned to a client.

As from the second superstep on, the first question can simply be answered by the ratio of the computation time consumed during the previous superstep and the number of concurrently running BSP processes.

In order to answer the second question, all clients regularly measure the *Available Computation Power (ACP)*. This value is defined as the computation power an additionally started BSP process would receive on a particular client, depending on the currently running processes and the external work load. We obtain this value by regularly starting a short benchmark process and measuring the computation power it receives. Though it is not possible to determine the CPU usage from the ACP value, this value is platform independent and thus comparable among all clients.

Since the first approach is more accurate, we will use it wherever possible, i. e., mainly, to decide whether a BSP process should migrate or has to be restarted. The ACP value will be used to determine the initial distribution of the BSP processes and to choose clients as migration targets and as hosts for restarted BSP processes.

5.1. The load balancing algorithms. We have implemented and analyzed the following four load balancing strategies, among them two parallel algorithms and two sequential ones.

Algorithm PwoR. The load balancing algorithm *Parallel Execution without Restarts (PwoR)* executes all processes of a given BSP program concurrently.

The initial distribution is determined by dint of the ACP values. Whenever a superstep is completed, all clients are checked whether the execution of the BSP processes on them took more than r times the average execution duration (a suitable value for r will be chosen in Section 7.2); in this case the BSP processes are redistributed among the active clients such that the expected execution duration for the next superstep is minimal, using as little migrations as possible.

Algorithm PwR. Using the the load balancing algorithm *Parallel Execution with Restarts (PwR)*, the execution of a superstep is performed in phases. The duration of a phase is r times the running time of the $\lceil s \cdot p^* \rceil$-th fastest of the (remaining) BSP processes, where p^* is the number of processes of the BSP program that have not yet completed the current superstep. Suitable values for the parameters $r > 1$ and $0 < s < 1$ will be chosen in Section 7.2. At the end of a phase, all incomplete BSP processes are aborted. In the next phase, they are restarted on faster clients. Whereas too slow BSP processes are migrated only after the end of the superstep using the PwoR algorithm, they are restarted already during the superstep using PwR.

At the end of each (except the last) superstep, the distribution of the BSP processes is optimized among the set of currently used clients by dint of the processes' execution times in the current superstep. The optimization of the distribution is performed such that the number of migrations is minimal.

Algorithm SwoJ. While the two load balancing strategies PwoR and PwR execute all BSP processes in parallel, the load balancing algorithm *Sequential Execution without Just-in-Time Assignments (SwoJ)* executes only one process of a BSP program per client at a time; the other BSP processes are kept in queues.

Like PwR, SwoJ operates in phases. At the end of a phase all uncompleted BSP processes are aborted and reassigned. Thereby the end of a phase is reached after r times the duration, in which the x-th fastest client has completed all assigned BSP processes, where x is a fraction s of the number of the affected clients. At the end of a superstep, the distribution of the BSP processes is optimized like in PwR.

Algorithm SwJ. Like SwoJ, the load balancing algorithm *Sequential Execution with Just-in-Time Assignments (SwJ)* executes only one process of a BSP program per client at a time and keeps the other processes in queues, too. The main difference, however, is that these queues are being balanced. More precisely, whenever a client has completed the execution of the last BSP process in its queue, a process is migrated to it from the queue of the most overloaded client (if there exists at least one overloaded client). Thereby a client is named "overloaded" in relation to another client if completing all but one BSP processes in the queue with the current execution speed of the particular client would take longer than executing one BSP process on the other client. Thereby the execution speed of a client is estimated by dint of the running time of the BSP process completed on it most recently.

BSP processes are aborted only if the corresponding queues on all affected clients are empty and if the processes do not complete within r times the duration of a process on the x-th fastest client, weighted by the number of BSP processes on the particular clients; thereby, again, x is a fraction s of the number of the affected clients.

6. The External Work Load. In order to understand the fluctuation of the external work load, we have analyzed totaling more than 100 PCs in altogether four departments of three German universities over a period of 21 resp. 28 days. The CPU frequencies varied from 233 MHz to 2.8 GHz. The installed operating systems were Debian Linux, RedHat Enterprise Linux, and SuSE Linux.

While analyzing the load, we have noticed that the CPU usage typically shows a continuous pattern for quite a time, then changes abruptly, then again shows a continuous pattern for some time, and so on. The reason therefore is that many users often perform uniform activities (e. g. word processing, programming, and so on) or no activity (e. g. at night or during lunch break).

A given CPU usage graph (e. g. of the length of a week) can thus be split into blocks, in which the CPU usage is somewhat steady or shows a continuous pattern. These blocks typically have a duration of some hours, but also durations from only half an hour (e. g. lunch break) up to several days (e. g. a weekend) do occur.

Based on the above observations we have designed a model to describe and classify the external work load. We describe the CPU usage in such a block by a rather tight interval with radius $\alpha \in \mathbb{R}$ ($\alpha < \frac{1}{2}$) around a median load value $\lambda \in \mathbb{R}$ ($0 \leq \lambda - \alpha,\ \lambda + \alpha \leq 1$), as illustrated in Fig. 6.1. The rates for the upper and lower deviations are bounded by $\beta^+ \in \mathbb{R}$ resp. $\beta^- \in \mathbb{R}$ ($\beta^+, \beta^- < \frac{1}{2}$). We will refer to such a block as a $(\lambda, \alpha, \beta^+, \beta^-, T)$-*load sequence* in the following.

In order to describe the frequency and duration of the deviations, we subdivide the load sequences into small sections of length T, called *load periods*. The values β^+ and β^- must be chosen such that the deviation rates never exceed them for an arbitrary starting point of a load period within the load sequence.

Let $D \in \mathbb{R}^+$ be the execution duration of a superstep of a BSP process in the case that we receive the full computation power of the used machine. Given that T is much shorter than the duration of a superstep, we can obtain this result:

THEOREM 6.1. *If a superstep of length D of a BSP process is executed completely within a $(\lambda, \alpha, \beta^+, \beta^-, T)$-load sequence, the factor between its minimal and maximal possible duration is at most $q' \in \mathbb{R}^+$ with*

$$q' \leq \delta(D) \cdot \frac{1 - (1 - \beta^-)(\lambda - \alpha)}{1 - (\beta^+ + (1 - \beta^+)(\lambda + \alpha))}$$

where $\delta(D)$ tends to 1 with $D \to \infty$. For $q \in \mathbb{R}^+$, $q \geq q'$ we call a $(\lambda, \alpha, \beta^+, \beta^-, T)$-load sequence q-bounded.

PROOF: Let $d(t) : \mathbb{R}_0^+ \mapsto \mathbb{R}^+$ denote the actually required execution time of the superstep depending on the external load if the execution starts at time $t \in \mathbb{R}_0^+$.

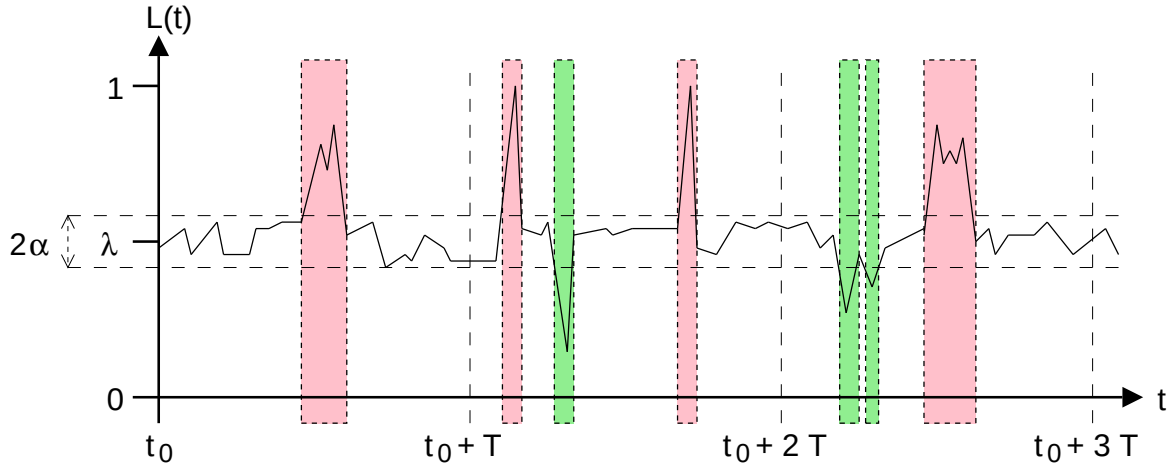


FIG. 6.1. *A three-load-period-long interval of a load sequence.*

First, we show that:

$$\frac{D}{1-(1-\beta^-)(\lambda-\alpha)} - \varepsilon_1 \leq d(t) \leq \frac{D}{1-(\beta^+ + (1-\beta^+)(\lambda+\alpha))} + \varepsilon_2 \tag{6.1}$$

where:

$$\varepsilon_1 = \beta^- T \frac{\lambda-\alpha}{1-(\lambda-\alpha)} \text{ and } \varepsilon_2 = \beta^+ T$$

As we are given that the superstep is executed completely within a $(\lambda, \alpha, \beta^+, \beta^-, T)$-load sequence, the external work load is bounded by:

$$\ell_{\min} := \beta^- \cdot 0 + (1-\beta^-)(\lambda-\alpha) \tag{6.2}$$

resp.

$$\ell_{\max} := \beta^+ \cdot 1 + (1-\beta^+)(\lambda+\alpha) \tag{6.3}$$

Dividing the execution duration D by these bounds, we get (6.1). It remains to show that our estimations for the corrections values ε_1 and ε_2 hold. These values are needed because $d(t)$ is not necessarily a multiple of T.

Since a load period may begin at any point within a load sequence, we can assume w.l.o.g. that the execution of the superstep starts at the beginning of a load period. Thus, we only have a fractionally utilized load period at the end of the superstep.

Let $x \in \mathbb{R}$, $0 < x < 1$, be the fraction to which this load period is used. If there were no correction values, this would mean that at most a fraction x of the deviations in the load period would affect the execution of the superstep. But this is not necessarily the case as the deviations can occur at arbitrary points within the load period by definition. Thus, we have to choose ε_1 and ε_2 such that an arbitrary amount of the deviations can interfere with the execution of the superstep.

Fig. 6.2 shows the influences of the upper deviations (the gray area is the computation power that the BSP process receives): If all the upper deviations occur in the first part of the load period, the end of the execution is delayed by up to $\varepsilon_2 = \beta^+ T$.[1]

In order to estimate the influences of the lower deviations, have a look at Fig. 6.3: The gray areas again are the computation power that the BSP process receives. The dark gray rectangle of case a) has been cut off in case b) and been replaced by a rectangle with the same area in the lower deviation. Thus, the execution duration

[1]This estimation actually is a bit too pessimistic as a fraction x of the deviations is already contained in (6.1) without the correction values.

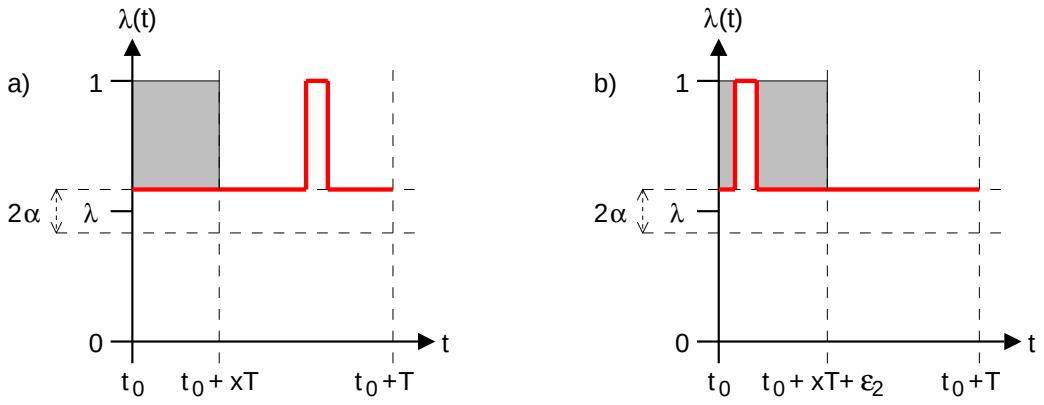


FIG. 6.2. *The influences of the upper deviations in the last load period.*

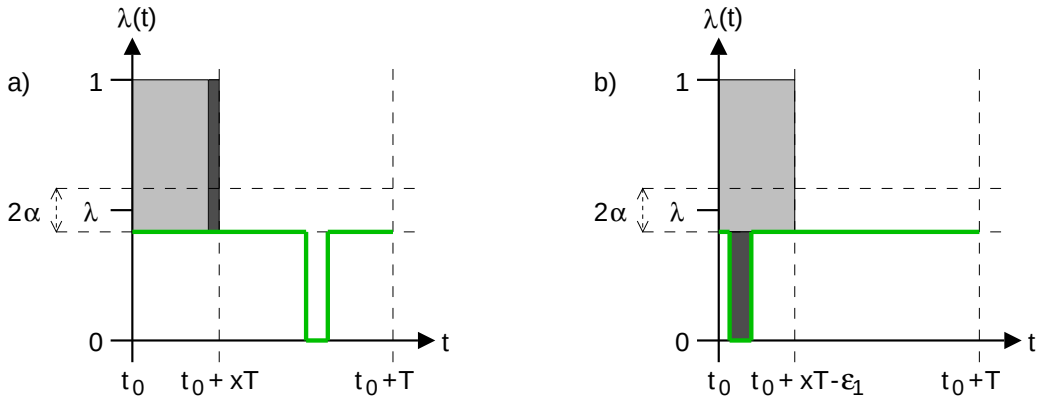


FIG. 6.3. *The influences of the lower deviations in the last load period.*

in case b) is shortened by the width of the dark gray rectangle of case a), which can be obtained by dividing the area of the dark gray rectangle of case b) by the height of the one of case a), i. e., $\varepsilon_1 = \beta^- T \frac{\lambda-\alpha}{1-(\lambda-\alpha)}$.[1]

Now we obtain the following estimation from (6.1) for some $q' \in \mathbb{R}^+$:

$$\frac{D}{1-(\beta^+ + (1-\beta^+)(\lambda+\alpha))} + \varepsilon_2 = q' \left(\frac{D}{1-(1-\beta^-)(\lambda-\alpha)} - \varepsilon_1 \right)$$

$$\Leftrightarrow \frac{D + \varepsilon_2(1-(\beta^+ + (1-\beta^+)(\lambda+\alpha)))}{1-(\beta^+ + (1-\beta^+)(\lambda+\alpha))} = q' \cdot \frac{D - \varepsilon_1(1-(1-\beta^-)(\lambda-\alpha))}{1-(1-\beta^-)(\lambda-\alpha)}$$

$$\Leftrightarrow q' = \frac{1-(1-\beta^-)(\lambda-\alpha)}{1-(\beta^+ + (1-\beta^+)(\lambda+\alpha))} \cdot \frac{D + \varepsilon_2(1-(\beta^+ + (1-\beta^+)(\lambda+\alpha)))}{D - \varepsilon_1(1-(1-\beta^-)(\lambda-\alpha))}$$

Now we define:

$$\delta(D) := \frac{D + \varepsilon_2(1-(\beta^+ + (1-\beta^+)(\lambda+\alpha)))}{D - \varepsilon_1(1-(1-\beta^-)(\lambda-\alpha))}$$

As we have $\varepsilon_1 = \beta^- T \frac{\lambda-\alpha}{1-(\lambda-\alpha)}$ and $\varepsilon_2 = \beta^+ T$ for some fixed T, we get:

$$\lim_{D\to\infty} \delta(D) \to 1$$

This concludes the proof. □

Theorem 6.1 guarantees that the running times of BSP processes, optimally scheduled based on the execution times of the previous superstep, differ at most by a factor q^2 within a load sequence. This fact will be utilized by the load balancing strategies.

Evaluating the collected data. When sectioning a given CPU usage sequence into load sequences, our goal is to obtain load sequences with a q-boundedness as small as possible and a duration as long as possible, while the rate of unusable time intervals should be as small as possible. Obviously, these three optimization targets depend on each other. We have processed the data collected from our PCs described in the beginning of this section (over 6.8 million samples) with a Perl program which yields an approximation for this non-trivial optimization problem.

The results. The average idle time over a week ranged from approx. 35% up to 95%, so there is obviously a huge amount of unused computation power. Time intervals of less than half an hour and such where the CPU is nearly fully utilized by the user or its usage fluctuates too heavily, are no candidates for a load sequence. The rate of wasted idle time in such intervals is less than 3%.

Choosing suitable values for the parameters of the load sequences, it was possible to section the given CPU usage sequences into load sequences such that the predominant part of the load sequences was 1.6-bounded.

On most PCs, the average duration of a load sequence was 4 hours or even much longer. Assuming the execution of a process, started at an arbitrary point during a load sequence, takes 30 minutes, the probability that it completes within the current load sequence is thus at least 87.5%. A detailed analysis of the results in each of the four networks can be found in [6].

Generating load profiles. In order to compare the load balancing strategies under the same circumstances, i. e., especially with exactly the same external work load, and to make experimental evaluations repeatable, we have extracted totaling eight typical load profiles from two of the networks, each using these time spans: Tuesday forenoon (9:00 a.m. to 1:00 p.m.), Tuesday afternoon (2:00 p.m. to 6:00 p.m.), Tuesday night (2:00 a.m. to 6:00 a.m.), and Sunday afternoon (2:00 p.m. to 6:00 p.m.). Besides, we have generated four artificial load profiles according to our model, using typical values for the parameters. A detailed discussion of the load profiles can be found in [6].

7. Experimental Evaluation. In the following we present a comparision of our Java-based library against a C-based implementation and the evaluation of our load balancing algorithms.

7.1. Performance Evaluation. In order to determine the performance drawback of PUBWCL in comparison to a BSP implementation in C, we have conducted benchmark tests with both PUBWCL and PUB under the same circumstances: We used a cluster of 48 dual Intel Pentium III Xeon 850 MHz machines, that were exclusively reserved for our experiments to avoid influences by external work load. The computers were interconnected by a switched Fast Ethernet. The used benchmark program was a sequence of 10 equal supersteps. Per superstep, each BSP process did a number of integer operations and sent a number of messages. We performed tests using every possible combination of these parameters:

- 8, 16, 24, 32, 48 BSP processes
- 10, 20, 30 messages per BSP process and superstep
- 10 kB, 50 kB, 100 kB message size
- 0, 10^8, $2 \cdot 10^8$, $3 \cdot 10^8$, ..., 10^9 integer operations per BSP process and superstep

Selected results of the benchmark tests are shown in Fig. 7.1. As you can see, both BSP libraries scale well, and there is a performance drawback of a factor 3.3. Note that the running time of this benchmark program is dominated by the sequential work. Communication tests with no sequential work showed a performance impact of a factor up to 8.7.

When porting existing BSP programs to PUBWCL, you cannot directly compare the running times due to the overhead of the Java memory management. For example, we ported our C-based solver for the 3-Satisfiability-Problem (3-SAT), which is a simple parallelized version of the sequential algorithm in [9], to PUBWCL. As in the case of the benchmark program, the algorithm is dominated by the sequential work. But in contrast to the benchmark program, it continuously creates, clones, and disposes complex Java objects. This is much slower than allocating, copying, and freeing structures in the corresponding C-program and has led to a performance drawback of a factor 5.4.

7.2. Evaluation of the Load Balancing Algorithms. In order to analyze our load balancing strategies, we have conducted experiments on 15 PCs running Windows XP Professional, among them 7 PCs with 933 MHz and 8 ones with 1.7 GHz. The PUBWCL server and the client used to control the experiments ran on a seperate PC.

We have simulated the external work load according to the load profiles mentioned in Section 6 and run

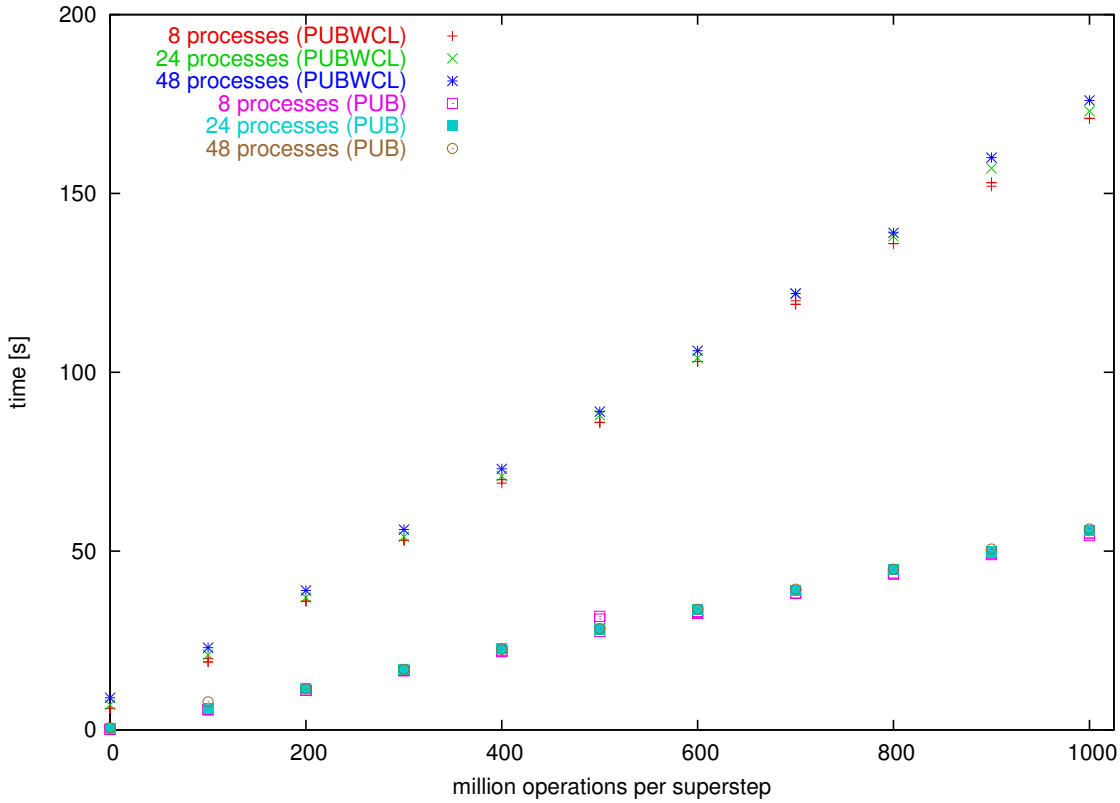


FIG. 7.1. *Results of the benchmark tests.*

the clients with the `/belownormal` priority switch, i. e., they could only consume the computation power left over by the load simulator.

The experiments were performed using a BSP benchmark program consisting of 80 equally weighted processes and 8 identical supersteps. Per superstep, each BSP process did $2 \cdot 10^9$ integer operations and sent (and received) 64 messages of 4 kB size each.

Results using PwoR. First we have to choose a suitable value for parameter r: At the beginning of a superstep the BSP processes are (re-) distributed over the clients such that they should complete execution at the same time. Supposing that, on any of the involved PCs, the current (q-bounded) load sequence does not end before completion of the superstep, none of the BSP processes should take longer than q times the average execution time. That means, choosing $r = q = 1.6$ guarantees that BSP processes are not migrated if the available computation power only varies within the scope of a q-bounded load sequence.

In comparison to experiments with no load balancing algorithm (i. e. initial distribution according to the ACP values and no redistribution of the processes during runtime), we could save 21% of the execution time averaged and even up to 36% in particular cases.

Comparing the execution times of the particular supersteps, we noticed that the execution time significantly decreases in the second superstep. The reason therefore is that the execution times of the previous superstep provide much more accurate values for load balancing than the estimated ACP values.

Results using PwR. Our experiments with the PwR algorithm resulted in noticealby longer execution times than those with the PwoR algorithm. We could obtain the best results with the parameters set to $r = 2$ and $s = \frac{1}{8}$; other choices led to even worse results.

On the one hand, this result is surprising as one would expect that PwR performs better than PwoR because it restarts BSP processes after some threshold instead of waiting for them for an arbitrarily long time. But on the other hand, restarting a BSP process is of no advantage if it would have completed on the original client within less time than its execution time on the new client. Our results show that the external work load apparently is not 'sufficiently malicious' for PwR to take advantage of its restart feature.

Results using SwoJ. Like with the PwR algorithm, $r = 2$ and $s = \frac{1}{8}$ is a good choice for the parameters because: In Section 6 we showed that a load sequence does not end inside a superstep with a probability of at least 87.5%. Thus the probability that a new load sequence with *more* available computation power starts inside a superstep is at most $\frac{1}{16}$. As we will use the normalized BSP process execution time on the x-th fastest client (where x is a fraction s of the number of affected clients) as a reference value for the abortion criterion, we ensure that no new load sequence has begun on this client with high probability by setting $s = \frac{1}{8}$ (instead of $s = \frac{1}{16}$). Provided that no new load sequence begins during the superstep, the factor between the fastest and the slowest normalized BSP process execution time on the particular clients is at most q^2. For a q-boundedness of $q = 1.6$ this yields $q^2 = 2.56$. We have actually chosen $r = 2$ because of our defensive choice of s.

Using these parameters, we could save 14% of the execution time averaged and even up to 25% in particular cases in comparison to the experiments with the PwoR algorithm; the savings in comparison to the experiments with no load balancing algorithm were even 32% averaged and up to 45% in isolated cases.

Results using SwJ. For the choice of the parameters, the same aspects as in the SwoJ case apply. In comparison to our experiments with the SwoJ algorithm we could save another 10% of the execution time averaged and even up to 27% in isolated cases; the savings in comparison to the experiments with no load balancing algorithm were even 39% averaged and up to 50% in particular cases.

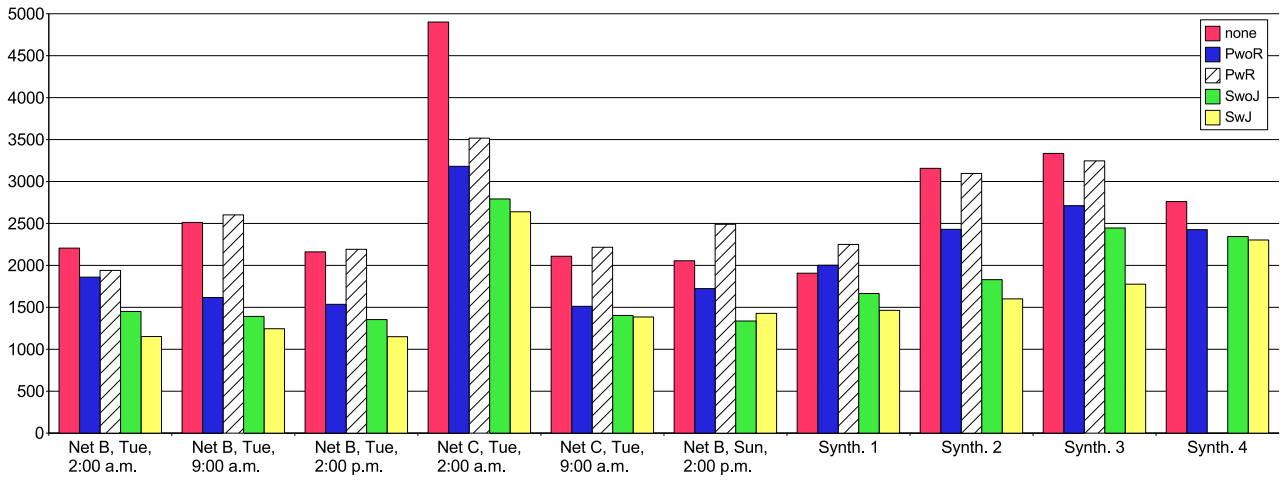


FIG. 7.2. *Running times depending on the load balancing algorithm.*

8. Conclusion. We have developed a web computing library that allows to execute BSP programs in a peer-to-peer network, utilizing only the computation power left over on the participating PCs. It features migration and restoration of the BSP processes in order to rebalance the load and increase fault tolerance because the available computation power fluctuates and computing nodes may join or leave the peer-to-peer system at any time.

We have implemented and analyzed different load balancing strategies for PUBWCL. The load balancing strategy SwJ performs better than SwoJ which, in turn, performs better than PwoR (cf. Fig. 7.2). In comparison to using no load balancing, we can save up to 50% of the execution duration using SwJ.

Due to security and portability reasons one has to use a virtual machine like Java's one, so a performance drawback cannot be avoided. The slowdown depends on the type of the BSP algorithm.

In order to further improve PUBWCL, work is in progress to realize PUBWCL as a pure peer-to-peer system in order to dispose of the bottleneck at the server, and to replace Java RMI in PUBWCL with a more efficient, customized protocol.

Additionally, we are working on an extension of PUBWCL which allows redundant execution of BSP processes, i. e., processes are started redundantly, but only the results of the fastest one are committed whereas the remaining processes are aborted when the first one completes. This will allow us to improve the load balancing strategies by starting additional instances of slow BSP processes on faster clients instead of just restarting them. Since, using the SwJ algorithm, typically only a very small fraction of the BSP processes was

restarted, this would mean only a low overhead but would significantly reduce the probability that they would have to be restarted another time.

Acknowledgement. Partially supported by DFG-SFB 376 "Massively Parallel Computation" and EU IST-2004-15964 (AEOLUS).

REFERENCES

[1] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004.
[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, *The Paderborn University BSP (PUB) library*, Parallel Computing, 29 (2003), pp. 187–207.
[3] *distributed.net*. `http://www.distributed.net/`
[4] S. Fünfrocken, *Transparent migration of Java-based mobile agents*, in Mobile Agents, 1998, pp. 26–37.
[5] J. Gehweiler, *Entwurf und Implementierung einer Laufzeitumgebung für parallele Algorithmen in Java*, Studienarbeit, Universität Paderborn, 2003.
[6] J. Gehweiler, *Implementierung und Analyse von Lastbalancierungsverfahren in einer Web-Computing-Umgebung*, Diplomarbeit, Universität Paderborn, 2005.
[7] *Great internet mersenne prime search (GIMPS)*. `http://www.mersenne.org/`
[8] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, 24 (1998), pp. 1947–1980.
[9] J. Hromkovic and W. M. Oliva, *Algorithmics for Hard Problems*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
[10] *JavaGgo RMI*. `http://www.joachim-gehweiler.de/en/software/javago.php`
[11] S. Leonardi, A. Marchetti-Spaccamela, and F. Meyer auf der Heide, *Scheduling against an adversarial network*, in SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, ACM Press, 2004, pp. 151–159.
[12] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau, *Delta execution: A preemptive Java thread migration mechanism*, Cluster Computing, 3 (2000), pp. 83–94.
[13] *The Paderborn University BSP library*. `http://wwwcs.uni-paderborn.de/~pub/`
[14] *The Paderborn University BSP-based Web Computing Library*. `http://wwwcs.uni-paderborn.de/~pubwcl/`
[15] T. Sakamoto, T. Sekiguchi, and A. Yonezawa, *Bytecode transformation for portable thread migration in Java*, in ASA/MA, 2000, pp. 16–28.
[16] L. F. G. Sarmenta, *An adaptive, fault-tolerant implementation of BSP for Java-based volunteer computing systems*, in Lecture Notes in Computer Science, vol. 1586, 1999, pp. 763–780.
[17] T. Sekiguchi, H. Masuhara, and A. Yonezawa, *A simple extension of Java language for controllable transparent migration and its portable implementation*, in Coordination Models and Languages, 1999, pp. 211–226.
[18] *Search for extraterrestrial intelligence (SETI@home)*. `http://setiathome.berkeley.edu/`
[19] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, *Portable support for transparent thread migration in Java*, in ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, London, UK, 2000, Springer-Verlag, pp. 29–43.
[20] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

Edited by: Przemysław Stpiczyński.
Received: March 31, 2006.
Accepted: May 28, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 15–23. http://www.scpe.org
ISSN 1895-1767

HEURISTIC LOAD BALANCING FOR CFD CODES EXECUTED IN HETEROGENEOUS COMPUTING ENVIRONMENTS

DANA PETCU*, DANIEL VIZMAN†, AND MARCIN PAPRZYCKI‡

Abstract. A graph partitioning-based heuristic load-balancing algorithm known as the Largest Task First with Minimum Finish Time and Available Communication Costs is modified to take into account the dynamic nature and heterogeneity of current large-scale distributed computing environments, like Grids. The modified algorithm is applied to facilitate load balancing of a known CFD code used to model crystal growth.

Key words. load balancing, grid and cluster computing, computational intensive problems

1. Introduction. One of the important new challenges in computational sciences is related to the rapid increase in size and computational power of heterogeneous computing platforms, like Grids. To be able to fully realize potential performance of parallel applications running on those platforms will require extra research effort. Opposite to a standard parallel computing environment (understood as a single parallel computer), Grid environment is highly unpredictable: available resources have different capacities, they can be added and removed practically at any time (and without warning), and availability of computational resources fluctuates over time (as individual node utilization changes). Therefore any application running in the Grid must react properly to those fluctuations, i. e. by utilizing dynamic load balancing.

One of research fields that is a possible candidate to attain good performance when using dynamic load balancing is computational fluid dynamics (CFD). Here, codes are computationally demanding, both in terms of memory usage and also in the number of arithmetic calculations and thus are large enough to be naturally partitioned into a number of sub-tasks. Furthermore, most natural methods of improving accuracy of a solution to the CFD problem are: (1) refining a mesh or (2) shortening the time step. Either of these approaches results in further substantial increase of both computational cost and total memory usage. Therefore, a natural tendency can be observed, to use whatever computational resources are available to the user.

Parallel CFD codes have been typically developed assuming their execution on a standard parallel computer, i. e. a homogeneous set of processors connected via a fast network. Recent ascent of Web and Grid-based technologies requires re-evaluation of these assumptions. Computational Grids are combining very large numbers of heterogeneous processors; through substantially slower (and heterogeneous in bandwidth) network connections. Furthermore, the migration process of CFD codes designed for parallel computing architectures towards Grids must take into account not only the heterogeneity of the new environment but also constant dynamic evolution of the pool of available computational resources. At the same time we have to acknowledge that, from the pragmatic point of view, assumption that it may be possible to fully rewrite the existing codes from the scratch is usually not a viable option, because of the cost involved in such an endeavor. Therefore a different approach has to be proposed to successfully port existing CFD codes to the grid.

In this context let us observe that a large body of research has been already devoted to dynamic load balancing in heterogeneous environments, and more recently in Grid environments. In this paper we will argue that this approach may provide us with an efficient way of solving CFD problems in the Grid. In the next section we present a short overview of related dynamic load balancing strategies. Section 3 describes the particular CFD code of our interest and its recent parallel implementation. Then, in Section 4, we present the load balancing algorithm that was modified to deal with specific conditions imposed by Grid environment, while Sections 5 discusses results of some tests.

2. Short overview of dynamic load balancing strategies. Different algorithms for load balancing have been proposed over the last twenty years. In this context, several studies were devoted to the classification of load balancing schemes, for instance [20]. More recently, load balancing strategies used in heterogeneous computing environments have been applied when designing and implementing applications in Grid environments. For example, in [5] a dynamic load-balancing scheme is used for a geophysical application running on a Grid

*Institute e-Austria Timişoara, and Computer Science Department, Western University of Timişoara, B-dul Vasile Parvan 4, 300223 Timişoara, Romania (petcu@info.uvt.ro).
†Physics Department, Western University of Timişoara, Romania
‡Computer Science Institute, SWPS, Warsaw, Poland

platform. Other approaches to dynamic load balancing were reported for a protein molecules docking application [3] or applied within a hydro-dynamics model computations [21].

Overall, load balancing algorithms can be classified into two categories: static or dynamic. In static algorithms, decisions related to load balancing are made at a compile time, when resource requirements are estimated and work appropriately divided. Dynamic load balancing algorithms allocate and reallocate resources at runtime based on current resource availability and information about tasks to be executed. Obviously, this approach is very likely to be more adequate to a Grid environment than a static one. Specifically, when considering a heterogeneous and dynamically changing computing environment, the load balancing algorithm should take into account at least the following parameters: memory requirements, computation costs in cycles, currently available memory, currently available idle cycles and current communication costs. In this context, a simple tool implementing a dynamic loop scheduling strategy to address load imbalance which may be induced by the heterogeneity of processors was recently reported in [4].

Looking from a slightly different perspective, approaches to load-balancing in distributed systems can be also classified into the following three categories: (1) graph-theoretic, (2) mathematical programming based, and (3) heuristic.

Graph-theoretic algorithms consider graphs representing the inter-task dependencies and apply graph partitioning methodologies to obtain approximately equal partitions of the graph such that the inter-node communication is minimized. A CFD simulation using such an approach is described in [8].

The mathematical programming method, views the load-balancing as an optimization problem and solves it using techniques originating from that domain.

Finally, heuristic methods provide fast solutions to the load balancing problem (even though usually sub-optimal ones) when the time to obtain the exact optimal solution is prohibitive. For example, in [2] a genetic algorithm is used as an iterative method to obtain near optimal solutions to a combinatorial optimization problem that is applied to job scheduling on the Grid.

One of the interesting implementation of heuristic approaches to lad balancing (in particular in the case of CFD problems) is the EVAH package [6]. It was developed to predict performance scalability of Grid applications executing on large numbers of processors. It consists of a set of allocation heuristics that consider the specific constraints inherent in multi-block CFD problems. In our work we are interested in a particular algorithm available within the EVAH package. The Largest-Task-First with Minimum-Finish-Time and Available-Communication-Costs (LTF_MFT_ACC) method combines graph partitioning approach to load balancing with a heuristic scheme. Efficiency tests performed with the LFT_MFT_ACC on an Origin2000 system and reported in [6] showed that it can provide an acceptable load balancing faster than other solutions. However, the main drawback of the original algorithm available within the EVAH package is that it assumes homogeneity of available resources.

It is also possible to approach the load balancing problem from the load management perspective. In this case possible approaches can be divided into (1) system level, and (2) user-level. A system-level centralized management strategy, which works over all running applications, uses schedulers to manage loads in the system. It is typically based on rules associated with job types or load classes.

An example of the user-level individual management of loads in a parallel computing environment is the Dynamic Load Balancing (DLB [12]) tool that lets the system balance loads without going through centralized load management and, furthermore, provides application level load balancing for individual parallel jobs. System load measurement of the DLB is modified using average load history provided by computing systems rather than by tracking processing of tasks. To illustrate the performance of the DLB tool, a CFD test case was used as an example (see [12] for more details).

In this paper we propose a modification of the LTF_MFT_ACC algorithm that can be applied in the case of a heterogeneous computing environment. Inspired by the DLB tool, our algorithm takes into account (1) the history of computation times on different nodes, (2) the communication requirements, and (3) the current network speeds. To study the robustness of the proposed improvements, the modified algorithm was used to port an existing parallel CFD code into a heterogeneous computing environment.

Finally, let us note that, according [10], load balancing algorithms can be defined by their implementation of the following policies: (1) information policy that specifies what workload information is to be collected, when it is to be collected and from where; (2) triggering policy that determines the appropriate period to start a load balancing operation; (3) resource type policy that classifies a resource as server or receiver of tasks according to its availability status; (4) location policy that uses the results of the resource type policy to find a suitable

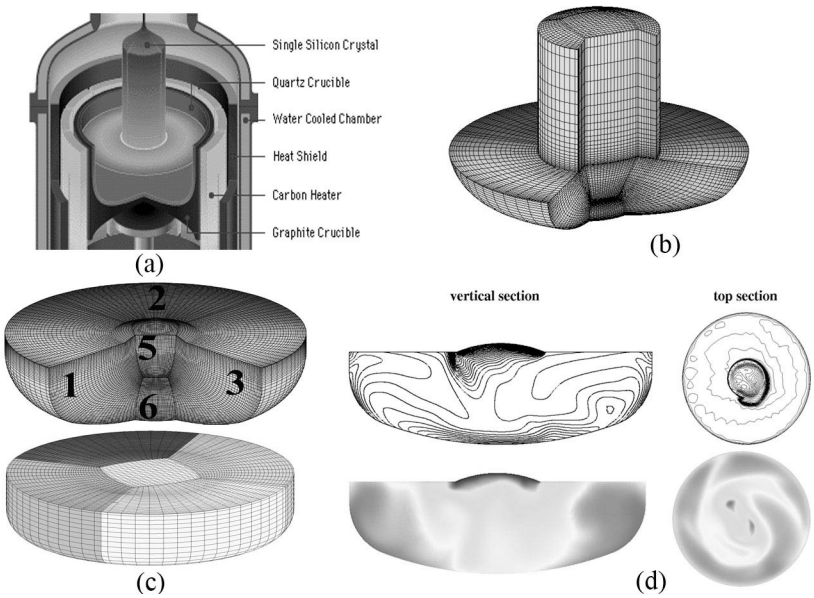


FIG. 3.1. *Crystal growth: (a) device; (b) control volumes; (c) blocks of control volumes; (d) code outputs—isoterms and animation frames*

partner for a server or receiver; (5) selection policy that defines tasks that should be migrated from overloaded resources to least utilized resources.

In what follows we refer only to the information and selection policies of the proposed load balancing algorithm, the other policies being subject of future work. In our work we utilize results presented in a useful study on information policy recently reported in [1].

3. Crystal growth simulation. Modeling materials processing systems often involves situations where a number of distinct materials and phases with significantly different thermo-physical and transport properties have to be taken into account. Understanding of complex transport phenomena in these systems is of vital importance for the design and fabrication of various desired products as well as optimization and control of the manufacturing process. Numerical simulations prove to be an effective tool for understanding of transport mechanisms. Three-dimensional simulations are necessary to yield a reliable description of the flow behavior. Typical computational methods applied to these problems include finite difference, finite volume, finite element, and spectral methods.

In particular, let us consider the Czochralski process [17] of bulk crystal growth that features a rod holding an oriented crystal seed which is lowered through the top surface of the molten liquid contained in a crucible. With thermal control to maintain the upper surface of the fluid at the melt temperature, growth begins on the seed and when the crystal reaches a specified diameter, the rod is slowly withdrawn to continue growth (Figure 3.1.a). The flow in the melt, from which the crystal is pulled, is transient and, depending on the size of the crucible, mostly turbulent.

The silicon melt flow into a rotating crucible is governed by three-dimensional partial differential equations describing mass, momentum, and heat transport. Solution methods that employ finite volume (see e.g. [7]) require generation of the solution grid that conforms to the geometry of the flow region (a grid of small volume elements for which the average values of flow quantities are stored). An important issue for the quality of the numerical simulations is the choice of the grid. Here, both the numerical resolution and the internal structure of the grid are very important. The second item that has to be considered, is the refining of the grid in the proximity of walls of the melt container, which is necessary to properly resolve boundary layers of the flow.

The finite volume code STHAMAS 3D (developed partially by the second author at the Institute of Materials Science in Erlangen) allows three-dimensional time-dependent simulations on a block-structured numerical grid. A matched multiblock method is used in simulations; the grid lines match each other at the block conjunction. A multiblock structured grid system [19] uses advanced linear solvers, for the inner iteration, and

a multigrid technique for the outer iterations. Furthermore, the computational domain is divided into blocks consisting of control volumes (from hundreds to millions; see Figure 3.1.b), while the SIP (Stone's strongly implicit procedure [18]) is used to solve the system of linear equation resulting from the discretization of PDEs for three-dimensional problems (it is applicable to seven-diagonal coefficient matrices that are obtained when central-difference approximation is used to discretize the problem). SIMPLE algorithm (Semi-Implicit Method for Pressure-Linked Equations, [13]) is used for the pressure correction and the implicit Euler method is applied for time integration. Note that the SIP and the SIMPLE were studied and compared with other solvers and shown to be very robust (in [9] the efficiency of multigrid SIMPLE based algorithms was examined when computing incompressible flows).

A simple example of the graphical output of the code is presented in Figure 3.1.d.

Time-dependence and three-dimensionality coupled with extensive parameter variations require a very large amount of computational resources and result in very long solution times. The most time-consuming part of the sequential code STHAMAS 3D is the numerical solution obtained, using the SIP applied to different blocks. In order to decrease the response time of STHAMAS 3D, a parallel version was recently developed by the first two authors and presented in [14]. It is based on a parallel version of the SIP solver, where simultaneous computations are performed on different blocks mapped to different processors (different colors in Figure 3.1.c). After each inner iteration, information exchanges are performed at the level of block surfaces (using calls to the MPI library). Thus far, the new parallel STHAMAS 3D was tested only utilizing homogeneous computing environments, in particular, on a cluster of workstations and a parallel computer.

For completeness it should be noted that a different parallel version of a crystal growth simulation has been reported in [11]. It utilizes a parallel version of the SSOR preconditioner and the BiCGSTAB iterative solver.

4. The modified algorithm for load balancing. The STHAMAS 3D code can be effectively utilized on a parallel computer. However, it has to adapted to be equally robust in a distributed and non-homogeneous computing environment. To achieve this goal we have decided to look into dynamic load balancing procedures offered with the, above described, EVAH package [6].

In the Largest Task First with Minimum Finish Time and Available Communication Costs algorithm (LTF_MFT_ACC, Figure 4.1) the size of a task is defined as the computation time required to perform that task (t_i in Figure 4.1). According to the Largest Task First (LTF) policy, to achieve load balancing the algorithm first sorts tasks in descending order by size (execution time). Then it systematically allocates tasks to processors respecting the rule of Minimum Finish Time (LTF_MFT). Note that the overhead involved in task distribution, caused by data exchanges between tasks, is also taken into account (in Figure 4.1 the communication cost between the sender processor o and the receiver processor q is denoted by c_{oq}). The LTF_MFT_ACC utilizes communication costs which are estimated from the inter-task data volume exchange and the inter-processor communication rate.

Inputs:
Task times: $t_i,\ i = 1, \ldots, N$,
Communication time between tasks: $c_{oq},\ o \neq q, o, q = 1, \ldots, P$
Output:
Distribution of the N blocks on P processors: $p(i) \in \{1, \ldots, P\},\ i = 1, \ldots, N$

To do:
Sort $t_i, i = 1, N$ in descending order
Set costs: $C_p = 0, p = 1, \ldots, P$
For each $t_i, i = 1, \ldots, N$ do
  Find the processor q with minimum load: $?q,\ C_q = \min_{p=1,\ldots,P} C_p$
  Associate block i with processor q: $p(i) \leftarrow q$
  Modify the costs: $C_q \leftarrow C_q + t_i$
  For each processor $o \neq q$ having assigned a task j communicating with task i:
    $C_o \leftarrow C_o + c_{oq}$
    $C_q \leftarrow C_q + c_{qo}$

FIG. 4.1. *Initial LTF_MFT_ACC algorithm*

Input:
Old distribution of the N tasks on P processors: $p(i) \in \{1, \ldots, P\},\ i = 1, \ldots, N$
Output:
New distribution of the N tasks on P processors: $p'(i) \in \{1, \ldots, P\},\ i = 1, \ldots, N$

Preparation phase:
Record the task times: $T_i,\ i = 1, \ldots, N$
Record the quantity of data to be send/receive between tasks: $V_{j,k},\ j, k = 1, \ldots, N$
Estimate communication time between each pair of processors depending on the quantity of transmitted data: $Send(p, q, dim), Recv(p, q, dim), p, q = 1, \ldots, P, p \neq q$
Record the time spent to perform a standard test: $c_p,\ p = 1, \ldots, P$
Compute the relative speeds of computers: $w_p \leftarrow c_p / \min_{p=1,\ldots,P} c_p,\ p = 1, \ldots, P$
Normalize computation time for each task: $t_i \leftarrow T_i / w_{p(i)}, i = 1, \ldots, N$

To do:
Sort $t_i, i = 1, N$ in descending order
Set costs: $C_p = 0, p = 1, \ldots, P$
For each $t_i, i = 1, \ldots, N$:
Find the processor q with minimum load: $?q,\ C_q = \min_{p=1,\ldots,P} C_p$
Associate task i with processor q: $p'(i) \leftarrow q$
Modify the costs: $C_q \leftarrow C_q + w_q t_i$
For each processor $o \neq q$ having assigned a task j sending a message to task i:
$C_o \leftarrow C_o + Send(o, q, V(j, i))$
$C_q \leftarrow C_q + Recv(q, o, V(j, i))$
For each processor $o \neq q$ having assigned a task j receiving a message from task i:
$C_o \leftarrow C_o + Recv(o, q, V(i, j))$
$C_q \leftarrow C_q + Send(q, o, V(i, j))$

FIG. 4.2. *Modified LTF_MFT_ACC algorithm*

Note that the original LTF_MFT_ACC algorithm, described in [6], was proposed for a computational environment with homogeneous resources. Therefore, the computational time t_i, for the task i, is "independent" from the processor where the task i is being executed. To take into account the variation of the computational power of grid resources, we have modified the LTF_MFT_ACC as depicted in Figure 4.2 (an extended discussion of the algorithm can be found in [16]).

For the dynamic load balancing to be effective, performance data concerning environment within which the algorithm is executed has to be collected. Therefore, when the load balancing procedure is activated, several counters are started to measure:

— the computer power, represented as the time of performing a cycle involving floating point operations; this information is further used to rank the resources;
— the computation time spent working on each task; this information and the relative computer power are used to assess the time that is going to be spent working on that task by other processors;
— the required volume of data to be exchanged between communicating tasks represented as the number of elementary data elements;
— samples of communication times between each pair of processors collected for several volumes data— additional required communication times are estimated by linear interpolation.

Note that the modified algorithm must work as well as the initial one in the case of the homogeneous environment (when $w_p = 1,\ p = 1, \ldots, P$).

The first main difference between the original and the modified algorithm can be seen in the input values: while the initial algorithm supposes the knowledge of the computation and communication times (can be provided as an input or theoretically estimated), the modified algorithm utilizes the original iterative process until a certain moment, when the load balancing procedure is called, so that a task distribution is already available and the computation and communication times are have been estimated utilizing this distribution. Registration of those times is similar to the one found in the DLB [12] that uses as inputs for its load balancing strategy

the timing of computation for parallel tasks and the dimension of data to be communicated—those times are referring to an average load history provided by computing systems that are parts of the grid environment that is used to solve the problem.

The second main difference is concerns the splitting of the communication time into two different components: the message sending time is registered as the cost at the message source and the message receiving time is registered as the cost at the message destination. This splitting is motivated by the fact that the message exchange is asynchronous and there is a possible time overlap between computations and communications, so the non-blocking sending event can take place some time before the receiving event.

5. Influence of processor heterogeneity—case study. In what follows we want to show that the first difference results in a better load balancing solution than the one provided by the initial algorithm. Let us start from a simple "theoretical" illustration and assume that (as in the above described case of crystal growth modeling) a grid of small volume elements (for which the approximated values of flow are stored) that conforms to the geometry of the flow region is generated. More precisely, a multiblock structured grid of volume elements with 6 blocks with different number of volumes is used (see the example from Figure 3.1.c). Here, inner iterations in each block are associated with a task. Figure 5.1 shows a graph representation of the relationships between those tasks.

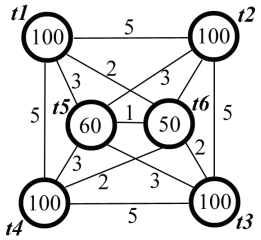


FIG. 5.1. *Relationships between tasks*

The smallest time registered in the system is represented by a time unit in the graph (respectively the communication time between blocks 5 and 6). Computation times registered for the inner iterations performed within each block before an outer iteration (not including any message exchanges) are shown inside graph vertices. The communication overhead generated by sending messages from one block to another is shown along the graph edges. In order to simplify the example we consider that in the solution process there are no delays in message exchanges (the receiver does not wait for incoming messages). To further simplify the example, we consider that the exchange values represent the total combined value of the send and the receive times that take place before an outer iteration.

Let us now consider the case of using $P = 3$ processors. When the computing environment is homogeneous, the LTF_MFT_ACC algorithm distributes tasks as follows:

processor 1: tasks 1, 4
processor 2: tasks 2, 5
processor 3: tasks 3, 6

the longest execution time being the one of processor 1. This time is dominated by the task times: over 200 time units.

When the computing environment is heterogeneous, the first four tasks will be performed resulting in different computing times. We consider the case of $w_1 = 1.5,\ w_2 = 1.8,\ w_3 = 1$. Thus, processor 3 is the fastest one and processor 2 is the slowest one. The LTF_MFT_ACC algorithm can provide an initial distribution of the tasks according to a theoretical estimation of the computation time (proportional to the number of volumes inside each block) and the communication time (proportional to the volume of data to be exchanged at block interfaces). In the considered case task distribution suggested by the LTF_MFT_ACC algorithm will be the same as the above described one, the longest time being still the one of the processor 1: over 300 time units (150 time units for each task).

The modified LTF_MFT_ACC algorithm can be applied only when an initial distribution is available and the computation and communication times are registered for this distribution. We can consider that the initial

distribution is the one provided by the initial algorithm. Dividing the registered computation times to calculate relative speedups we will obtain approximately the same values as these registered in the graph described in Figure 5.1. Then the new distribution suggested by the modified algorithm will be:

processor 1: tasks 1, 5
processor 2: tasks 2, 6
processor 3: tasks 3, 4

the longest time being the one of processor 2: over 270 time units (180 time units for the task 2 and 90 time units for task 6), which is an improvement compared to what the initial algorithm resulted in.

In the described case the communication time is not influencing the decision how to distribute tasks. If the ratio of computation time to communication time will be smaller than the above considered one it is possible to have another task distribution. In such a case it is important to also take into account the different speeds of the message exchanges between processors that can be captured by the second proposed modification of the initial algorithm. This case will not be discussed further.

6. Experiment. The proposed modified algorithm was applied to dynamically balance load of the computational effort of the parallel version of the STHAMAS 3D in a heterogeneous computing environment. Detailed test results have been presented in [15]; we mention here the most significant of them.

First, let us provide some background information related to other algorithms mentioned in our paper. The initial LTF_MFT_ACC algorithm was applied in [6] to a selected CFD problem—a Navier-Stokes simulation of vortex dynamics in the complex wake of a region around hovering rotors. The overset grid system consisted of 857 blocks and approximately 69 million grid points. The experiments running on the 512-processor SGI Origin2000 distributed-shared-memory system showed that the *EVAH* algorithm performs better than other heuristic algorithms.

Similarly, the CFD test case from the [12] was a solution of a heat transfer problem in a three-dimensional grid consisting of 27-block partitions with 40x40x40 grid points in each block (1.7 millions of grid points). For a range of relative speeds of computers from $1 \div 1.55$ a 21% improvement in the elapsed time was registered.

In our work we have considered a three-dimensional grid applied to the crystal growth simulation escribed above. We used 38-block-partitions with a variable number of grid points: the largest one had 25x25x40 points, while the smallest one had 6x25x13 points (total of 0.3 millions of grid points in the simulation).

We have utilized two computing environments: a homogeneous and a heterogeneous one. The homogeneous computing environment was a Linux cluster of 8 dual PIV Xeon 2GHz Processors with 2Gb RAM and a Myrinet2000 interconnection (`http://www.oscer.edu`). The heterogeneous computing environment was a Linux network of 17 PCs with variable computational power, from Intel Celeron running at 0.6GHz, with 128Mb RAM to Intel PIV running at 3GHz and with 1Gb RAM (the interval of the relative speeds of computers is $1 \div 2.9$); these machines were connected through an Ethernet 10 Mbs interconnection (`http://www.risc.uni-linz.ac.at`).

Initially block-partitions were distributed uniformly between the processors (e.g. in the case of two processors, first 19 blocks were send to the first processor, and the remaining 19 blocks were send to the second processor).

Due to the different number of grid points in individual blocks, the initial algorithm running in the homogeneous environment recommended a new distribution of nodes. The proposed re-distribution of nodes resulted in a reduction of 17% of the time spent by the code in the inner iteration (Figure 6.1.a).

In the case of the heterogeneous environment, the LTF_MFT_ACC algorithm performed better: we observed a time reduction ranging from 14% to 20% of the time spent by the CFD code in the inner iteration. A further reduction of the time was obtained when applying the modified LTF_MFT_ACC algorithm—varying from 20% to 31% (Figure 6.1.b). Comparing time results with these obtained for the initial distribution, a total time reduction attained in our experiments varied from 33% to 45%. We have also found out that the total overhead introduced by the proposed load balancing algorithm was about 6% of the running time.

7. Concluding remarks. The proposed load balancing technique shows to be useful in the considered case, a version of a CFD code running within heterogeneous computing environment. Tests must be further performed to compare several dynamic load balancing techniques with the proposed one, not only in what concerns the influence of the computer power variations as considered in this paper, but also of the network speed variation and memory availability. A larger testbed will be used in the near future to perform such tests.

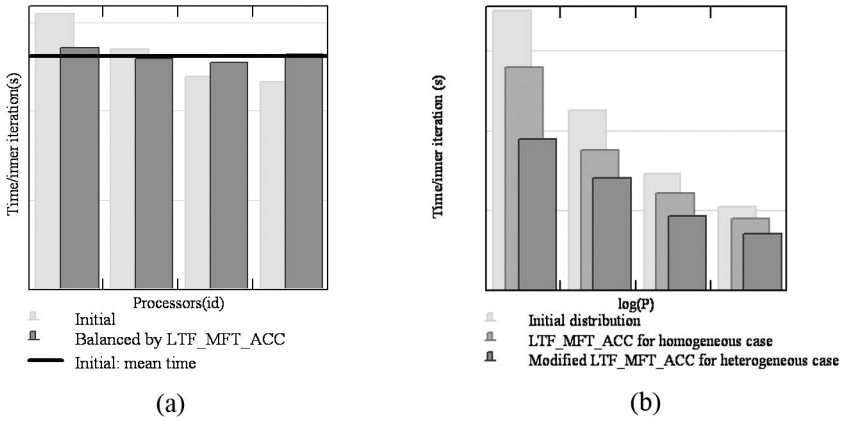


FIG. 6.1. *Load balancing results : (a) in the case of 4 processors of the homogeneous environment, the* LTF_MFT_ACC *algorithm reduces the differences between the computation time spent by each processor in the inner iteration; (b) in the case of 2, 4, 8 and 16 processors of the heterogeneous environment, the* LTF_MFT_ACC *reduces the time per inner iteration, but a further significant reduction is possible using the modified* LTF_MFT_ACC *algorithm*

Acknowledgments. This work was partially supported by the NanoSim project in the frame of Romanian CEEX Programme.

We would like to thank the Oklahoma University Supercomputing Center and the Research Institute for Symbolic Computing for providing us with access to their computing resources during this study.

REFERENCES

[1] M. BELTRAN, J.L. BOSQUE, AND A. GUZMAN, *Information policies for load balancing on heterogeneous systems*, in Procs. IEEE/ACM Internat. Symp. Cluster Computing and the Grid (2005) pp. 970-976.
[2] J. CAO, D. P. SPOONER, S. A. JARVIS, S. SAINI, AND G. R. NUDD, *Agent-based grid load balancing using performance-driven task scheduling*, in Procs.IPDPS03, IEEE Computer Press (2003).
[3] S. CHEN1, W. ZHANG, F. MA, J. SHEN, AND M. LI, *A novel agent-based load balancing algorithm for Grid computing*, in Procs. GCC 2004, H. Jin, Y. Pan, N. Xiao, and J. Sun (Eds.), LNCS 3252 (2004), pp. 156-163.
[4] R.L. CARINO AND I. BANICESCU, *A load balancing tool for distributed parallel loops*, Cluster Computing, 8 (2005), pp. 313-321.
[5] R. DAVID, S. GENAUD, A. GIERSCH, B. SCHWARZ, AND E. VIOLARD, *Source code transformations strategies to load-balance grid applications*, in Procs. GRID 2002, M. Parashar (ed.), LNCS 2536, Springer (2002), pp. 82–87.
[6] M. J. DJOMEHRI, R. BISWAS, N. LOPEZ-BENITEZ, *Load balancing strategies for multi-block overset grid applications*, NAS-03-007, available at `http://www.nas.nasa.gov/News/Techreports/2003/PDF/nas-03-007.pdf`
[7] J. H. FERZIGER, M. PERIĆ, *Computational Methods for Fluid Dynamics*, Springer Verlag, Berlin (1996).
[8] H. GAO, A. SCHMIDT, A. GUPTA, P. LUKSCH, *Load balancing for spatial-grid based parallel numerical simulations on clusters of SMPs*, in Procs. Euro PDP03, IEEE Computer Press (2003), pp. 75–82.
[9] O. ILIEV, M. SCÄFER, *A numerical study of the efficiency of* SIMPLE*-type algorithms in computing incompressible flows on streched grids*, in Procs. LSSC99, M. Griebel, S. Margenov, P. Yalamov (eds.), Notes on Numerical Fluid Mechanics 73, Vieweg (2000), pp. 207–214.
[10] H.D. KARATZA, Job scheduling in heterogeneous distributed systems, Journal of Systems and Software, 56 (2001), pp. 203-212.
[11] D. LUKANIN, V. KALAEV, A. ZHMAKIN, *Parallel simulation of Czochralski crystal growth*, in Procs. PPAM 2003, R. Wyrzykowski et al, LNCS 3019 (2004), pp. 469–474.
[12] R. U. PAYLI, E. YILMAZ, A. ECER, H.U. AKAY, AND S. CHIEN, DLB *A dynamic load balancing tool for grid computing*, in Procs. Parallel CFD04, G. Winter, A. Ecer, F. N. Satofuka, P. Fox (eds.), Elsevier (2005), pp. 391–399
[13] M. PERIĆ, *A finite volume method for the prediction of three-dimensional fluid flow in complex ducts*, Ph.D. Thesis, University of London (1985).
[14] D. PETCU, D. VIZMAN, J. FRIEDRICH, AND M. POPESCU, *Crystal growth simulation on clusters*, in Procs. of HPC2003, I. Banicescu (ed.), Simulation Councils Inc. San Diego (2003), pp. 41–46
[15] D. PETCU, D. VIZMAN, AND M. PAPRZYCKI, *Porting CFD codes towards grids—a case study*, in Procs. of PPAM 2005, R. Wyrzykowski et al (eds.), LNCS 3911 (2006), in print.
[16] D. PETCU, *Adapting a partitioning-based heuristic load-balancing algorithm to heterogeneous computing environments*, in Procs. SYNASC 2005, IEEE Computer Society Press, Los Alamitos, pp. 170–173.
[17] P. A. SACKINGER, R. A. BROWN, J. J. BROWN, *A finite element method for analysis of fluid flow, heat transfer and free interfaces in Czochralski crystal growth*, Internat. J. Numer. Methods in Fluids 9 (1989), pp. 453–492.
[18] H. L. STONE, *Iterative solution of implicit approximations of multidimensional partial differential equations*, SIAM J. Num. Anal. 5 (1968), pp. 530–558.

[19] D. Sun, *A multiblock and multigrid technique for simulations of material processing*, Ph.D. Thesis, State University of New York (2001).
[20] C. Z. Xu, F. C. M. Lau, *Load Balancing in Parallel Computers: Theory and Practice*, Kluwer Academic Publishers, Boston (1996).
[21] S. Yu, J. Casey, and W. Zhou, *A load balancing algorithm for Web based server Grids*, in Procs. GCC 2003, M. Li et al. (Eds.), LNCS 3033 (2004), pp. 121-128.

Edited by: Przemysław Stipiczyński.
Received: May 5, 2006.
Accepted: May 28, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 25–32. http://www.scpe.org
ISSN 1895-1767

BENCHMARKING OF A JOINT IRISGRID/EGEE TESTBED WITH A BIOINFORMATICS APPLICATION

J. HERRERA* , E. HUEDO* , R. S. MONTERO* , AND I. M. LLORENTE*

Abstract. Loosely-coupled Grid environments, based only on Globus services, allow a straightforward resource sharing, as the resources are accessed by using *de facto* standard protocols and interfaces, while providing non trivial levels of quality of service. This paper describes the execution of a Bioinformatics application over a highly distributed and heterogeneous testbed. This testbed is composed of resources devoted to EGEE and IRISGrid projects and has been integrated by taking advantage of the modular, decentralized and "end-to-end" architecture of the GridWay framework. Results obtained from the experiments have been analyzed using a performance model for high throughput computing applications.

1. Introduction. Different Grid infrastructures are being deployed within growing national and transnational research projects. The final goal of these projects is to provide the end user with much higher performance than that achievable on any single site. However, from our point of view, it is arguable that some of these projects embrace the Grid philosophy, and to what extent. This philosophy, proposed by Foster [7], defines *a grid* as a system (i) not subject to a centralized control and (ii) based on standard, open and general-purpose interfaces and protocols, (iii) while providing some level of quality of service (QoS), in terms of security, throughput, response time or the coordinated use of different resource types. In current projects, there is a tendency to ignore the first two requirements in order to get higher levels of QoS. However, these requirements are even more important because they are the key to the success of *the Grid*.

The Grid philosophy leads to computational environments, which we call *loosely-coupled* grids, mainly characterized by [3]: autonomy (of the multiple administration domains), heterogeneity, scalability and dynamism. In a loosely-coupled grid, the different layers of the infrastructure should be separated from each other, being only communicated with a limited and well defined set of interfaces and protocols. This layers are [3]: Grid fabric, core Grid middleware, user-level Grid middleware, and Grid applications.

The coexistence of several projects, each with its own middleware developments, adaptations or extensions, arise the idea of using them simultaneously (from an user's viewpoint) or contribute the same resources to more than one project (from an administrator's viewpoint). One approach could be the development of gateways between different middleware implementations [1]. Other approach, more in line with the Grid philosophy, is the development of client tools that can adapt to different middleware implementations. We hope this could lead to a shift of functionality from resources to brokers or clients, allowing the resources to be accessed in a standard way and easing the task of sharing resources between organizations and projects. We should consider that the Grid not only involves the technical challenge of constructing and deploying this vast infrastructure, it also brings up other issues related to security and resource sharing policies [13] as well as other socio-political difficulties [15].

Practically, the majority of the Grid infrastructures are being built on protocols and services provided by the Globus Toolkit[1], becoming a *de facto* standard in Grid computing. Globus architecture follows an hourglass approach, which is indeed an "end-to-end" principle [2]. Therefore, instead of succumbing to the temptation of tailoring the core Grid middleware to our needs (since in such case the resulting infrastructure would be application specific), or homogenizing the underlying resources (since in such case the resulting infrastructure would be a highly distributed cluster), we propose to strictly follow the "end-to-end" principle. Clients should have access to a wide range of resources provided through a limited and standardized set of protocols and interfaces. In the Grid, these are provided by the core Grid middleware, Globus, just as, in the Internet, they are provided through the TCP/IP protocols. Moreover, the "end-to-end" principle reduces the firewall configuration to a minimum, which is also welcome by the security administrators.

One of the most ambitious projects to date is EGEE[2] (Enabling Grids for E-sciencE), which is creating a production-level Grid infrastructure providing a level of performance and reliability never achieved before. EGEE currently uses the LCG[3] (LHC Computing Grid) middleware, which is based on Globus. Other much

*Departamento de Arquitectura de Computadores y Automática. Facultad de Informática, Universidad Complutense de Madrid. 28040 Madrid, Spain. {jherrera, ehuedo}@fdi.ucm.es, {rubensm, llorente}@dacya.ucm.es

[1] http://www.globus.org

[2] http://www.eu-egee.org

[3] http://lcg.web.cern.ch

TABLE 2.1
IRISGrid and EGEE resources contributed to the experiment.

Testbed	Site	Resource	Processor	Speed	Nodes	RM
IRISGrid	RedIRIS	heraclito	Intel Celeron	700MHz	1	Fork
		platon	2×Intel PIII	1.4GHz	1	Fork
		descartes	Intel P4	2.6GHz	1	Fork
		socrates	Intel P4	2.6GHz	1	Fork
	DACYA-UCM	aquila	Intel PIII	700MHz	1	Fork
		cepheus	Intel PIII	600MHz	1	Fork
		cygnus	Intel P4	2.5GHz	1	Fork
		hydrus	Intel P4	2.5GHz	1	Fork
	LCASAT-CAB	babieca	Alpha EV67	450MHz	30	PBS
	CESGA	bw	Intel P4	3.2GHz	80	PBS
	IMEDEA	llucalcari	AMD Athlon	800MHz	4	PBS
	DIF-UM	augusto	4×Intel Xeon†	2.4GHz	1	Fork
		caligula	4×Intel Xeon†	2.4GHz	1	Fork
		claudio	4×Intel Xeon†	2.4GHz	1	Fork
	BIFI-UNIZAR	lxsrv1	Intel P4	3.2GHz	50	SGE
EGEE	LCASAT-CAB	ce00	Intel P4	2.8GHz	8	PBS
	CNB	mallarme	2×Intel Xeon	2.0GHz	8	PBS
	CIEMAT	lcg02	Intel P4	2.8GHz	6	PBS
	FT-UAM	grid003	Intel P4	2.6GHz	49	PBS
	IFCA	gtbcg12	2×Intel PIII	1.3GHz	34	PBS
	IFIC	lcg2ce	AMD Athlon	1.2GHz	117	PBS
	PIC	lcgce02	Intel P4	2.8GHz	69	PBS

†These resources actually present two physical CPUs but they appear as four logical CPUs due to hyper-threading

more modest project is IRISGrid[4] (the Spanish Grid Initiative), whose main objective is the creation of a stable national Grid infrastructure. The first version of the IRISGrid testbed is based only on Globus services, and it has been widely used through the Grid*W*ay framework[5].

For the purposes of this paper we have used a Globus-based testbed to run a Bioinformatics application through the Grid*W*ay framework. This testbed was built up from resources inside IRISGrid and EGEE projects. The aim of this paper is to demonstrate the application of an "end-to-end" principle in a Grid infrastructure, and the feasibility of building loosely-coupled Grid environments based only on Globus services, while obtaining non trivial levels of quality of service through an appropriate user-level Grid middleware.

The structure of the paper follows the layered structure of Grid systems, from bottom-up. The Grid fabric is described Section 2. Section 3 describes the core Grid middleware. Section 4 introduces the functionality and characteristics of the Grid*W*ay framework, used as user-level Grid middleware. Section 5 describes the target application. Section 6 presents the experimental results, which are analyzed through a benchmarking model in Section 7. Finally, Section 8 ends up with some conclusions.

2. Grid Fabric: IRISGrid and EGEE resources. This work has been possible thanks to the collaboration of those research centers and universities that temporarily shared some of their resources in order to set up a geographically distributed testbed. The testbed results in a very heterogeneous infrastructure, since it presents several middlewares, architectures, processor speeds, resource managers (RM), network links, etc. A brief description of the participating resources is shown in Table 2.1.

Some centers are inside IRISGrid, which is composed of around 40 research groups from different spanish institutions. Seven sites participated in the experiment by donating a total number of 195 CPUs. Other centers

[4]http://www.irisgrid.es
[5]http://www.gridway.org

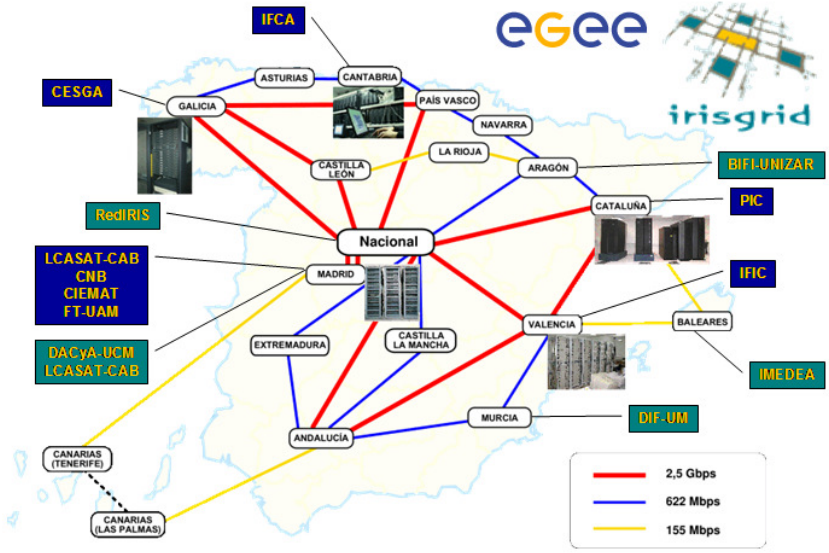


FIG. 2.1. *Geographical distribution and interconnection network of sites.*

participate in the EGEE project, which is composed of more than 100 contracting and non-contracting partners. Seven spanish centers participated by donating a total number of 333 CPUs.

Together, the testbed is composed of 13 sites (note that LCASAT-CAB is both in IRISGrid and EGEE) and 528 CPUs. In the experiments below, we limited to four the number of jobs simultaneously submitted to the same resource, with the aim of not saturating the whole testbed, so only 64 CPUs were used at the same time. All sites are connected by RedIRIS, the Spanish Research and Academic Network. The geographical location and interconnection links of the different sites are shown in Figure 2.1.

3. Core Grid Middleware: Globus. Globus services allow secure and transparent access to resources across multiple administrative domains, and serve as building blocks to implement the stages of Grid scheduling [14]. Table 3.1 summarizes the core Grid middleware components existing in both IRISGrid and EGEE resources used in the experiments. In the case of EGEE, we only used Globus basic services, ignoring any higher-level services, like the resource broker or the replica location service.

TABLE 3.1
Core Grid middleware.

Component	IRISGrid	EGEE
Security Infrastructure	IRISGrid CA and manually generated `grid-mapfile`	DATAGRID-ES CA and automatically generated `grid-mapfile`
Resource Management	GRAM with shared home directory in clusters	GRAM without shared home directory in clusters
Information Services	IRISGrid GIIS and local GRIS, using the MDS schema	CERN BDII and local GRIS, using the GLUE schema
Data Management	GASS and GridFTP	GASS and GridFTP

We had to introduce some changes in the security infrastructure in order to perform the experiments. For authentication, we used a user certificate issued by DATAGRID-ES CA, so we had to give trust to this CA on IRISGrid resources. Regarding authorization, we had to add an entry for the user in the `grid-mapfile` in both IRISGrid and EGEE resources.

4. User-Level Grid Middleware: GridWay. User-level middleware is required in the client side to make it easier and more efficient the execution of applications. Such client middleware should provide the end user with portable programming paradigms and common interfaces.

In a Globus-based environment, the user is responsible for manually performing all the submission steps [14] in order to achieve any functionality. To overcome this limitation, GridWay [11] was designed with a *submit & forget* philosophy in mind. The core of the GridWay framework is a personal *submission agent* that performs all scheduling stages and watches over the correct and efficient execution of jobs on Globus-based Grids. The GridWay framework provides adaptive scheduling and execution, as well as fault tolerance capabilities to handle the dynamic Grid characteristics.

A key aspect in order to follow the "end-to-end" principle is how job execution is performed. In EGEE, file transfers are initiated by a job wrapper running in the compute nodes, therefore they act as client machines, so needing network connectivity and client tools to interact with the middleware. In GridWay, however, job execution is performed in three steps by the following modules:

1. *prolog*: It prepares the remote system by creating a experiment directory and transferring the input files from the client.
2. *wrapper*: It executes the actual job and obtains its exit status code.
3. *epilog*: It finalizes the remote system by transferring the output files back to the client and cleaning up the experiment directory.

This way, GridWay doesn't rely on the underlying middleware to perform preparation and finalization tasks. Moreover, since both *prolog* and *epilog* are submitted to the front-end node of a cluster and *wrapper* is submitted to a compute node, GridWay doesn't require any middleware installation nor network connectivity in the compute nodes.

Other projects [5, 6, 8, 16] have also addressed resource selection, data management, and execution adaptation. We do not claim innovation in these areas, but remark the advantages of our modular, decentralized and "end-to-end" architecture for job adaptation to a dynamic environment.

In this case, we have taken full advantage of the modular architecture of GridWay, as we didn't have to directly modify the source code of the *submission agent*. We extended the *resource selector* in order to understand the GLUE schema used in EGEE. The *wrapper* module also had to be modified in order to perform an explicit file staging between the front-end and the compute nodes in EGEE clusters.

5. Grid Application: Computational Proteomics. One of the main challenges in Computational Biology concerns the analysis of the huge amount of protein sequences provided by genomic projects at an ever increasing pace. In the following experiments, we will consider a Bioinformatics application aimed at predicting the structure and thermodynamic properties of a target protein from its amino acid sequence [10].

The algorithm, tested in the 5th round of Critical Assessment of techniques for protein Structure Prediction (CASP5)[6], aligns with gaps the target sequence with all the 6150 non-redundant structures in the Protein Data Bank (PDB)[7], and evaluates the match between sequence and structure based on a simplified free energy function plus a gap penalty item. The lowest scoring alignment found is regarded as the prediction if it satisfies some quality requirements. In such cases, the algorithm can be used to estimate thermodynamic parameters of the target sequence, such as the folding free energy and the normalized energy gap.

To speed up the analysis and reduce the data needed, the PDB files are pre-processed to extract the contact matrices, which provide a reduced representation of protein structures. The algorithm is then applied twice, the first time as a fast search, in order to select the 200 best candidate structures, and the second time with parameters allowing a more accurate search of the optimal alignment.

We have applied the algorithm to the prediction of thermodynamic properties of families of orthologous proteins, i. e. proteins performing the same function in different organisms. The biological results of the comparative study of several families of orthologous proteins are presented elsewhere [4].

[6] `http://PredictionCenter.llnl.gov/casp5/`
[7] `http://www.pdb.org`

FIG. 6.1. *Testbed dynamic throughput during the five experiments and theoretical throughput of the most powerful site.*

The experiment files consists of: the executable (0.5MB) provided for all the resource architectures in the testbed, the PDB files shared and compressed (12.2MB) to reduce the transfer time, the parameter files (1KB), and the file with the sequence to be analyzed (1KB). The final name of the executable and the file with the sequence to be analyzed is obtained at runtime for each task and each host, respectively.

6. Experiences and Results. The experiments presented here consist in the analysis of a family of 80 orthologous proteins of the *Triose Phosphate Isomerase* enzyme (an enzyme is a special case of protein). Five experiments were conducted in different days during a week. The average turnaround time for the five experiments was 43.37 minutes.

Figure 6.1 shows the dynamic throughput achieved during the five experiments alongside the theoretical throughput of the most powerful site, where the problem could be solved in the lowest time, in this case DIF-UM (taking into account that the number of active jobs per resource was limited to four). The throughput achieved on each experiment varies considerably, due to the dynamic availability and load of the testbed. For example, resource ce00 at site LCASAT-CAB was not available during the execution of the first experiment. Moreover, fluctuations in the load of network links and computational resources induced by non-Grid users affected to a lesser extent in the second experiment, as it was performed at midnight.

7. Grid Benchmarking Model. In this section we apply a benchmarking methodology to analyze the performance of computational Grid infrastructures in the execution of a High Throughput Computing (HTC) applications [12]. In order to assess this model we will use the execution of the Bioinformatics application explained in previous sections. The benchmarking process used here provides a way to investigate performance properties of Grid environments, to predict the performance of this category of applications, and compare different platforms by inserting performance models in the benchmarking process.

7.1. Workload Characterization. In order to obtain the workload characterization of a grid, we have considered the formerly mentioned Bioinformatics application, that comprises the execution of a set of independent tasks, each of which performs the same calculation over a subset of parameter values. In the execution of this kind of applications, a grid can be considered, from the computational point of view, as an array of *heterogeneous* processors. Therefore, the number of tasks completed as function of time is given by the following equation:

$$n(t) = \sum_{i \in G} N_i \left\lfloor \frac{t}{T_i} \right\rfloor \tag{7.1}$$

where N_i is the number of processors in the Grid (G) that can compute a task in T_i seconds.

The best characterization of the Grid can be obtained if we take the line that represents the average behavior of the system. The next formula represents this line using the r_∞ and $n_{1/2}$ parameters defined by Hockney and Jesshope [9]:

$$n(t) = r_\infty t - n_{1/2} \text{ with } m = r_\infty \text{ and } b = -n_{1/2} \tag{7.2}$$

These parameters are called:

- Asymptotic performance (r_∞): the maximum rate of performance in task executed per second. In the case of an homogeneous array of N processors with an execution time per task T, we have $r_\infty = N/T$.
- Half-performance length ($n_{1/2}$): the number of task required to obtain the half of asymptotic performance. This parameter is also a measure of the amount of parallelism in the system as seen by the application. In the homogeneous case, for an embarrassingly distributed application we obtain $n_{1/2} = N/2$.

The following equation defines the performance of the system (tasks completed per time) on actual applications with a finite number of tasks based on the linear relation of Eq. 7.2:

$$r(n) = n(t)/t = \frac{r_\infty}{1 + n_{1/2}/n} \tag{7.3}$$

The half-performance length ($n_{1/2}$) provides a quantitative measure of the homogeneity in a grid. We can define the degree of homogeneity (v) as

$$v = \frac{2n_{1/2}}{N}. \tag{7.4}$$

This parameter varies from $v = 1$ in the homogeneous case, to $v \approx 0$ when the actual number of processors in the Grid is much greater than the apparent number of processors (highly heterogeneous).

7.2. Benchmarking Methodologies. Previously, we have defined the r_∞ and $n_{1/2}$ parameters to obtain a Grid model. These parameters can be determined in two ways:

- *Intrusive* benchmarking. The system parameters are calculated by linear fitting to the experimental results obtained in the execution of large-scale HTC applications. In order to empirically determine r_∞ and $n_{1/2}$ the benchmarking process should be intrusive to exercise all the resources in the testbed ($n >> N$).
- *Non-intrusive* benchmarking. In general, it may not be feasible to run such an intrusive high throughput benchmark for large Grid environments. In this situation, the r_∞ and $n_{1/2}$ parameters can be computed using 7.3 and raw performance data (average wall time per task, T_i) of each resource. Let us consider $T_i = T_i^{xfr} + T_i^{exe} + T_i^{sch}$, where T_i^{xfr} and T_i^{exe} are the average file transfer and execution times in host i; and T_i^{sch} is the scheduling time which represents the resource selection overhead. We can rewrite Eq. 7.1 as:

$$n(t) = \sum_{i \in G} N_i \left\lfloor \frac{t}{T_i^{xfr} + T_i^{exe} + T_i^{sch}} \right\rfloor. \tag{7.5}$$

This equation can be used to obtain the *non-intrusive* r_∞ and $n_{1/2}$ parameters by fitting the best straight line. We can also estimate the influence of the resource selection overhead by comparing the non-intrusive r_∞ and $n_{1/2}$, with those obtained by setting T^{sch} to 0 in Eq. 7.5.

7.3. Experiments and Results. We begin the analysis presenting the intrusive and non-intrusive measurement made on testbed of the parameters r_∞ and $n_{1/2}$. Figure 7.1 shows the experimental performance obtained in the first two executions of the mentioned Bioinfomatics application, along with that predicted by Eq. 7.3. The r_∞ and $n_{1/2}$ have been calculated by linear fitting to: (i) experimental results obtained in the execution of the application; (ii) Eq. 7.5 using the average file transfer and execution times of each host and $T^{sch} = 60s$; and also (iii) Eq. 7.5 with $T^{sch} = 0s$.

This figure shows the effects of the resource selection on the optimum performance. The resource selection process reduces the asymptotic performance of the Grid, because of a delay between different tasks submitted

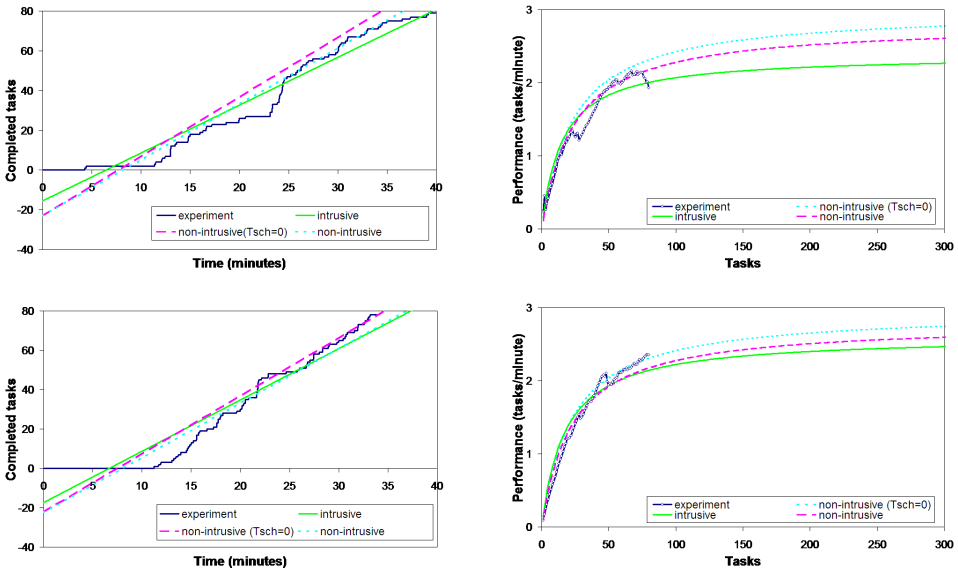


FIG. 7.1. *Measurements of r_∞ and $n_{1/2}$ on the testbed based on experimental data, and raw resource performance (left-hand charts). Experimental performance of the Bioinformatic application, along with that predicted by Eq. 7.3 (right-hand charts).*

to the same host. This delay is mainly due to the Globus MDS update frequency and the Grid*W*ay resource broker. However, it does not affect to the $n_{1/2}$ parameter, since the brokering overhead increases the execution time (T_i) by the same amount in all the resources.

Table 7.1 shows the values of the r_∞ and $n_{1/2}$ coefficients for each execution of the experiments. For the shake of the completeness, we also include the turnaround time (T_{Grid}). Based on the *intrusive* results, the testbed is characterized by an average asymptotic performance of 2.31 tasks per minute. Furthermore, in order to achieve the half of this asymptotic performance it is necessary to execute, on average, 15.34 tasks. And so, the apparent number of resources to the application is approximately 30.68, with an execution time of 13.26 minutes per task.

In Table 7.1, we can also see the degree of homogeneity (v) for the intrusive and non-intrusive case. Thus, the average of the intrusive v is 0.61 (low homogeneity degree), and the average of non-intrusive v is 0.9, closer to the homogeneous case ($v = 1$). This difference is mainly due to the experiments not saturating the testbed, as can be seen in the right-hand charts of Figure 7.1. Moreover, this difference is also consequence of calculating non-intrusive v by applying the model under ideal conditions, therefore yielding in an homogenization of the results.

TABLE 7.1
Turnaround time T_{Grid}, $r_\infty(tasks/minute)$, $n_{1/2}$ (tasks), and degree of heterogeneity coefficient v for each experiment.

	Intrusive			Non-Intrusive			
Experiment	r_∞	$n_{1/2}$	v	r_∞	$n_{1/2}$	v	T_{Grid}
1	2.38	15.02	0.58	2.85	23.20	0.89	42.00
2	2.61	17.48	0.70	2.79	22.54	0.90	33.82
3	2.04	17.04	0.71	2.74	21.52	0.90	46.05
4	2.14	13.88	0.51	2.67	23.64	0.88	50.37
5	2.40	13.27	0.53	2.95	22.73	0.91	44.87

8. Conclusions. Loosely-coupled grids allow a straightforward resource sharing since resources are accessed and exploited through *de facto* standard protocols and interfaces, similar to the early stages of the Internet. This way, the loosely-coupled model allows an easier, scalable and compatible deployment.

We have shown that the "end-to-end" principle works at the client side (i. e. the user-level Grid middleware) of a Grid infrastructure. Our proposed user-level Grid middleware, GridWay, can work with Globus, as a standard core Grid middleware, over any Grid fabric in a *loosely-coupled* way. The smooth process of integration of two so different testbeds, although both are based on Globus, demonstrates that the GridWay approach (i. e. the Grid way), based on a modular, decentralized and "end-to-end" architecture, is appropriate for the Grid. Moreover, the Grid performance model for High Throughput Computing Applications validates the experimental results obtained.

Acknowledgments. This research was supported by Consejería de Educación of Comunidad de Madrid, Fondo Europeo de Desarrollo Regional (FEDER) and Fondo Social Europeo (FSE) through BIOGRIDNET Research Program S-0505/TIC/000101, by BSCH/UCM through research grant PR27/05-14035-BSCH and by Ministerio de Ciencia y Tecnología, through the research grant TIC 2003-01321. The authors participate in the EGEE project.

We would like to thank all the institutions involved in the IRISGrid initiative and the EGEE project, in particular those who collaborated in the experiments. They are Red Académica y de Investigación Nacional (RedIRIS), Departamento de Arquitectura de Computadores y Automática (DACyA) at Universidad Complutense de Madrid (UCM), Laboratorio de Computación Avanzada, Simulación y Aplicaciones Telemáticas (LCASAT) at Centro de Astrobiología (CAB), Centro de Supercomputación de Galicia (CESGA), Instituto Mediterráneo de Estudios Avanzados (IMEDEA), Facultad de Informática (DIF) at Universidad de Murcia (UM), Instituto de Biocomputación y Física de Sistemas Complejos (BIFI) at Universidad de Zaragoza (UNIZAR), Instituto de Física de Cantabria (IFCA), Instituto de Física Corpuscular (IFIC), Centro Nacional de Biotecnología (CNB), Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT), Departamento de Física Teórica (FT) at Universidad Autónoma de Madrid (UAM) and Port d'Informació Científica (PIC).

We would like to also thank Ugo Bastolla, staff scientist in the Bioinformatics Unit at Centro de Astrobiología (CAB) and developer of the Bioinformatics application used in the experiments.

REFERENCES

[1] R. J. Allan, J. Gordon, A. McNab, S. Newhouse, and M. Parker, *Building Overlapping Grids*, tech. rep., University of Cambridge, Oct. 2003.

[2] E. B. Carpenter, *RFC 1958: Architectural Principles of the Internet*, June 1996.

[3] M. Baker, R. Buyya, and D. Laforenza, *Grids and Grid Technologies for Wide-Area Distributed Computing*, Software – Practice and Experience, 32 (2002), pp. 1437–1466.

[4] U. Bastolla, A. Moya, E. Viguera, and R. van Ham, *Genomic Determinants of Protein Folding Thermodynamics in Prokaryotic Organisms*, Journal of Molecular Biology, 343 (2004), pp. 1451–1466.

[5] F. Berman, R. Wolski, H. Casanova, et al., *Adaptive Computing on the Grid Using AppLeS*, IEEE Trans. Parallel and Distributed Systems, 14 (2003), pp. 369–382.

[6] R. Buyya, D.Abramson, and J. Giddy, *A Computational Economy for Grid Computing and Its Implementation in the Nimrod-G Resource Broker*, Future Generation Computer Systems, 18 (2002), pp. 1061–1074.

[7] I. Foster, *What Is the Grid? A Three Point Checklist*, GRIDtoday, 1 (2002). `http://www.gridtoday.com/02/0722/100136.html`

[8] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, *Condor/G: A Computation Management Agent for Multi-Institutional Grids*, Cluster Computing, 5 (2002), pp. 237–246.

[9] R. Hockney and C. Jesshope, *Parallel Computers 2:Architecture Programming and Algorithms*, Adam Hilgee Ltd., 1998.

[10] E. Huedo, U. Bastolla, R. S. Montero, and I. M. Llorente, *A Framework for Protein Structure Prediction on the Grid*, New Generation Computing, 23 (2005), pp. 277–290.

[11] E. Huedo, R. S. Montero, and I. M. Llorente, *A Framework for Adaptive Execution on Grids*, Intl. J. Software—Practice and Experience (SPE), 34 (2004), pp. 631–651.

[12] R. S. Montero, E. Huedo, and I. M. Llorente, *Benchmarking of High Ttroughput Computing Applications on Grids*, Parallel Computing, (2006). in press.

[13] O. San José, L. M. Suárez, E. Huedo, R. S. Montero, and I. M. Llorente, *Resource Performance Management on Computational Grids*, in Proc. 2nd Intl. Symp. Parallel and Distributed Computing (ISPDC 2003), IEEE CS, 2003, pp. 215–221.

[14] J. M. Schopf, *Ten Actions when Superscheduling*, Tech. Rep. GFD-I.4, Scheduling Working Group—The Global Grid Forum, 2001.

[15] J. M. Schopf and B. Nitzberg, *Grids: The Top Ten Questions*, Scientific Programming, special issue on Grid Computing, 10 (2002), pp. 103–111.

[16] S. Vadhiyar and J. Dongarra, *A Performance Oriented Migration Framework for the Grid*, in Proc. 3rd Intl. Symp. Cluster Computing and the Grid (CCGrid 2003), IEEE CS, 2003, pp. 130–137.

Edited by: Przemysław Stpiczyński.
Received: April 3, 2006.
Accepted: May 28, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 35–51. http://www.scpe.org
ISSN 1895-1767

MATHEMATICAL SERVICE DISCOVERY: ARCHITECTURE, IMPLEMENTATION AND PERFORMANCE

SIMONE A. LUDWIG*, OMER F. RANA†, WILLIAM NAYLOR‡, AND JULIAN PADGET§

Abstract. Service discovery and matchmaking in a distributed environment has been an active research issue since at least the mid 1990s. Previous work on matchmaking has typically presented the problem and service descriptions as free or structured (marked-up) text, so that keyword searches, tree-matching or simple constraint solving are sufficient to identify matches. In this paper, we discuss the problem of matchmaking for mathematical services, where the semantics play a critical role in determining the applicability or otherwise of a service. A matchmaking architecture supporting the use of match plug-ins is first described, followed by the types of plug-ins that can be supported. The matched services are ranked based on the score obtained from each plug-in, with the user being able to decide which plug-in is most significant in the context of their particular application. We consider the effect of pre- and post-conditions of mathematical service descriptions on matching, and how and why to reduce queries into DNF and CNF before matching. Application examples demonstrate in detail how the matching process works for all four algorithms. Additionally, an evaluation of the ontological mode is provided, regarding performance of loading ontologies, query response time and the overall scalability is conducted. The performance results are used to demonstrate scalability issues in supporting ontology-based discovery within a Web Services environment.

Key words. mathematical Web Services, matchmaking, match score, performance, scalability.

1. Introduction. The amount of machine-oriented data on the Web is increasing rapidly as semantic Web technologies achieve greater up-take. Humans typically use Google to search for suitable services, but they can filter out the irrelevant and spot the useful. Therefore, while UDDI (the Web Services registry) with keyword searching essentially offers something similar, it is a long way from being very helpful. Often, it is difficult to formulate a query in a precise way, therefore resulting in the results of search engines not being suitable for automated services—as many of the matches returned by the search may not be directly relevant to the query. A human user is therefore necessary to evaluate the outcome of a search, and make manual selection between the set of results that have been returned. Conversely, when using automated registry services, the options for specifying a query are limited, therefore resulting in a service that has limited usability.

To address this concern, much research has taken place on intelligent brokerage, such as Infosleuth [2], LARKS [3], and IBROW [4]. It is perhaps telling that much of the literature appears to focus on architectures for brokerage, which are as such domain-independent, rather than concrete or domain-specific techniques for identifying matches between a *task* or problem description and a *capability* or service description. Approaches to matching in the literature fall into two broad categories:

- *syntactic matching*, such as textual comparison or the presence of keywords in free text. In these approaches, the terms used in the query are matched with those in the service description. Either the existence (or not) of terms is used to undertake a comparison, or any attributes or data values associated with these terms are also compared.
- *semantic matching*, which typically seems to mean finding elements in structured (marked-up) data and perhaps additionally the satisfaction of constraints specifying ranges of values or relationships between one element and another.

For many problems this is both appropriate and adequate, indeed it is not clear what more one could do, but in the particular domain of mathematical services the actual mathematical semantics are critical to determining the suitability (or otherwise) of the capability for the task. The requirements are neatly captured in [5] by the following condition:

$$T_{in} \geq C_{in} \wedge T_{out} \leq C_{out} \wedge T_{pre} \Rightarrow C_{pre} \wedge C_{post} \Rightarrow T_{post} \tag{1.1}$$

where T refers to the task, C to the capability, *in* are inputs, *out* are outputs, *pre* are pre-conditions and *post* are post-conditions. What the condition expresses is that the signature constrains the task inputs to be at least as great as the capability inputs (i. e. enough information), that the inverse relationship holds for the

*School of Computer Science, Cardiff University, UK (Simone.Ludwig@cs.cardiff.ac.uk)
†School of Computer Science, Cardiff University, UK (O.F.Rana@cs.cardiff.ac.uk)
‡Department of Computer Science, University of Bath, UK (wn@bath.ac.uk)
§Department of Computer Science, University of Bath, UK (jap@bath.ac.uk)

outputs and there is a pairwise implication between the respective pre- and post-conditions. This however leaves unaddressed the matter of establishing the validity of that implication.

In the MONET [6, 8] and GENSS [9] projects the objective is mathematical problem solving through service discovery and composition by means of intelligent brokerage. *Mathematical* capability descriptions turn out to be both a blessing and a curse: precise service description are possible thanks to the use of the OpenMath [10] mathematical semantic mark-up, but service matching can rapidly turn into intractable (symbolic) mathematical calculations unless care is taken.

As Grid computing adopts the Service Oriented Architecture for service usage and deployment, a matchmaker can be seen as another infrastructure service—deployed as part of a discovery infrastructure. Many of the capabilities of a "matchmaker" are similar to those of a "broker", as adopted in many existing Grid computing applications. However, a key distinction we make is that a broker does not provide any support for scheduling or executing a service once it has been discovered. The focus here is on numerical services, as they are generally much better understood, and therefore may also serve as useful benchmarks for evaluating a system. Furthermore, multiple instances of mathematical services (implemented by a variety of different vendors) can be found, thereby providing the capability to choose between similar services made available over different deployment platforms.

This paper discusses our experience with developing such a domain-specific matchmaking approach for mathematical services. We suggest the use of a different matchmaking mechanisms to answering the implication question posed above. The set of ontologies provided by the MONET project have been used to specify a query. This query is subsequently used to undertake a search against the services that have been registered in a registry. Performance issues are also discussed. In the next section we discuss the description of mathematical services, in section 3 we describe the web services-based matchmaking architecture and detail the roles of the components within it, in section 4 we provide application examples outlining integer factorisation. Performance results are presented in section 5, and in section 6 we summarise and conclude this paper.

2. Description of Mathematical Services. The OpenMath [10] has been introduced as a means to describe properties of mathematical objects (for exchange between computer programs). OpenMath is a mark up language for representing the semantics (as opposed to the presentation) of mathematical objects in an unambiguous way. It may be expressed using an XML syntax. OpenMath expressions are composed of a small number of primitives. The definition of these may be found in [10], for instance: `OMA` (OpenMath Application), `OMI` (OpenMath Integer), `OMS` (OpenMath Symbol) and `OMV` (OpenMath Variable). Symbols are used to represent objects defined in the Content Dictionaries (to be discussed). Applications specify that the first child is a function or operator to be applied to the following children. As an example, the expression $x + 1$ might look like:[1]

```
<om:OMA>
  <om:OMS cd="arith1" name="plus"/>
  <om:OMV name="x"/>
  <om:OMI> 1 </om:OMI>
</om:OMA>
```

where the symbol `plus` is defined in the *Content Dictionary* (CD) `arith1`. Content Dictionaries are definition repositories in the form of files defining a collection of related symbols and their meanings, together with various *Commented Mathematical Properties* (for human consumption) and *Formal Mathematical Properties* (for machine consumption). The symbols may be uniquely referenced by the CD name and symbol name via the attributes `cd` and `name` respectively, as in the above example. Another way of thinking of a CD is as a small, specialised ontology.

MathML comes in two forms:

- for presentation (rendering in browsers) and
- for content (semantics),

and both are W3C recommendations. The specification ([11] section 5.1) and [12] identify ambiguities in presentation MathML. Content MathML is designed to handle the semantics of a limited subset of mathematics up to *K-12* level, mathematics beyond this may be encoded by using OpenMath and the *semantics* tag, alternatively *parallel markup* may be utilised [11]. There are various ways in which OpenMath can help in matchmaking:

[1] Throughout the paper, the prefix `om` is used to denote the namespace: `http://www.openmath.org/OpenMath`

- OpenMath can be used to encode the mathematical part of a problem to be solved in a query, for example a differential equation or an integral.
- OpenMath may be used to encode the input parameters to be sent to a service and the values returned by the service.
- The function of a service (together with the signature of the input parameters, and output objects) may be described in OpenMath, these may then be encoded using specialist tags to form a mathematical service description; described in Mathematical Service Description Language (MSDL) [13].

MSDL is an extension of WSDL that was developed as part of the MONET project [6], incorporating more information about a service, in particular pre- and post-conditions, taxonomic references etc.

3. Mathematical Matchmaking. Matchmaking allows potential producers of information to advertise their capability, and subsequently consumers of information to send messages describing their information needs. These descriptions, represented in rich, machine-interpretable description languages, are unified by the matchmaker to identify potential matches [7]. A Web Services-based matchmaking architecture is presented, describing the role played by different components within the architecture. Also described are the schemas and ontologies that may be used as part of the producer advertisement or consumer requirement.

3.1. Schemas and Ontologies. The XML document schemas we are using in GENSS are, at the moment, those developed for the MONET project. There are three main schemas:

- Mathematical Service Description Language (MSDL)
- Mathematical Problem Description Language (MPDL) and
- Mathematical Query Description Language (MQDL)

whose purposes are apparent in the second word in each case. The obvious question, even criticism, is why develop this range of languages when there is DAML-S [14] and OWL-S [15]? The answer is purely practical: at the time of the MONET project (April 2002 to March 2004) OWL and OWL-S were still subject to change and there were hardly any tools available, while DAML and DAML-S were clearly about to be made obsolete by OWL/OWL-S and the tool situation was hardly any better. Thus a pragmatic decision was made to take the principles needed to enable the MONET deliverables from DAML/OWL and embed them in some simple restricted languages over which the project had full control. Thus we see the adoption of pre- and post-condition driven descriptions of capabilities and tasks, following the ideas set out in DAML/OWL and being explored in various semantic brokerage projects such as InfoSleuth [2], RETSINA [3] and IBROW [16], while embedding WSDL in MSDL to provide the necessary information about how to invoke the service. It is our intention to explore how we can move from MSDL towards OWL/OWL-S during the lifetime of the current GENSS project, since this will greatly aid interoperability and enable the utilisation of the increasing range of tools (for OWL/OWL-S) that have become available.

History is also the explanation for the ontology language we use. OpenMath [10] provided an early use of XML for providing an application-specific description—before the availability of RDF. Nevertheless, OpenMath stands as probably the most developed ontology of mathematics, because in contrast to MATHML-C [11] it is extensible through the mechanism of content dictionaries which were developed to handle the absence of modularisation facilities or namespaces in the original XML. The OpenMath 2 standard [18] replaces DTDs with schemas, provides compatibility with RDF tools, utilises XML namespaces and generally aims to bring OpenMath in line with developments in ontology representation over the last five years, whilst keeping where feasible, backwards compatibility with OpenMath 1.1. Thus we make use of OpenMath as the primary representation language for mathematical content in our work.

3.2. Related Matchmaking Approaches. A variety of matchmaking systems have been reported in literature, we review some related systems below.

The SHADE (SHAred Dependency Engineering) matchmaker [19] operates over logic-based and structured text languages. The aim is to dynamically connect information sources. The matchmaking process is based on KQML (Knowledge Query and Manipulation Language) communication [20]. Content languages of SHADE are a subset of KIF (Knowledge Interchange Format) [21] as well as a structured logic representation called MAX (Meta-reasoning Architecture for "X"). Matchmaking is carried out solely by matching the content of advertisements and requests. There is no knowledge base and no inference performed. The SHADE matchmaker is implemented entirely as a forward-chaining rule-based program using MAX. This allows adding features such as new rules dynamically.

COINS (COmmon INterest Seeker) [19] is a matchmaker which operates over free text. The motivation for the COINS is the need for matchmaking over large volumes of unstructured text on the Web or other Wide Area Networks and the impracticality of using traditional matchmakers in such an application domain. Initially the free text matchmaker was implemented as the central part of the COINS system but it turned out that it was also useful as a general purpose facility. As in SHADE the access language is KQML. The System for the Mechanical Analysis and Retrieval of Text (SMART) [22] information retrieval system is used to process free text. The text is converted into a document vector using SMART's stemming and "noise" word removal. Then the document vectors are compared using an inverse document frequency algorithm.

LARKS (Language for Advertisement and Request for Knowledge Sharing) [3] was developed to enable interoperability between heterogeneous software agents and had a strong influence on the DAML-S specification. The system uses ontologies defined by a concept language ITL (Information Terminology Language). The technique used to calculate the similarity of ontological concepts involves the construction of a weighted associative network, where the weights indicate the belief in relationships. While it is argued that the weights can be set automatically by default, it is clear that the construction of realistically weighted relationships requires human involvement, which becomes a hard task when thousands of agents are available.

InfoSleuth [2] is a system for discovery and retrieval of information in open and dynamically changing environments. The brokering function provides reasoning over the advertised syntax and the semantics. InfoSleuth aims to support cooperation among several software agents for information discovery, where agents have roles as core, resource or ontology agents. A central service is the broker agent which is equipped with a matchmaker which matches agents that require services with agents that can provide those services. To apply this procedure an advertising agent has to register with the broker agent. The broker inserts the agent's description into its broker repository. The broker can then execute queries by requesting agents. These queries are formulated by agents who need other agents to fulfil their tasks.

The GRAPPA [23] (Generic Request Architecture) system allows multiple types of matchmaking mechanisms to be employed within a system. It is based on receiving arbitrary matchmaking offers and requests, where each offer and request consist of multiple criteria. Matching is achieved by applying distance functions which compute the similarities between the individual dimensions of an offer and a request. Using particular aggregate functions, the similarities are condensed to a single value and reported to the user.

The MathBroker (and MathBroker II) project(s) at RISC-Linz have some elements in common with those described here, including providing semantic descriptions of mathematical services. They too use MSDL, however it seems that most of the matchmaking is achieved through traversing taxonomies, while actual reasoning about the pre- and post-conditions is still an open problem.

Most of the projects above have focused on providing a generic matchmaker, capable of being adapted for a particular application. However, the motivation for many such projects has primarily been e-commerce (as a means to match buyers with sellers, for instance). Some projects are also focused on the use of a particular multi-agent interaction language (such as KQML), to enable communication between the matchmaker and other agents. Our approach, however, is centered on the implementation of a matchmaker that is specific to mathematical relations. Similar to GRAPPA, our matchmaker can support multiple comparison techniques.

3.3. Matchmaking Requirements. To achieve matchmaking:

- we want sufficient input information in the task to satisfy the capability, while the outputs of the matched service should contain at least as much information as the task is seeking, and
- the task pre-conditions should be more than satisfied by the capability pre-conditions, while the post-conditions of the capability should be more than satisfied by the post-conditions of the task.

These constraints reflect work in component-based software engineering and are, in fact, derived from [24]. They are also more restrictive than is necessary for our setting, by which we mean that some inputs required by a capability can readily be inferred from the task, such as the lower limit on a numerical integration or the dependent variable in a symbolic integration. Conversely, a numerical integration routine might only work from 0 to the upper limit, while the lower limit of the problem is non-zero. A capability that matches the task can be synthesised from the composition of two invocations of the capability with the fixed lower limit of 0. Clearly the nature of the second solution is quite different from the first, but both serve to illustrate the complexity of this domain. It is precisely this richness too that dictates the nature of the matchmaking architecture, because as these two simple examples show, very different reasoning capabilities are required to resolve the first and the second. Furthermore, we believe that given the na-

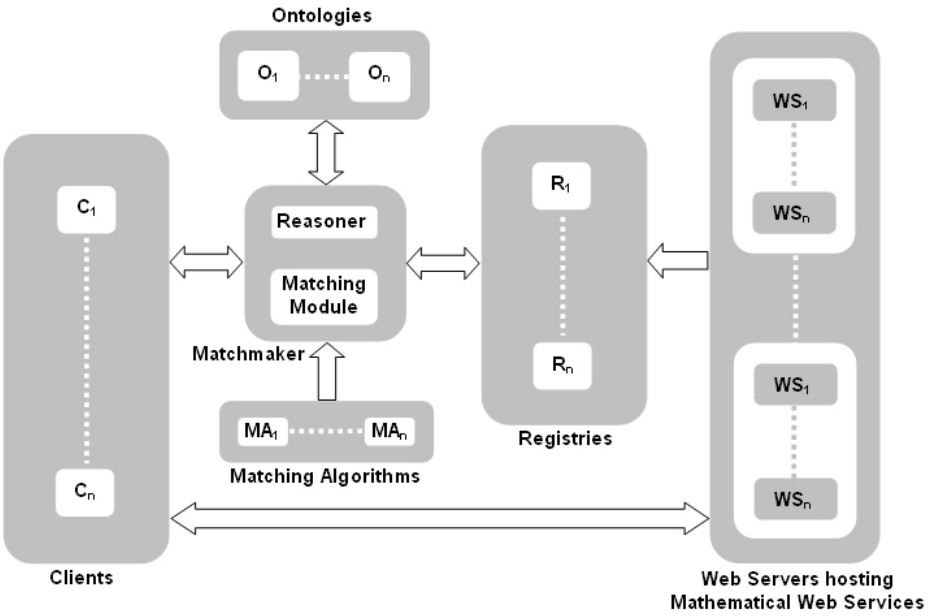


FIG. 3.1. *Matchmaking Architecture*

ture of the problem, it is only very rarely that a task description will match exactly a capability description and so a range of reasoning mechanisms must be applied to identify candidate matches. This results in:

> **Requirement 1:** A plug-in architecture supporting the incorporation of an arbitrary number of matchers.

The second problem is a consequence of the above: there will potentially be several candidate matches and some means of indicating their suitability is desirable, rather than picking the first or choosing randomly. Thus:

> **Requirement 2:** A ranking mechanism is required that takes into account pure technical (as discussed above in terms of signatures and pre- and post-condition) and quantitative and qualitative aspects—and even user preferences.

3.4. Matchmaking Architecture. Our matchmaking architecture is outlined in Figure 3.1 and comprises the following parts:

- Client interface: which may be employed by the user to specify their service request. The client interface may be a Web browser, or may be a special purpose interface that has been embedded within an existing application.
- Matchmaker: which contains a reasoning engine and the matching module. The reasoning engine relies on the existence of one or more domain specific ontologies.
- Matching algorithm Web Services: where the logic of the matching mechanism is defined. These Web Services primarily provide the plug-in capability that has been identified in Requirement 1 above. Currently, there is no mechanism to combine multiple match algorithms. Each algorithm therefore must be used independently.
- Mathematical ontologies: such as OpenMath CDs, GAMS etc. These ontologies need to be used alongside the matchmaker and the reasoner.
- Registry service: where the mathematical service descriptions are stored. A registry in this context contains the set of "advertisement" (or metadata) that define numerical algorithms that have been made available for use. In our architecture, we assume that these algorithms have been implemented as Web Services. The registry does not contain any executables, only references to executables that are maintained elsewhere.
- Mathematical Web Services: available on third party sites, accessible over the Web. These are the real executables associated with descriptions that are maintained in the registry.

The interactions of a search request are as follows:

- a user contacts the matchmaker, then
- the matchmaker loads the matching algorithms specified by the user; in the case of an ontological match, further steps are necessary;
- the matchmaker contacts the reasoner which in turn loads the corresponding ontology;
- having additional match values results in the registry being queried, to see whether it contains services which match the request and finally
- service details are returned to the user via the matchmaker. These details generally include an end point reference for the service that is being maintained on a remote file system.

The parameters stored in the registry (a database) are service name, URL/end point reference, taxonomy/ontology, input, output, pre- and post-conditions. Using contact details of the service from the registry, the user can then call the Web Service and interact with it. Each component of the architecture is now described in more detail.

3.4.1. Matching Algorithms. Currently four matching algorithms have been implemented within the matchmaker:

- structural match;
- syntax and ontological match;
- algebraic equivalence match;
- value substitution match.

These matchers are complementary and constitute the polyalgorithmic approach mentioned in the abstract. The structural match only compares the OpenMath symbol element structures (e.g. `OMA`, `OMS`, `OMV` etc.). The syntax and ontological match algorithm goes a step further and compares the OpenMath symbol elements and the content dictionary values of `OMS` elements. If a syntax match is found, which means that the values of the content dictionary are identical, then no ontology match is necessary. If an ontology match is required, the query structure is matched using the content dictionary hierarchy. The algebraic equivalence match and value substitution match do actual mathematical reasoning using the OpenMath structure.

The **structural match** works as follows: the pre- and post-conditions are extracted and an SQL query is constructed to find the same OpenMath structure of the pre- or post-conditions of the service descriptions in the database.

The **ontological match** is performed similarly, however, the OpenMath elements are compared with an ontology [25] representing the OpenMath elements. The matchmaking mechanism allows a more efficient matchmaking process by using mathematical ontologies such as the one for sets shown in Figure 3.2. OWL-JessKB [26] was used to implement the ontological match. It is intended to facilitate reading Ontology Web Language (OWL) files, interpreting the information as per OWL and RDF languages, and allowing the user to query on that information. To give an example the user query contains the OpenMath element:

```
<om:OMS cd='setname1' name='Z'/>
```

and the service description contains the OpenMath element:

```
<om:OMS cd='setname1' name='P'/>
```

which refer to the set of integers and the set of positive prime numbers respectively.

The query finds the entities `Z` and `P` and determines the similarity value depending on the distance between the two entities (inclusive, on one side) which in this case is $SV = \frac{1}{n} = 0.5$, where n is the number of nodes in the ontology that separate `Z` from `P`. As can be seen from figure 3.2 this is 2—the figure also identifies that concept `P` is subsumed by concept `Z`. Our use of distance in this context therefore differs from how it is used in general when evaluating similarity between concepts within an ontology. A detailed description of the match making algorithm can be found in [1].

For both the ontological and structural match, it is necessary that the pre- and post- conditions are in some standard form. For instance, consider the algebraic expression $x^2 - y^2$, this could be represented in OpenMath as:

```
<om:OMOBJ><om:OMA>
 <om:OMS cd="arith1" name="minus"/>
 <om:OMA>
   <om:OMS cd="arith1" name="power"/>
   <om:OMV name="x"/>
```

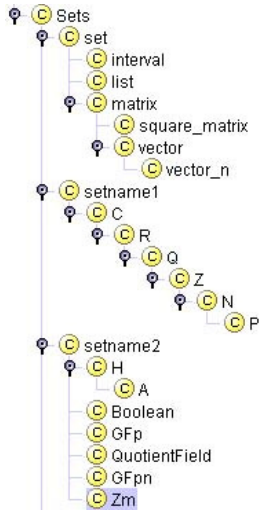


FIG. 3.2. *Set Ontology Fragment*

```
    <om:OMI>2</om:OMI>
  </om:OMA>
  <om:OMA>
    <om:OMS cd="arith1" name="power"/>
    <om:OMV name="y"/>
    <om:OMI>2</om:OMI>
  </om:OMA></om:OMA>
</om:OMOBJ>
```

however, $x^2 - y^2 = (x + y)(x - y)$, leading to the ontologically and structurally different markup:

```
<om:OMOBJ><om:OMA>
  <om:OMS cd="arith1" name="times"/>
  <om:OMA>
    <om:OMS cd="arith1" name="plus"/>
    <om:OMV name="x"/>
    <om:OMV name="y"/>
  </om:OMA>
  <om:OMA>
    <om:OMS cd="arith1" name="minus"/>
    <om:OMV name="x"/>
    <om:OMV name="y"/>
  </om:OMA></om:OMA>
</om:OMOBJ>
```

Both are "right", it just depends on what information is wanted, so there can in general be no canonical form. So in order to address the above observation, we must look deeper into the mathematical structure of the expressions which make up the post-conditions. Most of the conditions examined may be expressed in the form: Q(L(R)) where:

- Q is a quantifier block, e.g. $\forall x \exists y$ s.t. $\cdots$
- L is a block of logical connectives, e.g. $\wedge, \vee, \Rightarrow, \cdots$
- R is a block of relations, e.g. $=, \leq, \geq, \neq, \cdots$

Processing the Quantifier Block. In most cases, the quantifier block will just be a range restriction. In other cases it may be possible to use *quantifier elimination* to replace the quantifier block by an augmented logical block. Quantifier elimination is a problem for which code exists in many computer algebra systems; e.g. *RedLog* in Reduce.

Processing the Logical Block. Once the quantifier elimination has been performed on the query descriptions and the service descriptions, the resulting logical blocks must be converted into normal forms. The logical block

of both the service and query descriptions are converted to disjunctive normal form (DNF—that is a form which only contains a disjunction of conjunctions of terms and term negations). We need to now calculate a value which determines how well equation (1.1) is satisfied. That is to say, the pre-conditions of the service must be satisfied by the pre-conditions of the query and the post-conditions of the query must be satisfied by the post-conditions of the service. In all the following , we consider pre- and post- conditions in DNF, so $x \in C_{pre}$ means *x is a conjunct in the DNF for the service pre-condition.* Superfluous service pre-conditions (query post-conditions) do not effect whether the function may be performed. It is necessary, however, that there are no extra query pre-conditions (service post-conditions) as this might allow the client to provide conditions incompatible with the service pre-condition (service post-condition). This may be formalised in the following:

$$\forall x_1 \in T_{pre} \; \exists y_1 \in C_{pre} \;\; \text{s.t. } \mathrm{x}_1 \Rightarrow \mathrm{y}_1 \tag{3.1}$$

and

$$\forall x_2 \in C_{post} \; \exists y_2 \in T_{post} \;\; \text{s.t. } \mathrm{x}_2 \Rightarrow \mathrm{y}_2 \tag{3.2}$$

One way of proceeding is to treat the pre- and post- conditions separately in order to get two similarity values $\mathcal{S}_{pre}$ and $\mathcal{S}_{post}$. If it so happens that the pre- and post- conditions are equally important, then the average of these values will provide a good measure for the similarity value, however this will not always be the case, and other feasible measures are to weight $\mathcal{S}_{pre}$ and $\mathcal{S}_{post}$ linearly with the number of matching disjuncts in the pre-condition match as opposed to the post-condition match. This can be justified by observing that there are a linear number of different ways for the conditions to match.

We shall denote the DNF for C_{pre} (or T_{post}) by $R = R_1 \vee \cdots \vee R_n$ and for C_{post} (or T_{pre}) by $S = S_1 \vee \cdots \vee S_{\tilde{n}}$. To calculate a value $\in [0.0, 1.0]$ indicating how well equations 3.1, 3.2 are satisfied, we shall use the formula:

$$similarity(R, S) = \sum_{i=1..\tilde{n}} M_1(R, S_i) \frac{1}{\tilde{n}} \tag{3.3}$$

where M_1 is a function which indicates how well the expression $S_i \Rightarrow R$ holds. This is equivalent to stating how well S_i matches with one of the conjuncts making up R. A good formula to calculate this is:

$$M_1(R, S_i) = \max_{j=1\cdot\cdot n} \{M_2(R_j, S_i)\} \tag{3.4}$$

where M_2 is a similarity function for conjuncts. We may calculate a value for $M(R_i, S_j)$ as:

$$M_2(R_j, S_i) = \sum_{k=1\cdot\cdot\delta} m(R_j, S_{i,k}) \frac{1}{\delta} \tag{3.5}$$

where δ is the number of terms in S_i, $S_{i,k}$ are terms in S_i and:

$$\begin{array}{ll} m(R_j, S_{i,k}) \text{ returns} & \begin{array}{l} 1.0 \text{ if } S_{i,k} \text{ matches a term in } R_j, \\ 0.0 \text{ otherwise.} \end{array} \end{array} \tag{3.6}$$

Processing the Relations Block. In order to perform the term matches necessary to calculate 3.6 we shall consider two possible methods. It is useful to note that a term is of the general form: $T_L \succ T_R$ where $\succ$ is some relation i. e. a predicate on two arguments. In the case that T_L and T_R are real values, we may proceed as follows: we have two terms we wish to compare $Q_L \succ Q_R$ and $S_L \succ S_R$, we first isolate an output variable r, this will give us terms $r \succ Q$ and $r \succ S$. There are two approaches which we now try in order to prove equivalence of $r \succ Q$ and $r \succ S$:

- **Algebraic equivalence**: With this approach we try to show that the expression $Q - S = 0$ using algebraic means. There are many cases were this approach will work, however it has been proved [27] that in general this problem is undecidable. Another approach involves substitution of r determined from the condition $r \succ S$ into $r \succ Q$, and subsequently proving their equivalence.
- **Value substitution**: With this approach we try to show that $Q - S = 0$ by substituting random values for each variable in the expression, then evaluating and checking to see if the valuation we get is zero. This is evidence that $Q - S = 0$, but is not conclusive, since we may have been unlucky in the case that the random values coincide with a zero of the expression.

3.4.2. Service Registry. The mathematical service descriptions are stored in a database comprising the following tables: service name, taxonomy, input, output, pre- and post-conditions, and omsymbol. For the matching of pre- and post-conditions, the tables omsymbol, precond and postcond are used. The other tables give additional details about a service once the matching is done, in order for the user to select the appropriate service from the returned list.

3.4.3. Matchmaker. For all services in the database, first the pre-conditions are read and for each the matching algorithm selected is applied—which returns a similarity value. For all similarity values of pre-conditions a match value is calculated and stored. The same procedure is then used for the post-conditions. For each service the match values for all pre- and post-conditions are calculated and stored together with the service details. The methods are those detailed in section 3.4.1 resulting in a match score in the interval $[0, 1]$.

4. Application Example. For the case study we only consider the four matching modes. The Factorisor service we shall look at is a service which finds all prime factors of an Integer. The Factorisor has the following post-condition:

```
<om:OMOBJ>
  <om:OMA>
    <om:OMS cd ='relation1' name ='eq'/>
    <om:OMV name ='n'/>
    <om:OMA>
      <om:OMS cd ='fns2' name ='apply_to_list'/>
      <om:OMS cd ='arith1' name ='times'/>
      <om:OMV name ='lst_fcts'/>
    </om:OMA>
  </om:OMA>
</om:OMOBJ>
```

where `n` is the number we wish to factorise and `lst_fcts` is the output list of factors.

As the structural and ontological modes compare the OpenMath structure of queries and services, and the algebraic equivalence and substitution modes perform mathematical reasoning, the case study needs to reflect this by providing two different types of queries.

For the structural and ontological mode let us assume that the user specifies the following query:

```
<om:OMOBJ>
  <om:OMA>
    <om:OMS cd ='fns2' name ='apply_to_list'/>
    <om:OMS cd ='arith1' name ='plus'/>
    <om:OMV name ='lst_fcts'/>
  </om:OMA>
</om:OMOBJ>
```

For the structural match, the query would be split into the following `OM` collection: `OMA`, `OMS`, `OMS`, `OMV` and `/OMA` in order to search the database with this given pattern. The match score of the post-condition results in a value of 0.27778 using the equations described earlier.

The syntax and ontology match works slightly different as it also considers the *values* of the `OM` symbols. In our example we have three `OM` symbol structures. There are two instances of `OMS` and one of `OMV`. First the query and the service description are compared syntactically. If there is no match, then the ontology match is called for the `OMS` structure. The value of the content dictionary (CD) and the value of the name are compared using the ontology of that particular CD. In this case the result is a match score of 0.22222. If the `OM` structure of the service description is exactly the same as the query then the structural match score is the same as for the syntax and ontology match.

The post-condition for the Factorisor service represents:

$$\mathtt{n} = \prod_{i=1}^{l} \mathtt{lst\_fcts}_i \qquad \text{where } \mathtt{l} = |\mathtt{lst\_fcts}| \tag{4.1}$$

Considering the algebraic equivalence and the value substitution, a user asking for a service with post-condition:[2]

$$\forall i | 1 \leq i \leq |\texttt{lst\_fcts}| \Rightarrow n \bmod \texttt{lst\_fcts}_i = 0 \wedge \\ m \notin \{\texttt{lst\_fcts}_1, \cdots, \texttt{lst\_fcts}_l\} \Rightarrow n \bmod |m| \neq 0 \tag{4.2}$$

should get a match to this Factorisor service.
To carry out the algebraic equivalence match we use a proof checker to show that:

- equation (4.1) ⇒ equation (4.2): This is clear since the value of n (the RHS of equation 4.1) may be substituted into equation (4.2) and the resulting equality will be true for each value in `lst_fcts`.
- equation (4.2) ⇒ equation (4.1): The first term in the conjunct holds by the definition of **mod**, whilst the second term says that there are no other numbers which divide n.

To compute the value substitution match we must gather evidence for the equivalense of expressions 4.1 and 4.2 by taking a number of random values and substituting these into the pre- and post-conditions, since we are checking the truth of relations, these will be trivially satisfied (or not). This will however only give us evidence for the equivalence of the conditions, as we may have chosen values for which the expressions just happened to be true. So this technique can only give a probability of correctness. In order to know the probability of correctness after a certain number of independent tests, it is necessary to know the size of the zero sets for the expressions which make up the pre- and post-conditions. Another limitation of this technique, is that (in its simplest form) it can only be applied to real valued expressions. We shall consider how the technique may be applied to the above problem:

- We first need to decide on the length of the list for our random example. A good basis would be to take $|\texttt{lst\_fcts}| = \lceil \log_2(\texttt{n}) \rceil$, this represents a bound on the number of factors in the input number.
- We then collect that number of random numbers, each of size bounded by $\sqrt{\texttt{n}}$.
- Then we calculate their product, from equation (4.1), this gives a new value for `n`.
- We may now check equation (4.2). We see that it is true for every value in `lst_fcts`.

If we try this for a few random selections, we obtain evidence for the equivalence of equations (4.1) and (4.2).

5. Measurements. Of the four matching algorithms specified in section 3.4.1, our performance measurements are primarily focused on the "ontological match". This is because this mechanism provides the most computationally intensive match requirement. Structural match can already be undertaken with a variety of XML-based tools—that make use of XPath query, for instance. Similarly, algebraic equivalence also relies on the use of an ontological match to rewrite a mathematical expression prior to undertaking a match.

A set of measurements are described that: (1) evaluate the time it takes to load the MONET ontologies—briefly described in section 5.1, (2) the associated query response times and (3) the overall scalability of the ontological mode. Scalability analysis involved increasing the number of services hosted in the registry to 100,000.

5.1. The MONET Ontologies. The ontology importation graph in Figure 5.1 essentially outlines the relationships between the various ontologies[3] (the Ontology Web Language (OWL) is used to describe each ontology) that were originally developed in the MONET project [31] for the purpose of demonstrating the basic end-to-end functionality from problem statement, through service discovery and invocation, to the delivery of results. It should be emphasized that with two notable exceptions—OpenMath and GAMS—these are not general-purpose ontologies but were engineered to meet a specific short-term need. Nevertheless, significant care went into their relative organization with a view to future developments. From the application's perspective, there is just the one ontology called MONET and that imports all the others, which is where we find the real content:

- **GAMS (Guide to Available Mathematical Software):** is described as "A cross-index and virtual repository of mathematical and statistical software components of use in computational science and engineering"[4] and provides a human browseable taxonomy of mostly numerical software, for example the extract given in Figure 5.2 is that used to classify numerical quadrature routines. As this extract makes clear, GAMS is represented as textual data and is intended for human consumption. Thus for program use an OWL analogue was generated making each node in the taxonomy into an OWL class, as shown in Figure 5.3.

[2] In this example, all factors are assumed prime (this could be given as another post-condition).
[3] `http://users.cs.cf.ac.uk/Simone.Ludwig/ontologies/monet/monet.owl`
[4] `http://gams.nist.gov/`

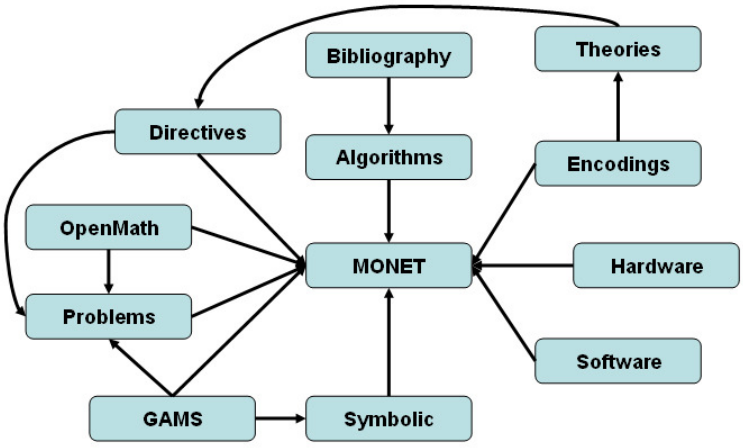


FIG. 5.1. *Relationship between MONET Ontologies*

- **Symbolic:** The GAMS taxonomy has a single category ("O") for symbolic computation that contains just four links to general-purpose computer algebra packages and one link to the ACM TOMS collected algorithms. Because a major focus of MONET was the combined application of numerical and symbolic techniques, this branch has undergone some extension to enable the description and advertisement of a range of symbolic mathematical services, such as group theory operations, indefinite integration, power series and algebraic geometry.
- **OpenMath:** This is not OpenMath[5] that we have used for mathematical semantic markup, but rather the interface to that ontology. Because OpenMath development started many years before OWL it does not (yet) use many of the current ontology technologies and is simply an XML-based encoding format for the representation of mathematical expressions and objects. Terms (referred to as "Constants") of the language have semantics attached to them and are called symbols (e.g., sin, integral, matrix, etc.), and groups of related symbols are collected in content dictionaries (CDs). What the OpenMath component of the MONET ontology defines is OWL classes for each of the symbols in the current OpenMath CDs.
- **Hardware:** This is used to describe either machine types or individual machines. The idea is that a user might request that a service run on a particular architecture (e.g. Sun Enterprise 10000), a general class of machine (e.g. shared memory), or a machine with a certain number of processors. Here is a fragment that describes a Sun shared memory machine:

```
<owl:Class rdf:ID="Enterprise10000">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#SharedMemory"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Sun"/>
  </rdfs:subClassOf>
</owl:Class>
```

- **Software:** allows a user to express a preference for a service that makes use of a particular programming language or software library, for example it represents a variety of programming languages (FORTRAN 77, FORTRAN 2000, C, etc.) and numerous packages (NAG library version 7 in C, Maple release 8, etc.).
- **Problems:** may be described in terms of inputs and outputs, pre-conditions and post-conditions, and make use of pre-defined XML schema [32]. Within this ontology, each problem is represented as a class, which can have properties indicating bibliography entries and their generalizations. The most interesting property is `openmath head` whose range is an object from the `OpenMathSymbol` class. This represents a particular symbol which can be used to construct an instance of the problem in question.

[5] `http://www.openmath.org`

```
H2.     Quadrature (numerical evaluation of definite integrals)
H2a.       One-dimensional integrals
H2a1.         Finite interval (general integrand)
H2a1a.           Integrand available via user-defined procedure
H2a1a1.             Automatic (user need only specify required accuracy)
H2a1a2.             Nonautomatic
```

FIG. 5.2. *An extract from the on-line GAMS taxonomy*

```
<owl:Class rdf:ID="GamsH2">
  <rdfs:comment>GAMS classification, Differentiation, integration,
                Quadrature (numerical evaluation of definite integrals)
  </rdfs:comment>
  <rdfs:label>Quadrature (numerical evaluation of definite integrals)</rdfs:label>
  <rdfs:subClassOf rdf:resource="#GamsH"/>
</owl:Class>
\dots
<owl:Class rdf:ID="GamsH2a">
  <rdfs:comment>GAMS classification, Differentiation, integration,
                Quadrature (numerical evaluation of definite integrals),
                One-dimensional integrals
  </rdfs:comment>
  <rdfs:label>One-dimensional integrals</rdfs:label>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#GamsH2"/>
  </rdfs:subClassOf>
</owl:Class>
```

FIG. 5.3. *An extract from the OWL representation of GAMS*

- **Algorithms:** there are two sub-classes in this ontology: (1) Algorithm: which describes well-known algorithms for mathematical computations, and (2) Complexity: which provides classes necessary for representing complexity information associated with an Algorithm.
- **Directives:** this ontology is a collection of classes which identify the task that is performed by the service as described in [32]—for example to decide, solve or prove a particular mathematical expression.
- **Theory:** this ontology collects classes that represent available formalized theories in digital libraries of mathematics.
- **Bibliography:** represents entries in well-known indices such as Zentralblatt MATH [33] and MathSciNet [34] and allows them to be associated with particular algorithms.
- **Encoding:** this ontology contains a (small) collection of classes which represent the formats used for encoding mathematical objects.
- **Monet:** As stated at the beginning of this section, MONET imports all the ontologies described above.

5.2. Methodology. The ontological mode of the matchmaker is based on OWLJessKB—a memory-based reasoning tool that may be used over ontologies specified in OWL. To store service descriptions, a MySQL database was used, residing on a different machine. OWLJessKB uses the Java Expert System Shell (JESS) [36] as its underlying reasoner. The OWLJessKB implementation loads multiple ontologies (the location of each is specified as a URL) into memory from a remote Web server. Reasoning in this instance involves executing one or more JESS rules over the ontology. JESS makes use of the Rete algorithm [37], which is intended to improve the speed of forward-chained rule systems (this is achieved by limiting the effort required to recompute the conflict set after a rule is fired). However, it has high memory requirements due to the loading of the ontology into the Rete object. Once it is created and set, it is fast to call rules and queries in order to infer the semantic relations of the ontology loaded. A key limiting factor in OWLJessKB is the time to load the ontology into primary memory (RAM), and the total RAM size on the host platform.

The implementation used for the ontological mode is the OWLJessKB version owljesskb20040223.jar [6]. The test environment included an Intel Pentium III processor 996MHz, 512MB RAM, and Windows XP Professional, running Java SDK 1.5.0 and Jess 6.1p8.

Measurements—Loading ontologies: These measurements include memory tests for OWLJessKB as heap size value were set for the loading of different ontology sizes. Additional measurements were carried out to evaluate the performance of loading the ontologies for the OWLJessKB implementation.

Measurements—Query response times: Four different query types were chosen. These were the following:
(1) Simple assertion: Find all instances of a class `x`;
(2) Simple assertion: Verify whether instance `x` exists;
(3) Assertion individual: Confirm if constraint `y` is satisfied via a single object;
(4) Assertion aggregate: Confirm if constraint `y` is satisfied for an entire group.

Measurements—Scalability: These measurements involved analyzing the performance of query response times for populated registries, starting with 100 services stored and going up to 100,000 services. Hence, scalability is evaluated as the change in query response behaviour as additional services were added to the registry.

5.3. Results.

5.3.1. Loading Ontology Performance. The previously described MONET ontologies (Figure 5.1) were chosen for the performance measurements. The MONET ontologies consist of 2031 classes, 78 slots and 10 facets. When increasing the ontology size, only the classes within the ontology were considered and expanded. Different ontology sizes were used having the number of classes as shown in Table 5.1.

TABLE 5.1
Ontology Sizes

Ontology size	No. of classes
1	2031
1.5	3046
2	4062
2.5	5077
3	6092
3.5	7107
4	8122

During preliminary measurement tests it was found that the OWLJessKB implementation needs the heap size in the Java Virtual Machine to be modified to load all ontology sizes. The first set of measurements were undertaken to investigate the necessary heap size required for the different ontologies. Figure 5.4 shows the memory heap size for varying ontology sizes. It shows a linear distribution starting from 109MB for ontology size 1 and ending at 847MB for ontology size 4. The regression line and the derived equation shown in the figure allow to calculate the memory heap size needed for a particular ontology size.

The maximum heap size was set to 1 GB, which was found to be a sufficient value for the measurements shown in Figure 5.4. The following set of measurements show the load time of the OWLJessKB implementation. Ten test runs were conducted for each ontology size.

5.3.2. Query Performance. The next set of measurements were carried out having four queries (simple assertion finding all instances of a class x, simple assertion asking whether instance x exists, assertion individual and assertion aggregate) to test the response times of varying ontology sizes for both target systems. The maximum heap size in the target system was set to 1 GB. For each query ten test runs were conducted and the average values were taken.

Queries for the OWLJessKB implementation need to be specified in the JESS notation, which is the following:

```
(defquery query-sub-class ‘‘Find all sub-classes"
  (triple
    (predicate ‘‘http://www.w3.org/2000/01/rdf-schema#
```

[6] `http://edge.cs.drexel.edu/assemblies/software/owljesskb/`

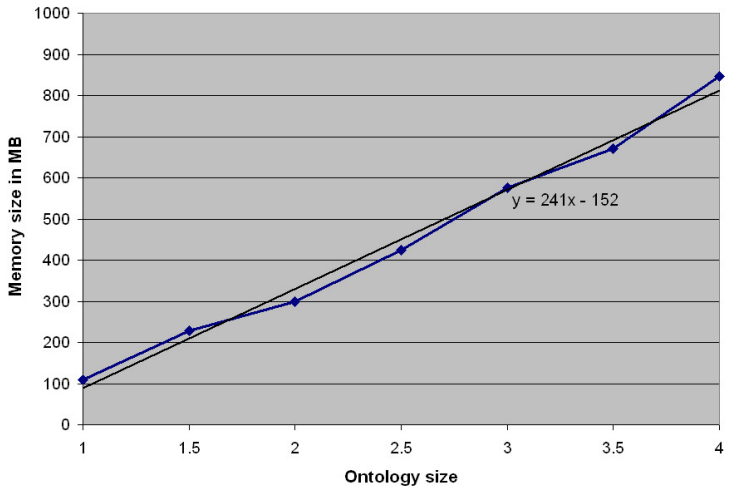


FIG. 5.4. *Memory heap size*

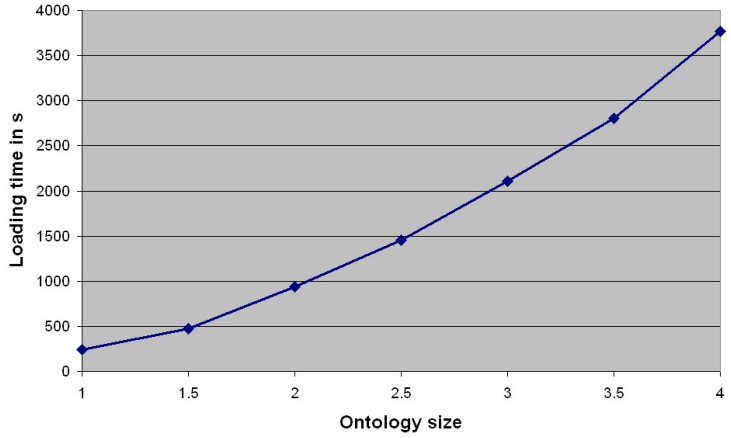


FIG. 5.5. *Load time*

```
            subClassOf")
    (subject ‘‘http://monet.nag.co.uk/owl#
             service_algorithm")
    (object ?y)))
```

A service stored in the registry is `nagopt`—fitting the GAMS taxonomy `G1a1a` [6] (a variant of unconstrained optimisation) and using OpenMath as its I/O format. In this case the description also indicates that the service uses NAG's implementation of the safeguarded quadratic-interpolation algorithm.

```
<service name="nagopt">
    <classification>
        <gams_class>GamsG1a1a</gams_class>
        <problem>constrained_minimisation</problem>
        <input_format>OpenMath</input_format>
        <output_format>OpenMath</output_format>
        <directive>find</directive>
    </classification>
    <implementation>
        <software>NAG_C_Library_7</software>
        <platform>PentiumSystem</platform>
```

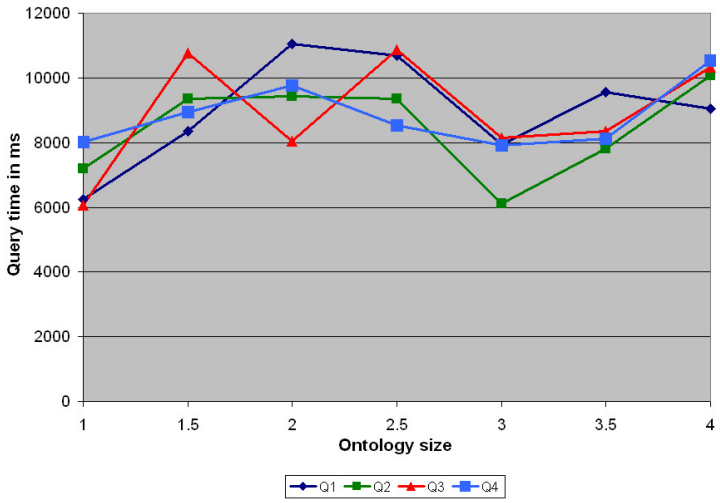


FIG. 5.6. *Query Performance*

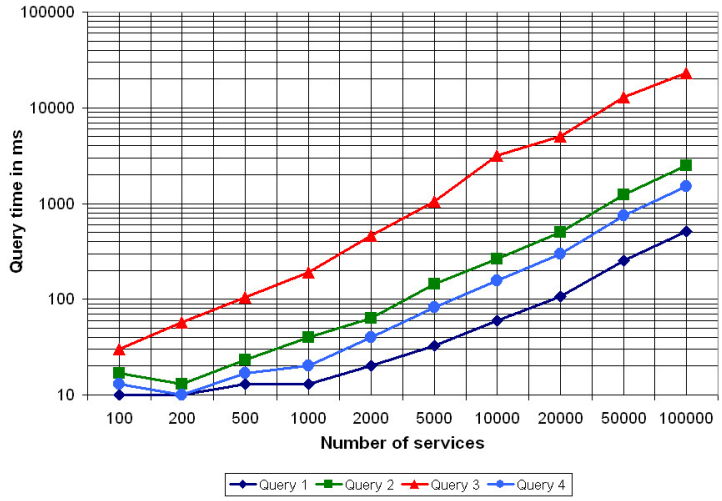


FIG. 5.7. *Scalability of populated services*

```
        <algorithm>Safeguarded_Quadratic-Interpolation</algorithm>
    </implementation>
</service>
```

In Figure 5.6 the response time of the four queries are shown. No increase in query response time for larger ontology sizes can be seen, which means that the ontology size does not increase the query response time. The average deviation (specifying measurement accuracy) is 1.8 ms. Please note that Q1, Q4 times identical after ontology of size 2.

5.3.3. Scalability Performance. Measurements were taken populating the registry with an increasing number of services. Ontology size 1 was taken for this set of measurements, with a maximum heap size of 1 GB. The registry was populated with 100, 200, 500, 1,000, 2,000, 5,000, 10,000, 20,000, 50,000 and 100,000 services (therefore, Figure 5.7 use a logarithmic scale). In Figure 5.7 a linear distribution between query time and number of services can be seen. However, the time measurements for 100, 200 and 500 services in Query 1, 2 and 4 are scattered around 10 ms and 20 ms, this is due to the measurement accuracy.

6. Conclusion and Further Work. We have presented an approach to matchmaking in the context of mathematical semantics. The additional semantic information greatly assists in identifying suitable services in

some cases, but also significantly complicates matters in others, due to their inherent richness. Consequently, we have put forward an extensible matchmaker architecture supporting the dynamic loading of plug-in matchers that may employ a variety of reasoning techniques, including theorem provers and computer algebra systems as well as information retrieval from textual documentation of mathematical routines (this last is under development at present). Although our set of application examples is as yet quite small, the results are promising and we foresee the outputs of the project being of widespread utility in both the e-Science and Grid communities, as well as more generally advancing semantic matchmaking technology. The performance measurements conducted showed the scalability of the system with the drawback of a relatively long ontology load time. Although the focus here is on matchmaking mathematical capabilities, the descriptive power, deriving from quantification and logic combined with the extensibility of OpenMath creates the possibility for an extremely powerful general purpose mechanism for the description of both tasks and capabilities. In part, this appears to overlap, but also to complement the descriptive capabilities of OWL and, in much the same way as it was applied in MONET, we expect to utilise OWL reasoners as plug-in matchers in the architecture we have set out.

REFERENCES

[1] S. A. Ludwig, O. F. Rana, J. Padget and W. Naylor, *Matchmaking Framework for Mathematical Web Services,* Journal of Grid Computing, Springer Verlag, 2006.

[2] M. Nodine, W. Bohrer and A. H. Ngu, *Semantic brokering over dynamic heterogenous data sources in InfoSleuth,* Proceedings of the 15th International Conference on Data Engineering, pp. 358-365, 1999.

[3] K. Sycara and S. Widoff and M. Klusch and J. Lu, *Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace,* Journal of Autonomous Agents and Multi Agent Systems, Kluwer, 2002.

[4] V. R. Benjamins and B. Wielinga and J. Wielemaker and D. Fensel, *Towards Brokering Problem-Solving Knowledge on the Internet,* Ed. Dieter Fensel and Rudi Studer, Proceedings of the 11th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW-99), LNAI1621, Springer, 1999.

[5] M. Gomez and E. Plaza, *Extended matchmaking to maximize capability reuse,* Proceedings of The Third International Joint Conference on Autonomous Agents and Multi Agent Systems, ACM Press, 2004.

[6] MONET Consortium, MONET Home Page, Available from \tthttp://monet.nag.co.uk 2002.

[7] D. Kuokka and L. Harada, *Matchmaking for information integration,* Journal of Intelligent Information Systems, Feb 1996.

[8] O. Caprotti and M. Dewar and J. Davenport and J. Padget, *Mathematics on the (Semantic) Net,* Proceedings of the European Symposium on the Semantic Web, Springer Verlag, 2004.

[9] The GENSS Project, GENSS Home Page, Available from http://genss.cs.bath.ac.uk 2004.

[10] OpenMath Society, OpenMath website, Available from http://www.openmath.org, 2002.

[11] W3C MathML, *Mathematical Markup Language (MathML) Version 2.0,* W3C, Available from http://www.w3.org/TR/MathML2/, 2003.

[12] W. Naylor and S. Watt, *Meta-stylesheets for the conversion of mathematical documents into multiple forms,* Annals of Mathematics and Artificial Intelligence Journal, vol. 38, no. 1–3, May, 2003.

[13] S. Buswell and O. Caprotti and M. Dewar, *Mathematical Service Description Language,* Available from the MONET website: http://monet.nag.co.uk/cocoon/monet/publicdocs/monet-msdl-final.pdf, 2003.

[14] A. Ankolekar, M. Burstein, J.R. Hobbs, O. Lassila, D.L Martin, S.A. McIlraithe, S. Narayanan, M. Paolucci, T. R. Payne, K. Sycara, H. Zeng, *DAML-S: Semantic Markup for Web Services,* Proceedings of 1st International Semantic Web Conference (ISWC 02), 2002.

[15] W3C Coalition, *OWL-S: Semantic Markup for Web Services,* In Technical White paper (OWL-S version 1.0), 2003.

[16] V. R. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, B. Wielinga, G. Schreiber, Z. Zdrahal and S. Decker, *An Intelligent Brokering Service for Knowledge-Component Reuse on the World-WideWeb,* Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW98), Banff, Canada, 1998.

[17] R. J. Brachman and J. G. Schmolze, *An overview of the KL-ONE knowledge representation system,* Cognitive Science, 9(2): 171–216, 1985.

[18] The OpenMath Society, The OpenMath Standard, The OpenMath Society, Available from http://www.openmath.org/cocoon/openmath/standard/om20/index.html, 2004.

[19] D. Kuokka and L. Harada, *Integrating information via matchmaking,* Journal of Intelligent Information Systems 6(2-3), pp. 261–279, 1996.

[20] T. Finin and R. Fritzson and D. McKay and R. McEntire, *KQML as an agent communication language,* Proceedings of 3rd International Conference on Information and Knowledge Management, pp. 456–463, 1994.

[21] M. Genesereth and R. Fikes, *Knowledge interchange format,* version 3.0 reference manual, Computer Science Department, Stanford University, Available from http://www-ksl.stanford.edu/knowledge-sharing/papers/kif.ps, 1992.

[22] G. Salton, *Automatic Text Processing,* Addison-Wesley, 1989.

[23] D. Veit, *Matchmaking in Electronic Markets,* An Agent-Based Approach towards Matchmaking in Electronic Negotiations, Springer, LNCS2882, 2003.

[24] A. Moormann Zaremski and J. M. Wing, *Specification Matching of Software Components,* ACM Transactions on Software Engineering and Methodology, 1997.

[25] J. F. Sowa, *Ontology, Metadata, and Semiotics, Conceptual Structures: Logical, Linguistic, and Computational Issues,* Lecture Notes in AI #1867, Springer-Verlag, 2000.

[26] J. Kopena, *OWLJessKB,* Available at http://edge.cs.drexel.edu/assemblies/software/owljesskb/, 2004.

[27] D. Richardson, *Some Unsolvable Problems Involving Elementary Functions of a Real Variable,* Journal of Computational Logic, 1968.
[28] W3C WSDL, Web Services Description Language (WSDL) 1.1, Available from `http://www.w3.org/TR/wsdl`, 2004.
[29] L. Verlet, *Computer Experiments on Classical Fluids I. Thermodynamical Properties of Lennard-Jones Molecules,* Phys. Rev., Vol. 159, pp. 98–103, 1967.
[30] Maozhen Li, O.F.Rana, David W. Walker, *Wrapping MPI-Based Legacy Codes as Java/CORBA Components,* Future Generation Computer System(FGCS): The Int. Journal of Grid Computing: Theory, Methods and Applications, vol. 18, Issue 2, pages 213–223, October 2001.
[31] O. Caprotti, M. Dewar and D. Turi, *Mathematical service matching using Description Logic and OWL,* In Proceedings of 3rd Int'l Conference on Mathematical Knowledge Management (MKM'04), Vol:3119, pages 144–151. Springer-Verlag, 2004.
[32] O. Caprotti, D. Carlisle, A. Cohen and M. Dewar, *Problem Ontology: final version,* The MONET Consortium Technical Report Deliverable D11. Available from: `http://monet.nag.co.uk`
[33] Zentralblatt MATH. Available from: `http://www.emis.de/ZMATH/`
[34] American Mathematical Society, *MathSciNet: Mathematical Reviews on the Web.* Available from: `http://www.ams.org/mathscinet`.
[35] S. A. Ludwig, O. F. Rana, W. Naylor and J. Padget, *Matchmaking of Mathematical Web Services,* In Proceedings of 6th International Conference on Parallel Processing and Applied Mathematics, Poznań, Poland, September 2005.
[36] Ernest J. Friedman-Hill, *Java Expert Systems Shell.* Available from: `http://herzberg.ca.sandia.gov/jess/docs/61/index.html`
[37] C. Forgy, *Rete: A fast algorithm for the many pattern/many object pattern match problem.* Journal of Artificial Intelligence, 19: 17–37, 1982.
[38] I. Horrocks, U. Sattler and S. Tobies, *Reasoning with individuals for the description logic SHIQ.* Proceedings of the 17th International Conference on Automated Deduction (CADE-17), Springer-Verlag, 2000.

Edited by: Przemysław Stpiczyński.
Received: April 6, 2006.
Accepted: May 28, 2006.

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 53–63. http://www.scpe.org
ISSN 1895-1767

PARALLEL IMPLEMENTATION OF UNIFORMIZATION TO COMPUTE THE TRANSIENT SOLUTION OF STOCHASTIC AUTOMATA NETWORKS

HAÏSCAM ABDALLAH*

Abstract. Analysis of Stochastic Automata Networks (SAN) is a well established approach for modeling the behaviour of computing networks and systems, particularly parallel systems. The transient study of performance measures leads us to time and space complexity problems as well as error control of the numerical results. The SAN theory presents some advantages such as avoiding to build the entire infinitesimal generator and facing the time complexity problem thanks to the tensor algebra properties.

The aim of this study is the computation of the transient state probability vector of SAN models. We first select and modify the (stable) uniformization method in order to compute that vector in a sequential way. We also propose a new efficient algorithm to compute a product of a vector by a tensor sum of matrices. Then, we study the contribution of parallelism in front of the increasing execution time for stiff models by developing a parallel algorithm of the uniformization. The latter algorithm is efficient and allows to process, within a fair computing time, systems with more than one million states and large mission time values.

Key words. Parallel systems, stochastic automata networks, transient solution, uniformization, parallelism.

1. Introduction. This paper presents a parallel version of the *transient analysis* for Continuous Time Markov Chains (CTMCs) via Stochastic Automata Networks (SANs). The computation of the transient distribution of CTMC gives the main performance measures such as reliability and availability. Generally, we are facing the problem of computation time due to the explosive growth of the state space and the stiffness. SANs, introduced by Brigitte Plateau [1], may be a good solution to that problem.

The use of SANs is becoming important in performance modeling issues related to parallel and distributed computer systems [2]. Those systems are often viewed as collections of components that operate more or less independently. They require only infrequent interaction such as synchronizing their actions or operating at different rates depending on the state of parts of the overall system. The components are modeled as individual stochastic automata that interacts with each other. A module (automaton) is modeled by a set of *states*, and the event or action of the module is modeled by a *transition* from a state to another. A single automaton represents only the state of one module; additional information is used to express *interactions* among the modules. On each transition, a label gives information about the timing and the probability of events occurrence. The transitions and the events are described by some matrices, for each automaton. We focus on synchronizing dependence, i. e., *local and events matrices are constant*. Under appropriate Markovian assumptions, the behaviour of the set of automata may be modeled by a CTMC, which state space is the cartesian product of the all the space states of the automata. It has been shown that the infinitesimal generator of the resulting CTMC, also known as the *descriptor*, can be obtained automatically into a compact formula, by means of Kronecker (tensor) algebra [3]. The consequence is that the state transition matrix is not stored, not even generated.

We are interested in the computation of the transient solution of SANs while they are very often analyzed in the stationary or quasi-stationary cases [4, 5]. The challenge is to choose a method that, at the same time, bounds the global error and deals with the time complexity. Moreover, the algorithms of that method must be parallelizable. Some studies have been done in the transient case, when the state space is reasonable. Among them, IRK3 (Implicit Runge-Kutta method of order 3) has been used [6, 7]. This method deals efficiently with stiff models, for large mission time t. But unfortunately, its time complexity is unpredictable and the global error is difficult to bound. A consequence of this latter drawback is that an accuracy cannot be chosen *a priori*. The Uniformized Power technique (UP) has also been proposed [8]. This method is very fast for systems with large values of t, but only usable in the case of moderate state spaces. The Standard Uniformization method (SU) has proved its efficiency for reasonable values of t, even when the state space size is important [9, 10]. This efficiency is altered when t increases. The main advantage of this technique is the possibility of bounding the global error and predicting the time complexity. The most part of the algorithms are sequential.

In this study, we first adapt the SU method to compute the transient solution of SANs. Next, we make a parallel implementation of the SU method in order to deal with the case of large values of t. We also derive a new parallel algorithm which computes the multiplication of a vector by a Kronecker tensor sum of matrices. This implementation uses an efficient parallel multiplication of a vector by a tensor product of matrices [11].

*Dpt MASS, Université de Rennes 2 Place du Recteur Henri Le Moal, CS 24307 35043 Rennes cedex, France, haiscam.abdallah@uhb.fr

The resulting global algorithm has a speedup which remains in average greater than 80%. It allows us to deal efficiently with problems that has not been solved yet. The structure of the paper is as follows. Section 2 sets the problem and includes a detailed sequential analysis. Section 3 is dedicated to the parallel algorithms. The numerical results on an Asynchronous Transfer Mode (ATM) network are presented in Section 4. The paper is concluded with Section 5.

2. Problem formulation. We consider a SAN that consists of N individual automata, $\mathcal{A}^{(k)},\ k = 1,\ldots,\ N$. The number of states in the k^{th} automaton is denoted by n_k for $k = 1,\ 2,\ \ldots,\ N$. Let $X^{(k)}$, for $k = 1,\ldots,\ N$, be the unidimensional CTMC associated with the automaton $\mathcal{A}^{(k)}$, $Q^{(k)}$ the infinitesimal generator of $X^{(k)}$ and $\Pi^{(k)}(0)$ its initial distribution vector. Let us denote by $\Pi^{(k)}(t)$ the state probability vector of $X^{(k)}$ at time t. The overall model (the SAN) is described by a multidimensional CTMC $X = \{X_t,\ t \geq 0\}$, which state space and size are respectively $E = \prod_{k=1}^{N} E^{(k)}$ and $M = \prod_{k=1}^{N} n_k$. Its infinitesimal generator (descriptor) and its initial distribution are denoted by Q and $\Pi(0)$. At time t, the transient distribution of the CTMC X is given by the vector $\Pi(t)$. Our goal is the computation of $\Pi(t)$ for a given value of the system's mission time t. The vector $\Pi(t)$ is solution of the first homogenous linear differential equations, known as Chapman-Kolmogorov equations:

$$\frac{\partial}{\partial t}\Pi(t) = \Pi(t)\mathrm{Q};\ \ \Pi(0) \text{ given.} \tag{2.1}$$

When the automata are independent, the descriptor Q is the tensor sum of the N infinitesimal generators $Q^{(k)},\ k = 1,\ldots,\ N$, resulting from the local transitions:

$$Q = \bigoplus_{k=1}^{N} Q^{(k)}.$$

It follows that:

$$\Pi(t) = \bigotimes_{k=1}^{N} \Pi^{(k)}(t). \tag{2.2}$$

Relation (2.2) leads us to the computation of vector $\Pi^{(k)}(t),\ k = 1,\ldots,\ N$, for CTMCs with moderate state space size n_k. An efficient method, like SU or UP, can be selected according to the problem's size and mission time t.

2.1. Dependent automata. Let T be the total number of the system's synchronizing events and for all $i = 1,\ldots,\ T$, $E_{i+}^{(k)}$ the matrix of event i over automaton $\mathcal{A}^{(k)}$, $k = 1,\ldots,\ N$; $E_{i-}^{(k)}$ is the regularisation matrix. The descriptor Q is then expressed as an ordinary sum of a local part and a synchronizing part [3, 4]:

$$Q = \bigoplus_{k=1}^{N} Q^{(k)} + \sum_{i=1}^{2T} \bigotimes_{k=1}^{N} E_i^{(k)} \ \text{ where } \ E_i^{(k)} \in \left\{E_{i+}^{(k)}, E_{i-}^{(k)}\right\}. \tag{2.3}$$

It is important to note that $\bigoplus_{k=1}^{N} Q^{(k)}$ can be transformed into ordinary sum of tensor products of matrices as follows

$$\bigoplus_{k=1}^{N} Q^{(k)} = \sum_{k=1}^{N} I_{n_1} \otimes \cdots \otimes I_{n_{k-1}} \otimes Q^{(k)} \otimes I_{n_{k+1}} \otimes \cdots \otimes I_{n_N}, \tag{2.4}$$

where $I_{n_k},\ k = 1,\ \ldots,\ N$, is the n_k order identity matrix. Consequently, the descriptor Q can be written under the form:

$$Q = \sum_{i=1}^{N+2T} \bigotimes_{k=1}^{N} Q_i^{(k)}, \ \text{ with } \ Q_i^{(k)} \in \{I_{n_i}, Q^{(k)}, E_i^{(k)}\}. \tag{2.5}$$

Most of the time, expression (2.5) is used to solve the overall model (the CTMC X), in order to transform the problem into the computation of a product vector-matrix, where the matrix is a tensor product of several

matrices. *We adopt expression (2.3) to compute the transient distribution.* Indeed, we develop, at the end of this section, a specific algorithm that computes the product of a vector by a tensor sum of matrices. This algorithm has the same time complexity as that which computes the product of a vector by a tensor product of matrices. Consequently, compared to (2.5), the form (2.3) allows an important saving in time complexity.

Let us remind that the major problem is the choice of methods that solve (2.1) taking into account the global error. An efficient error control mechanism is used by UP and SU methods. The UP technique performs well for CTMCs with large values of t and moderate state spaces while the SU method works better for CTMCs with moderate values of t and large state spaces. *The basic idea is, first of all, to implement the SU method for the SAN and next, analyze the contribution of parallelism when t increases.*

For $Q = (q_{ij})_{i,j=1,\,...,\,M}$ given, the expression of $\Pi(t)$ obtained by the SU method is [6, 12]:

$$\Pi(t) = \sum_{n=0}^{\infty} p(n, qt)\Pi(0)\tilde{P}^n, \tag{2.6}$$

where $q \geq \ max_{1\leq i\leq M} \mid q_{ii} \mid$, $p(n, qt) = e^{-qt}\frac{(qt)^n}{n!}$, and $\tilde{P} = I + Q/q$; I is the M order identity matrix.

The previous infinite sum can be truncated at a step N_T such that

$$1 - \sum_{n=0}^{N_T} p(n, qt) \leq \varepsilon, \tag{2.7}$$

where ε is a tolerance, given *a priori* by the user. That tolerance also bounds the global error on $\Pi(t)$ by SU. Let us note that for a given value of ε, N_T is always greater than qt.

Let $\tilde{\Pi}^{(n)}$, $n = 1, \ldots,\ N_T$, be the vector $\Pi(0)\tilde{P}^{(n)}$. These N_T vectors are computed by the following recurrence relation:

$$\tilde{\Pi}^{(n)} = \tilde{\Pi}^{(n-1)}\tilde{P}\ ,\ n \geq 1\ ;\ \tilde{\Pi}^{(0)} = \Pi(0). \tag{2.8}$$

From a time complexity point of view, the computation of $\Pi(t)$ requires one vector-matrix product by iteration (relation (2.8)). Using a compact storage for Q, the time complexity may be reduced to $O(N_T\eta)$ where η is the number of non zero elements in Q.

The SAN methodology has a first advantage of avoiding to build the whole generator Q. Using relations (2.3) and (2.8), we have:

$$\begin{aligned}\tilde{\Pi}^{(n)} = \tilde{\Pi}^{(n-1)} &+ \frac{1}{q}\left[\tilde{\Pi}^{(n-1)} \oplus_{k=1}^{N} Q^{(k)}\right] \\ &+ \frac{1}{q}\sum_{i=1}^{2T}\left[\tilde{\Pi}^{(n-1)} \otimes_{k=1}^{N} E_i^{(k)}\right],\ \ n \geq 1,\end{aligned} \tag{2.9}$$

with $\tilde{\Pi}^{(0)} = \Pi(0) = \otimes_{k=1}^{N}\Pi^{(k)}(0)$ given.

Another advantage of the SAN is the possibility of developing specific sequential and parallel algorithms to compute a vector-tensor product or sum of matrices. Those algorithms are often faster than the classical ones for which Q is entirely given (cf. 2.2.2 and 3).

2.2. Sequential approach.

2.2.1. Product of a vector by a tensor product of matrices. The computation of the vector $y = x\bigotimes_{k=1}^{N} A^{(k)}$ given the vector x and the N (small) matrices $A^{(k)}$, can be done by expressing each element of $\bigotimes_{k=1}^{N} A^{(k)}$ as a product of N elements of $A^{(k)}$, $k = 1, ..., N$. The time complexity of the algorithm is $O(N\prod_{k=1}^{N}\eta(A^{(k)}))$, where $\eta(A^{(k)})$ is the number of non-zero elements in the matrix $A^{(k)}$.
Because this time complexity is generally very large and the matrix $A^{(k)}$ (here $Q^{(k)}$) have a small value of $\eta(A^{(k)})$, an algorithm based on the perfect shuffle is used [1, 13]. Such an algorithm, called $TENS$, has the following time complexity

$$O\left(M\sum_{k=1}^{N}\frac{\eta\left(A^{(k)}\right)}{n_k}\right) \tag{2.10}$$

Let $\alpha_k = \frac{\eta(A^{(k)})}{n_k}$ be the mean number of non-zero terms in rows or columns of $A^{(k)}$ and let us set it to α. It is clear that an algorithm based on the perfect shuffle is better than a classical one if $\alpha > N^{\frac{1}{N-1}}$. The SANs satisfy very often the case.

2.2.2. Product of a vector by a tensor sum of matrices. For computing the vector $z = x \bigoplus_{k=1}^{N} A^{(k)}$, we transform $\bigoplus_{k=1}^{N} A^{(k)}$ into an ordinary sum of tensor product of matrices using relation (2.4). More precisely, if we define

$$M_l^u = \prod_{k=u}^{l} n_k \tag{2.11}$$

and $\bar{n}_k = M/n_k$, we have

$$x \bigoplus_{k=1}^{N} A^{(k)} = \sum_{k=1}^{N} x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right). \tag{2.12}$$

The computation of the vector z is based on the product

$$x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right). \tag{2.13}$$

At each iteration k, all the matrices, except $A^{(k)}$, are set to identity matrix. Algorithm 1, called $TENS_k$ (k^{th} iteration of $TENS$), is a particular case of a perfect shuffle to compute (2.13). In this algorithm, we consider $n_{left} = M_1^{k-1}$ and $n_{right} = M_{k+1}^N$. The time complexity of the algorithm $TENS_k$ is $O\left(\bar{n_k} \,.\, \eta\left(A^{(k)}\right)\right)$. Lines 5-8 and 11-16 describe the permutations required by this algorithm. If $TENS^+$ is the algorithm which computes z by calling N times $TENS_k$, the time complexity of such an algorithm is

Input : $n_k,\ A^{(k)},\ x,\ n_{left},\ n_{right}$
Output : $Y = x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^K} \right)$

1: $base \leftarrow 0\ ;\ jump \leftarrow n_k.n_{right}$
2: **for** $block = 0, \ldots,\ n_{left} - 1$ **do**
3: **for** $offset = 0, \ldots,\ n_{right} - 1$ **do**
4: $index \leftarrow base + offset$
5: **for** $h = 0, \ldots,\ n_k - 1$ **do**
6: $Z_h \leftarrow x_{index}$
7: $index \leftarrow index + n_{right}$
8: **end for**
9: $Z' = Z.A^{(k)}$
10: $index \leftarrow base + jump$
11: **for** $h = 0, \ldots,\ n_k - 1$ **do**
12: $Y_{index} \leftarrow Y_{index} + Z'_h$
13: $index \leftarrow index + jump$
14: **end for**
15: **end for**
16: $base \leftarrow base + jump$
17: **end for**

Algorithm 1: Algorithm $TENS_k$ for computing $Y = x \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right)$

$$O\left(\sum_{k=1}^{N} \bar{n}_k \,.\, \eta\left(A^{(k)}\right) \right) = O\left(M \sum_{k=1}^{N} \frac{\eta\left(A^{(k)}\right)}{n_k} \right) \tag{2.14}$$

It is important to note that this time complexity is identical to that of the computation of $x \bigotimes_{k=1}^{N} A^{(k)}$ given by relation (2.10). In the following section, we give a parallel version of this algorithm.

3. Parallel implementation. Our goal is the parallelization of the computation algorithm of the vector $\Pi(t)$ based on relation (2.9). Taking into account the quantity of data (matrices and vectors) to be processed, it is quite necessary to choose a program scheme in which each processor owns some data, to which it applies some instruction streams. These instruction streams can be chosen identical for all the processors (SPMD: Single Program, Multiple Data). This scheme is easier to implement than the case where several algorithms are built, one for each processor (MIMD: Multiple Instruction streams, Multiple Data). We place ourselves in the SPMD mode. A crucial problem in this kind of implementation is the load balancing. Each node must have the same amount of work in order to assure a certain equilibrium and efficiency. It is therefore important to use a good decomposition and task repartition technique. This repartition must minimize a function, relative to the execution time of the program over P processors.

In order to make a parallel implementation in SPMD mode over P processors, we need a data decomposition into P subsets, determining each processor's task. This task is mainly composed with computation phases ended by synchronization phases.

The first part of our work consists in implementing relation (2.9) over P processors. At each step, this relation is essentially made with matrix-vector products, where the matrix is a tensor product or a tensor sum of matrices. First, we are going to deal with the tensor product case. Next, we shall proceed with the tensor sum. We shall end by the global algorithm.

3.1. Product of a vector by a tensor product of matrices. In order to compute $y = x \bigotimes_{k=1}^{N} A^{(k)}$ over P processors, we are going to use the algorithm proposed in [11]. The data are distributed according to the following scheme:

— Decomposition of P in N integers $d_1,\ d_2,\ \ldots, d_N$ such that d_k divides n_k. Let us notice that this decomposition is not unique, but all the possible decompositions are equivalent from a time complexity point of view. Nevertheless, a good criterium of choosing a decomposition instead of another is that the sum of the terms must be minimum.

— For each $k \in \{1,\ \ldots,\ N\}$, build a partition of $G_k = \{1,\ \ldots,\ n_k\}$ in d_k subsets G_{kl}, $l = 1,\ \ldots,\ d_k$, i. e.,

$$\bigcup_{l=1}^{d_k} G_{kl} = G_k.$$

That partition must be made respecting the load balancing between the P processors.
Let us consider the following indexation scheme.

$$(l_1,\ l_2, \ldots,\ l_{N-1},\ l_N) = l_{[1,N]} \rightleftharpoons (\ldots((l_1)n_2 + l_2)n_3 \ldots) = \sum_{k=1}^{N} l_k M_{k+1}^{N}, \tag{3.1}$$

where M_l^u is given by relation (2.11). For any processor p, considering relation (3.1), we have the following correspondence:

$$p \rightleftharpoons w_{[1,N]}$$

and thus the vector allocation is done the following way:

$$w_{[1,N]} \longleftarrow y_{(l_1,\ l_2,\ldots,\ l_N)},\ \textit{with}\ l_k \in G_{kw_k},\ k = 1,\ \ldots,\ N.$$

The proposed algorithm is recursive with N steps. It is based on the canonical factorization of the tensor product, such that *at each step, only one matrix of the product is used*. Each processor uses its own data for its computations, then sends the results to the processors that will need them in the following step. In the same time, it receives the data it will need for the next step. The communications between processors are expressed using simple primitives:

send: a processor send a message to a single processor.

receive: a processor receives a message from a single processor.

broadcast: a processor send a message to several processors.

These primitives are efficiently implemented over almost all the existing architectures [14]. The described algorithm uses an overlapping of communications and computations, avoiding bufferization. Another advantage

of that algorithm is a message is sent to a processor if and only if it needs it. The number of communications is therefore reduced to the minimum.

The execution time of a parallel program depends essentially on the communications. The transmission time of a message of $\mathcal{M}$ bytes between two processors p_1 and p_2 over a distance $d = dist(p_1, p_2)$ is represented by the linear model [15]:

$$t(d, \mathcal{M}) = \mathcal{M}.t_c(d, \mathcal{M}) + \tau(d, \mathcal{M}),$$

where $t_c(d, \mathcal{M})$ is the transmission time of one byte and $\tau(d, \mathcal{M})$ is the start-up time. It depends on d, but both of the parameter may be function of $\mathcal{M}$ if the computer uses different protocols of communication according message size (e. g. Intel iPSC/860). Finally, the execution time is the product of one communication time (the above linear function) by the number of communications.

The algorithm of a product vector-tensor product of matrices, that we shall refer to as *PARATENS*, executes at most $\Gamma(P)$ communication steps, $\Gamma(P)$ being the number of communication steps necessary to broadcast a message to P processors. This number depends on the topology, for example $\Gamma(P) = log(P)$, for hypercube topology. Let us note that the arguments of *PARATENS* are the vector and the matrices of the tensor product.

3.2. Product of a vector by a tensor sum of matrices. Let us remind that (relation (2.4)):

$$\bigoplus_{k=1}^{N} A^{(k)} = \sum_{k=1}^{N} \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right). \tag{3.2}$$

The obvious way of computing $z = x \bigoplus_{k=1}^{N} A^{(k)}$ is as follows: at each iteration k, all the matrices arguments, except one, are set to identity. Then, PARATENS is used to complete the step. This results in executing PARATENS N times. But, we notice that at any iteration k, the expression

$$V \left(I_{M_1^{k-1}} \otimes A^{(k)} \otimes I_{M_{k+1}^N} \right)$$

is computed, where V is the vector resulting from the iteration $k-1$. Therefore, the result may be obtained by executing an algorithm such that the execution time is equal to that of PARATENS. That algorithm, called $PARATENS^+$, is presented below (Algorithm 2). In this algorithm, $PARATENS_k$ denotes the procedure that carries out the k^{th} iteration of PARATENS.

Input: $x,\ n_k,\ A^{(k)},\ k = 1, \ldots,\ N$
Output: $z = x \bigoplus_{k=1}^{N} A^{(k)}$
  $Y = 0$
  for $k = 1, \ldots, N$ **do**
    $Y \leftarrow Y + PARATENS_k(n_k, A^{(k)},\ Y)$
  end for

Algorithm 2: Parallel algorithm of the product vector-tensor sum of matrices

3.3. Implementation of the global algorithm. Algorithm 3 computes the vector $\Pi(t)$ over P processors. The first step of this algorithm consists in computing the uniformized rate q by following way. By definition,

$$q \geq \max_i \mid q_{ii} \mid,\ i = 1, \ldots, M_1^N,$$

where q_{ii} are the diagonal elements of the descriptor Q (relation (2.3)). Because the square matrix Q is an infinitesimal generator, then we have

$$q_{ii} = \sum_{j=1}^{M} q_{ij},\ j \neq i,$$

Input: $Q^{(k)}$, $E_i^{(k)}$, $\Pi(0)$, t, ε
Output: $\Pi(t)$
  Compute q and N_T * Relation (2.7)*\
  Compute Q/q
  $e_0 \leftarrow 1$; $\tilde{\Pi}^p(0) \leftarrow \Pi_0^p$; $\Pi^p(t) \leftarrow \Pi_0^p$
  for $j = 1, \ldots, N_T$ **do**
    $e_j \leftarrow \frac{qt}{j} e_{j-1}$
    $PARATENS^+(Q^{(1)}, \ldots, Q^{(N)}, \tilde{\Pi}^{(j-1)})$ * $\tilde{\Pi}^{(j-1)} \oplus_{k=1}^N Q^{(k)}$ *\
    for k=1,...,2T **do**
      $PARATENS(E_k^{(1)}, \ldots, E_k^{(N)}, \tilde{\Pi}^{(j-1)})$ * $\tilde{\Pi}^{(j-1)} \otimes_{k=1}^N E_i^{(k)}$*\
    end for * Results in $\tilde{\Pi}^{(j)^p}$ for each processor p *\
    $\Pi^p(t) \leftarrow \Pi^p(t) + e_j \tilde{\Pi}^{(j)^p}$
  end for

Algorithm 3: Parallel algorithm of the computation of $\Pi(t)$ over P processors

where $M = M_1^N$ is the size of the matrix Q. Finally,

$$q \geq \max_i \mid \sum_{j=1}^{M} q_{ij} \mid,\ i = 1, \ldots, M.$$

The rate q is obtained by computation of the vector

$$Y = Q \times \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix},$$

where all the diagonal elements of Q are substituted by zero. Next, we choose q such that

$$q = \max_i \mid Y_i \mid,\ i = 1, \ldots,\ M.$$

The computation of $\Pi(t)$ requires, at each step, one execution of $PARATENS^+$ and $2T$ executions of $PARATENS$. *The time complexity of the global algorithm is minimum due to the fact that all the algorithms are optimum in communications number.*

4. Numerical results.

4.1. The ATM example. In this example, we describe a congestion control of an ATM (Asynchronous Transfer Mode) network. The ATM [16] was conceived to face the transmission of new types of data (voice, video, etc.). It is a specific packet oriented transfer mode based on fixed length cells of 53 bytes. These cells result from the splitting of the input streams. A source connected to the network inserts its cells into free spaces not used by the other sources. High speed connections of this kind may exceed $600 Mbits/s$.

A crucial problem in such networks is the congestion control. Moreover, this problem must be treated with integration of quick adaptation and reaction to high speed connections. This problem is responsible for loss of information (buffers saturation) and transmission time increase. Solving this problem consists in reducing the congestion with a preventive and adaptive method. The classical techniques are not always applicable, on account of the high speed in the ATM networks. It is therefore necessary to establish adapted control mechanisms. The congestion control in ATM network may be executed at different level according to the kind of information carried and the traffic's characteristics. Three levels are possible:

— Admission level
— Burst level
— Cell level.

At admission level, the system determines whether a connection can be progressed or should be rejected based on the resource availability in the network: it is an access control. At burst level, a control mechanism (such as

the leaky bucket) checks, permanently, that the input flow respects the negotiated traffic contract: it is a flow control. At cell level, the control is done using the CLP bit (Cell Loss Priority) contained in the header of each cell. This bit makes the difference between cells according to their priority. In congestion case, low priority cells are destroyed. Only high priority cells are kept in the network.

The Leaky Bucket (LB) [17] is an access control mechanism in the ATM network. This control is performed using tokens. These tokens are given to each cell when it enters the network. This mechanism may be implemented in several ways. We focus on the one called the Virtual Leaky Bucket (VLB) [18]. In this kind of LB, three buffers are needed. The first one welcomes the user's cells, and the others are respectively used for green and red tokens. When a cell arrives in the first buffer B_c, if that one is not full, it is kept there waiting to be served. A service to a cell consists in giving it a green token coming from the buffer B_g. This token represents its permission to access into the network. Otherwise, if B_c is full, the cell may be lost (rejected), even if the network has sufficient resources to accept it, without altering the quality of service. In order to avoid this situation, red tokens are generated by the buffer B_r. A threshold S is fixed. If B_g is empty while there are less than S cells in B_c, those cells should wait for new green tokens to be generated, before they enter the network. On the contrary, if there are more than S cells in B_c when B_g is empty, they should be able to access the network with red tokens if, of course, B_r is not empty. This mechanism is described by figure 4.1.

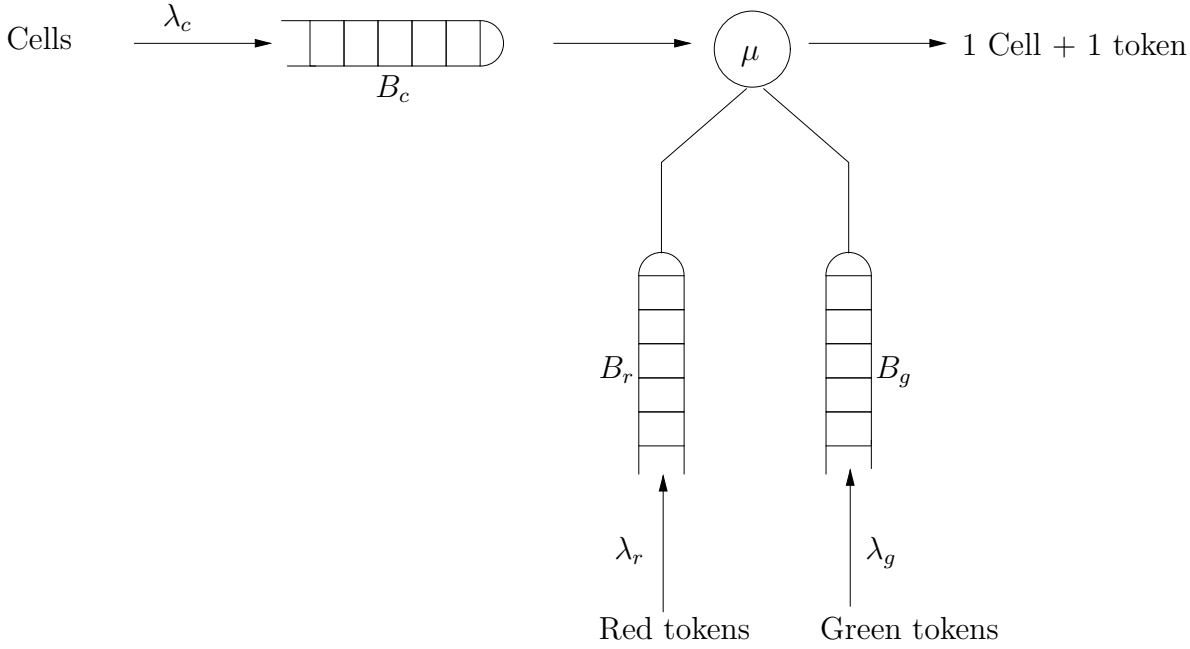


FIG. 4.1. *The Virtual Leaky Bucket*

The ATM mechanism is modeled by a SAN composed with three automata $\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$ and $\mathcal{A}^{(3)}$. These automata represent respectively the content of buffers B_c, B_g and B_r. Each automaton $\mathcal{A}^{(k)}$ is supposed to haves n_k states, $k = 1, 2, 3$. These states are numbered from 0 to $n_k - 1$. The threshold of buffer B_c is set to S. The events that occur in the system are as follows:

- Local events
  - — arrival of a cell with rate λ_c, a local event to $\mathcal{A}^{(1)}$
  - — arrival of a green token with rate λ_g, a local event to $\mathcal{A}^{(2)}$
  - — arrival of a red token with rate λ_r, a local event to $\mathcal{A}^{(3)}$.
- Synchronizing events
  - — s_1: departure of a cell with a green token (rate μ), acting on both $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$
  - — s_2: departure of a cell with a red token (rate μ), acting on both $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(3)}$.

Figure 4.2 shows the behaviour of each automaton for the case where $n_1 = 4$, $n_2 = n_3 = 3$ and $S = 1$.

The transitions in $\mathcal{A}^{(1)}$ are either cells arrivals (with rate λ_c) or cells departures (service). A departure is also synchronization transition because a cell leaves B_c with a token (green if the number of cells in B_c is less

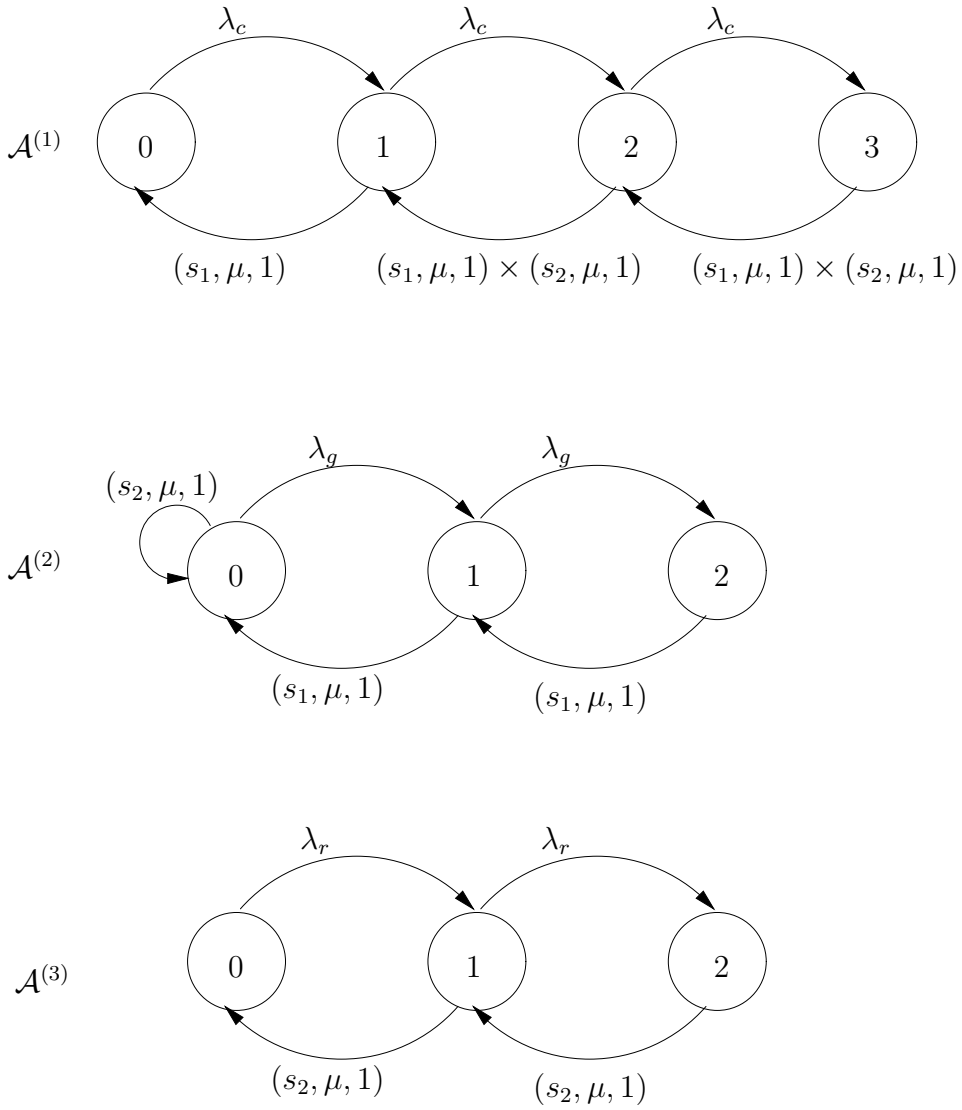


FIG. 4.2. *The automata transitions diagram of the SAN associated to the Virtual Leaky Bucket*

than S and red if not). The service rate is always μ and the occurrence probability (routing probability) is 1, only one choice being possible. The transitions in $\mathcal{A}^{(2)}$ and $\mathcal{A}^{(3)}$ are arrivals (with rates λ_g or λ_r) of green or red tokens or their departures. A token departure means that a cell has been served, therefore synchronization events s_1 and s_2 stand. These transitions have rate μ and also an occurrence probability equals to 1. The fact that a red token is usable only when B_g is empty is modeled by the loop around state 0 of automaton $\mathcal{A}^{(2)}$.

The SAN modeling the VLB is determined by the descriptor Q and the initial distribution $\Pi(0)$ as follows. Let be, for $k = 1, 2, 3$,

$Q^{(k)}$ the infinitesimal generator associated to automaton $\mathcal{A}^{(k)}$,
$E_1^{(k)}$ the positive event matrix of event s_1 over automaton $\mathcal{A}^{(k)}$, and
$E_2^{(k)}$ the positive event matrix of event s_2 over automaton $\mathcal{A}^{(k)}$.

The *descriptor* is given by

$$Q = Q^{(1)} \oplus Q^{(2)} \oplus Q^{(3)} + E_1^{(1)} \otimes E_1^{(2)} \otimes E_1^{(3)} + E_2^{(1)} \otimes E_2^{(2)} \otimes E_2^{(3)}.$$

If $\Pi^{(k)}(0)$, $k = 1,\ 2,\ 3$, denotes the initial distribution of the k^{th} automaton $\mathcal{A}^{(k)}$, we have $\Pi_1^{(k)}(0) = 1$ and for $i \geq 2$, $\Pi_i^{(k)}(0) = 0$. The global initial distribution (of the SAN) is $\Pi(0)$ such that $\Pi(0) = \otimes_{k=1}^{3} \Pi^{(k)}(0)$.

4.2. Performance analysis. In order to compute the vector $\Pi(t)$ for our SAN model, we consider $n_1 = 256$, $n_2 = 128$ and $n_3 = 32$. This situation means that buffers B_c, B_g and B_r have a limited capacity of 255, 127 et 31 respectively. The threshold S is fixed to 200. Thus, the descriptor Q has an order $M = 2^{20}$ and **the system has 1.048.576 states**. It is important to note that only the matrices $Q^{(k)}$ and $E_i^{(k)}$, $k = 1, 2, 3$ and $i = 1, 2$, are stored by using a compact storage scheme. The rates values are such that $\lambda_b = \lambda_c = \lambda = 0.5$ and $\mu = 1$. For evaluating the performance of our global algorithm, we execute it on a CRAY T3E, a distributed memory parallel machine. It possesses up to 256 processing elements, each running at 300 MHZ. The computations of the program (written in Fortran) are done in numerical arithmetic double precision.

We first focus on the CPU time of our algorithm as function of the mission time t and the number of processors P. We consider $t = 10^i$, $i = 0, \ldots, 5$ and $P = 32, 64, 128$. Next, we compute the speedup $\mathcal{S}_P$ and the efficiency $\mathcal{E}_P$ as function of P. For the SU method, the value of ε is fixed to 10^{-10} (cf. relation (2.7)).

TABLE 4.1
CPU time (s) for computing $\Pi(t)$ as function of t and P

t	1	10	10^2	10^3	10^4	10^5
P=32	61.72	222.17	1745.45	14599.34	137696.21 (e)	1350907.9 (e)
P=64	30.55	137.76	912	7628.15	71945 (e)	705842.6 (e)
P=128	1.6	7.96	48.16	427.93	3791.89	37201.16

Table 4.1 includes the CPU time values for computing $\Pi(t)$ as function of t and P. In this table, *the notation (e) means that corresponding CPU time values are estimated taking advantage of the SU method for estimating a priori the time complexity.* A lecture of this table shows that even when $t = 10^5$, the vector $\Pi(t)$ can be evaluated with 128 processors in about 10 CPU hours. This table also shows the feasibility limits of some problems as function of the state space size M, mission time t, and number of processors P. Speedup

TABLE 4.2
Speedup and efficiency as function of P

P	32	64	128
$\mathcal{S}_P$	29	54	100
$\mathcal{E}_P$	0.90	0.84	0.78

and efficiency, as function of P, are given in Table 4.2. The values of $\mathcal{S}_P$ increases with P; the CPU times is then inversely proportional to P. Table 4.2 shows that the value of $\mathcal{E}_P$ remains in average greater than 80%, meaning a good use of processors and a low communication time.

It is important to note that the given numerical results depend on the SAN model. The speedup $\mathcal{S}_P$ decreases when N increases [19]. It is then possible to aggregate some automata [20] in order to obtain the same value of N. The mean number α of non-zero terms in rows or columns of matrices used in the tensor sums and products increases without modifying significantly the efficiency $\mathcal{E}_P$. If we consider complex SANs for which N is large (N increases), it is difficult to predict the expected speedup starting from the given application. More complex applications constitute the goal of our future work.

5. Conclusion. In this study, we addressed the problem of transient solution of SAN. This modelisation methodology allows to treat complex parallel systems by avoiding the built of the entire infinitesimal generator. In the computation of the transient state probability vector, we faced the problem of computation time, especially, for large mission time values. We first adapted the SU method to compute this vector and developed an new sequential algorithm which computes a product of a vector by a tensor sum of matrices. Next, we implemented a parallel algorithm of the SU method in order to deal with the increasing of the time complexity with the mission time. The parallel version of this implementation uses an efficient algorithm computing a product of a vector by a tensor product of matrices and a parallel version of the presented algorithm. The interest of used algorithms is the minimization of communication time between processors. We presented numerical results of a SAN modeling an ATM network with 1048000 states and large mission time values. Some CPU time values are given as function of mission time and number of processors. We also given speedup and efficiency as function of number of processors. Even a decreasing of the efficiency when the number of processors increases, the value of this efficiency remains, in average, greater than 80%.

REFERENCES

[1] B. Plateau, *On the Stochastic Structure of Parallelism and Synchronisation Models for Distributed Algorithms,* Performance Evaluation, 13(5):142–154, 1985.

[2] W. J. Stewart, K. Atif, and B. Plateau, *The numerical solution of stochastic automata networks,* European Journal of Operation Research, 86:503–525, 1995.

[3] B. Plateau, *On the Stochastic Structure of Parallelism and Synchronisation Model for Distributed Systems,* ACM SIGMETRICS conference On Measurement and Modeling of Computer Systems, pages 147–153, May 1985.

[4] B. Plateau and K. Atif, *Stochastic Automata Networks for Modelling Parallel Systems,* IEEE Transactions On Software Engineering, 17(10):1093–1108, 1991.

[5] M. S. Bebbington, *Parallel implementation of an aggregation/disaggregation method for evaluating quasi-stationary behaviour in continuous-time Markov chains,* Parallel Computing, 23:1545–1559, 1997.

[6] J. K. Muppala M. Malhotra, K. S. Trivedi, *Stiffness-tolerant methods for transient analysis of stiff Markov chains,* Microelectronic Reliability, 34(11):1825–1841, 1994.

[7] K. S. Trivedi and A. L. Riebman, *Numerical Transient Analysis of Markov Models,* Computer and Operations Research, 15(1):19–36, 1988.

[8] H. Abdallah and R. Marie, *The Uniformized Power Method for Transient Solutions of Markov Processes,* Computers and Operations Research, 20(5):515–526, April 1993.

[9] C. Lindemann, M. Malhotra and K. S. Trivedi, *Numerical Methods for Reliability Evaluation of Markov Closed Fault-Tolerant Systems,* IEEE Transactions on Reliability, 44(4):694–704, 1995.

[10] H. Abdallah and M. Hamza, *Sensitivity analysis of instantaneous transient measures of highly reliable systems* 11^{th} European Simulation Symposium (ESS'99), Erlangen-Nuremberg, Germany, october 26-28, pages 652–656, 1999.

[11] C. Tadonki and B. Philippe, *Parallel Multiplication of a Vector by a Kronecker Product of Matrices (part I),* Parallel and Distributed Computing Practices, 2(4):413–427, December 1999.

[12] H. Abdallah and M. Hamza, *Sensitivity analysis of the expected accumulated reward using uniformization and IRK3 methods,* Second Conference on Numerical Analysis and Applications (NAA'2000), Rousse, Bulgaria, June 11-15, 2000.

[13] M. Davio, *Kronecker Products and Shuffle Algebra* IEEE Transactions on Computers, 30:116–125, 1981.

[14] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computing.* Prentice-Hall, Englewood Cliffs, N. J., 1988.

[15] S. M. Mller A. Bingert, A. Formella and W. J. Paul, *Isolating the Reasons for the Performance of Parallel Machines on Numerical Programs,* In International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performace Prediction, pages 34–64, Austin, Texas, USA, 1993.

[16] B. Weiss *ATM.* Hermes, Paris, 1995.

[17] I. Cidon and I. S. Gopal, *An approach to Integrated High-Speed Private Network,* International Journal on Digital and Analogical Systems, 1:77–86, 1988.

[18] M. Hirano and N. Watanabe, *Characteristics of a cell multiplexer for bursty ATM traffic,* IEEE ICC'89, pages 1321–1325, 1989.

[19] C. Tadonki and B. Philippe, *Parallel Multiplication of a Vector by a Kronecker Product of Matrices (part II),* Parallel and Distributed Computing Practices, 3(3), September 2000.

[20] O. Gusak, T. Dayar, and J. M. Fourneau, *Lumpable continuous-time stochastic automata networks,* International Conference on Mathematical Modeling and Scientific Computing, Middle East Technical University and Selcuk University, Ankara and Konya, Turkey, April 2001.

Edited by: Roman Trobec
Received: January 15th, 2002
Accepted: June 26th, 2002

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 65–86. http://www.scpe.org
ISSN 1895-1767

A SIMD ENVIRONMENT FOR GENETIC ALGORITHMS WITH INTERCONNECTED SUBPOPULATIONS

DEVARAYA PRABHU* AND BILL P. BUCKLES† AND FREDERICK E. PETRY†

Abstract. The algorithmic form of GAs conforms well to SIMD computing environments with relatively minor adjustments to the operators. In this paper we consider in detail a GA implementation on a MasPar machine. The question of the degree to which control parameters affecting intercommunication impact performance is addressed using ANOVA methods. The purpose is to supplant anecdotal experience with statistical evidence. A set of control parameters—topology, migration operator, migration radius, and migration probability—were chosen together with four representative levels of each. Metrics for three response variables—efficiency, diversity, and schema propagation—were developed that allowed insight into the behavior under the various parametric conditions. These were incorporated into three $4 \times 4 \times 4 \times 4$ randomized factorial experiment designs. Among other things, it was determined that the interconnection topology is not in itself a significant factor but the extent of connectivity and frequency of communication are. An important outcome of this study is that, while the individual factors are significant, the factors do not interact in unexpected ways.

Key words. Genetic algorithms, massively parallel computation, communicating subpopulations, migration, Royal Road functions, experiment design.

1. Introduction. Because of the complexity of large optimization problems, the value of parallel realizations of optimization algorithms has become quite evident [10, 11]. In particular, in this paper we present a study of an approach to parallel genetic algorithms.

Amenability to parallelization has always been viewed as a major strength of Genetic Algorithms (GAs). GAs operate on populations of individuals. When the population is distributed into (perhaps overlapping) subpopulations, the introduction of new operators is possible as well as altering the semantics of existing ones. We consider a GA to be parallel if its operators reflect the new interactions that are enabled by the presence of subpopulations.

Furthermore, we are interested in the case for which there are thousands, not dozens, of subpopulations. When there are a massive number of subpopulations, there are larger number of local behaviors competing for the opportunity to expand and their expansion is more acutely observable. The operators that affect subpopulation interactions are defined in part by parameters. In some cases, the number of alternative values for these parameters increase as the number of subpopulations increase. Thus choices made by algorithm designers must be more informed. These two factors, measurability of effects and the need for insights, make massively parallel GAs a useful domain of investigation.

One aspect of this paper is the methodology that is applied, namely, experimental design methods [21, 29]. It has the purpose of establishing the quantitative significance of various factors, singly and in combination, that affect performance. This is in contrast to the "typical" GA experiment that quantifies performance prediction but only qualifies the factors affecting the prediction. The method used here, we believe, should be used more often because, beyond quantification, it is able to elicit evidence of the effects of factor interaction. That is, how the effect of factors change when used in combination with others.

The SIMD parallel genetic algorithm is analyzed. Its structure tends to be uniform across many applications and platforms thus identification of factors to investigate requires less subjective judgment. Many SIMD computers are in use and the architecture itself is now re-emerging in the context of special-purpose VLSI plug-ins.

Using experiment design methods tends to require many individual trials over a small number of factors. In this case, more than 2,500 individual instances of the algorithm comprised the single experiment reported. The outcome showed all factors investigated – network topology, migration policy, migration probability, and migration radius—were significant at confidence levels ranging from 90% to 99%. A key question was: Do these factors interact with each other in a simple additive fashion or in a more complex manner? It was found that the interactive effects were sufficiently close to additive that no special precautions by the practitioner are necessary.

*i2 Technologies, 1603 LBJ Freeway, Suite 780, Dallas TX 75234
†The Center for Intelligent and Knowledge-based Systems, Department of Electrical Engineering & Computer Science, Tulane University, New Orleans, LA 70118. Dev_Prabhu@i2.com {buckles, petry}@eecs.tulane.edu

2. Background.

2.1. Sequential GAs. Genetic algorithms originated from the studies of cellular automata, conducted by John Holland [22]. A GA is a search procedure modeled on the mechanics of natural selection rather than a simulated reasoning process. Domain knowledge is embedded in the abstract representation of a candidate solution termed an organism. Organisms are grouped into sets called populations. Successive populations are called generations. A generation GA creates an initial generation, *G(0)*, and for each generation, *G(t)*, generates a new one, *G(t + 1)*. An abstract view of the algorithm is

generate initial population, $G(0)$
evaluate $G(0)$
$t := 0$
repeat
  $t := t + 1$
  generate $G(t)$ **using** $G(t-1)$
  evaluate $G(t)$
until solution is found.

The operation "evaluate *G(t)*" refers to the assignment of a figure of merit to each of the population's organisms. In simple GAs, an alternative exists to replacing an entire population at once; that alternative is to replace one organism in the population whenever a new organism is created. This variant is known as a steady state GA.

In most applications, an organism consists of a single chromosome. A chromosome of length n is a vector of the form $\langle x_1, x_2, \ldots, x_n \rangle$. where each xi is an allele, or gene. The domain of values from which xi is chosen is called the alphabet of the problem. Frequently, the alphabet used consists of the binary digits {0, 1}. We can view a specific chromosome as representative of many patterns. Using # as "don't care," an example pattern or schema over the binary alphabet is $\langle \#\#110\# \rangle$. The chromosomes $\langle 111100 \rangle$ and $\langle 101100 \rangle$ are specific instances of this and other schemata; for example, $\langle \#\#100 \rangle$. The order of a schema is the number of non-# symbols it contains. Its length is the distance from the first to the last non-# position. Thus, the length of $\langle \#1\#0\#1 \rangle$ is four, and its order is three.

GAs differ from traditional search techniques in several ways

- First, GAs optimize the trade-off between exploring new points in the search space and exploiting the information discovered thus far. This was proven using an analogy with the k-armed bandit (an extension of the one-armed bandit) problem.
- Second, GAs have the property of implicit parallelism. Implicit parallelism means that the GA's effect is equivalent to an extensive search of hyperplanes of the given space, without directly testing all hyperplane values. Each schema denotes a hyperplane.
- Third, GAs are randomized algorithms, in that they use operators whose results are governed by probability. The results for such operations are based on the value of a random number.
- Fourth, GAs operate on several solutions simultaneously, gathering information from current search points to direct subsequent search. Their ability to maintain multiple solutions concurrently makes GAs less susceptible to the problems of local maxima and noise.

As noted, GAs are randomized—but not random—search algorithms. Each organism represents a point (that is, an intersection of hyperplanes) in the search space. Randomization must balance two competing concerns, exploration and exploitation. A solution cannot be tested unless it appears as an organism. Therefore, a reasonable number of solutions must be explored. On the other hand, unlimited exploration would not be efficient search. The strength of highly fit organisms must be exploited and allowed to propagate in the population. Yet, giving too much precedence to such organisms results in premature termination at a local optimum.

We can compare GA recombination operators to controlled breeding among, say, thoroughbred horses. The objective is to combine highly fit organisms to produce a still more fit individual. Both the selection of "parents" and the steps within the recombination operators are randomized. Parent selection dynamics are based on an application-dependent measure of an organism known as the fitness function, f_i (f_i is a figure of merit, computed using any domain knowledge that applies). In principle, this is the only point in the algorithm at which domain knowledge is necessary. Organisms are chosen using the fitness value as a guide; organisms having higher fitness values are chosen more often. Selecting organisms based on fitness value is a major factor

in the strength of GAs. The greater the fitness value of an organism, the more likely that the organism will be selected for recombination.

There are two popular approaches for implementing selection. The first, roulette selection, assigns a probability to each organism, i, computed as the proportion

$$F_i = f_i / \sum_j f_j$$

A parent is then randomly selected, based on this probability. A second method, deterministic sampling, assigns to each organism, i, a value

$$C_i = RND(F_i \times n) + 1$$

where n organisms reside in the population (RND means round to integer). The selection operator then assures that each organism participates as a parent exactly C_i times.

Parents participate in the later recombination operations. Alleles from the parents are mixed via an operator called a crossover rule, of which many exist. Simple one-point crossover of chromosomes from two parents at a random point, j, is illustrated by

$$\langle x_i\ x_2\ \cdots x_j\ x_{j+i}\ x_{j+2} \cdots x_n \rangle \tag{2.1}$$
$$+ \tag{2.2}$$
$$\langle y_1\ y_2\ \cdots y_j\ y_{j+1}\ y_{j+2} \cdots y_n \rangle \tag{2.3}$$
$$= \tag{2.4}$$
$$\langle x_1\ x_2\ \cdots y_j\ y_{j+1}\ y_{j+2} \cdots y_n \rangle \tag{2.5}$$

where the result is a chromosome of the offspring and is placed in the next generation.

A mutation is the random change of an allele from one alphabet value to another. For a problem over the binary alphabet, the original allele is exchanged for its complement. The mutation operator offers the opportunity for new genetic material to be introduced into a population. From the theoretical perspective, it assures that—given any population—the entire search space is connected. The new genetic material does not originate from the parents and is not introduced into the child by crossover. Rather, it occurs after crossover a small percentage of the time.

Several stopping criteria exist for the algorithm. The algorithm may be halted when all organisms in a generation are identical, when $f_i = f_j$ for all i and j, or when $|f_i - f_j| < TOL$ for some small value TOL and all i and j. An alternative criterion would halt after a fixed number of evaluations and take the best solution found.

2.2. Parallel GAs. Parallel versions of GAs have, more often than not, different semantics from the canonical serial algorithm [6, 45]. In other words, the behavior of the algorithm in terms of the computational actions taken as well as the the results generated for any given set of inputs can be expected to differ for the two versions. In contrast, for most other algorithms, parallelization usually implies a parallel version of the algorithm which maintains the semantics of the serial version. Not surprisingly, early parallel GA implementations (e.g. [34]) focused mainly on achieving computational speedup. There have been a variety of applications of parallel GAs including topics such as fuzzy logic controller design [1], VLSI routing [25], financial market computations [32], SAT methods [15], land use modeling [49], and many others [2].

The changed semantics of a parallel GA is primarily the result of the way in which the population members are distributed among subpopulations as well as the nature and the extent of interactions among such subpopulations. A serial GA which uses a single centralized population and global access to all its members (as evidenced by panmictic selection) simply constitutes one end of a spectrum of possibilities [8]. At the other end of that spectrum lies a fine-grained parallel GA with completely decentralized population (resulting in singleton subpopulations) and a local neighborhood-based selection. Most other parallel GAs can be seen as being somewhere in between these two cases. They employ a varying extent of decentralization of the global population and access among the resulting subpopulations during selection. The subpopulations resulting from the distribution of the global population members can be called *demes*, a very evocative term from the field of population biology. Parallel GA literature also refers to them as *islands* [48, 27].

The interaction among the demes can take two forms: *migration* and *overlapping selection* [7]. Migration refers to the process of periodic import of population members from neighboring demes into the local subpopulation. In the artificial setting of parallel GAs, this involves a sequence of export-import-replace actions. First, every deme must decide on candidates for export or emigration from the local subpopulation, which in turn can be imported by the neighboring demes. Next, a deme can decide to import or immigrate some of such candidate emigrants available in the neighboring demes. The frequency of import is determined by the *migration probability*. Since most parallel GA models enforce fixed size populations, a necessary third step would be to incorporate the newly imported members into the local subpopulation by replacing local members.

A different kind of interaction among the demes is based on the notion of selection using effectively overlapping subpopulations. In this scheme, during the selection and reproduction stage of GAs, some of the neighboring subpopulations are also taken into consideration along with the local subpopulation. Thus, using k neighboring demes and a subpopulation size of n at each deme, selection and reproduction step results in sampling n members from the pool of size $(k+1) \times n$. Depending on such overlap, the effective subpopulations employed over all the demes constitute either a partition or a covering set of the global population. Obviously, a meaningful migration operation in such a model can be designed only by considering the demes which are not in the same selection pool.

2.3. Related Research. Here we focus on parallel GAs that support the deme concept. It was recognized early that a variety of choices existed with respect to the spatial distribution of population members [41, 5]. At one end of the spectrum are so-called coarse-grained parallel GAs [3]. In this case, the global population is divided into a few distinct non-overlapping subpopulations. Communication among them is accomplished by periodic migration [12, 13]. Selection within a subpopulation is an issue but there are no inter-deme selection issues. If a parallel computer is applied to coarse-grained GA simulation, a MIMD architecture is most appropriate [43].

Rigorous experimentation with coarse-grained GAs was first performed by Tanese [44]. She examined the frequency and subpopulation fraction for migration. It was discovered that moderate values of each are the most effective. She also noted something else that was peculiar: Even in the absence of any inter-deme communication, the parallel GA with small subpopulations performed better than the serial GA. Forrest *et al.,* in follow-up work [17], determined that this was an artifact of the problem to which the GA was applied (a subset of Walsh functions). The coarse-grained approach allows considerable freedom such as varying control parameters, e.g., crossover rate, between subpopulations [44], varying the representation from one subpopulation to the next [37], and even dynamically varying the subpopulation sizes and number of subpopulations [46, 20].

In contrast, a fine-grained parallel GA is, strictly speaking, one in which the number of demes is the same as the population size. That is, each deme consists of one individual. A classic example is one reported by Manderick *et al.* [26]. In this class of GAs the principal issue is selection policy (radius, for example) [14]. A SIMD architecture is the more appropriate parallel computer for fine-grained GA simulations. Some other terms that have been used for fine-grained GAs include cellular GAs [47] and diffusion GAs [33]. The fine-grained approach has also been applied to evolution strategies to study different selection schemes [18]. Cellular programming of cellular automata is an approach that is also related to fine-grained parallel GAs [40, 9].

Mühlenbein initially developed a fine-grained GA [30] on a double-ring grid that incorporated hill-climbing. Later, the system was extended [31] in a manner that incorporated non-singleton subpopulations with both inter-deme selection and migration issues and so was not strictly fine-grained. The algorithm we have used is also not strictly fine-grained, but tends in that direction [35]. Small subpopulations were isolated with respect to selection but connected with respect to migration. This was in accordance with the controlled variables we wished to study. Otherwise, the algorithm as illustrated in Figure 2.1 has all the recognizable elements of a serial GA.

3. Logical Architecture. This experiment focuses primarily on the interactions of subpopulations. Such interactions are constrained by the connectivity among the subpopulations and the consequent neighborhood structure within the logical architecture of the parallel GA. The physical architecture does not prohibit any specific logical architecture. Suppose that n nodes are given. Any graph that weakly connects them induces a *topology*. The topologies considered in this study further assume that the graph is not directed. Specifically, the four topologies considered are shown in Figure 3.1. Choice of a topology results in a *neighborhood* for each node. A neighborhood can be compact or not. A compact neighborhood of size r consists of a reference node together with all other subpopulations reachable by traversing at most r links of the topology. A noncompact

```
Set the number of subpopulations;
Set the topology and migration radius;
For all nodes synchronously Do:
{
    Randomly generate initial subpopulations;
    Evaluate all members of initial subpopulations;
    While termination conditions not met, Do:
    {
        /* Migration stage */
        Export:  for every node copy a candidate to export;
        Choose a subset of nodes based on migration neighborhood;
        For these nodes synchronously Do:
        {
            Import:  Pick zero or one from the import candidates based on
                         migration probability;
            Replace: Replace a local member by candidate chosen;
        }
        endFor

        /* Reproduction stage */
        Selection:  For each node, assign fitness-based target
                        sampling rates; /* ideal value for re-use as parent */
        Sampling:   Sample members based on target sampling rates;

        /* Transformation stage */
        Crossover:  Stochastically recombine pairs generating offspring pair;
        Mutation:   Stochastically make small changes in offspings;

        /* Evaluation and iteration stage */
        Evaluate each new individual;
        Replace parent subpopulation by offsprings; /* generational GA */
    }
    endWhile
}
endFor
```

FIG. 2.1. *The parallel genetic algorithm*

neighborhood consists of any other strongly connected subgraph of the topology. Figure 3.2 shows an example of compact neighborhoods for $r = 1$ and $r = 2$ for a node in a 4-neighbor torus topology.

Selection using overlapping subpopulations—a form of subpopulation interaction—is based on the notion of compact neighborhoods. We define the *selection neighborhood* of size r_s, termed *selection radius*, for a node to be the corresponding compact neighborhood of that node. Let K_S denote the set of subpopulations in the neighborhood. By extending the chances of reproduction in the local node to members of all the subpopulations in the selection neighborhood, overlapping subpopulations can be effectively simulated based on the selection neighborhood. We consider only the cases for which $r_s = 0$, that is, the selection neighborhood is the local subpopulation.

Subpopulation interaction by migration provides access to members of subpopulations that lie outside the selection neighborhood. Given a selection radius of r_s, we define the *migration neighborhood* of size r_m to be the noncompact neighborhood resulting from deleting the selection neighborhood from the compact neighborhood of size $(r_s + r_m)$ where r_m is termed *migration radius*. In other words, if we denote K_{S+M} to be the set of

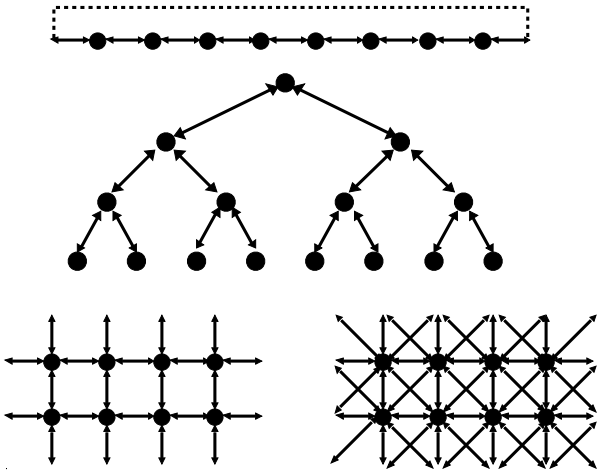


FIG. 3.1. *Subpopulations in ring, tree, 4- and 8-neighbor torus topologies. Open-ended arrows denote wrap-around connectivity.*

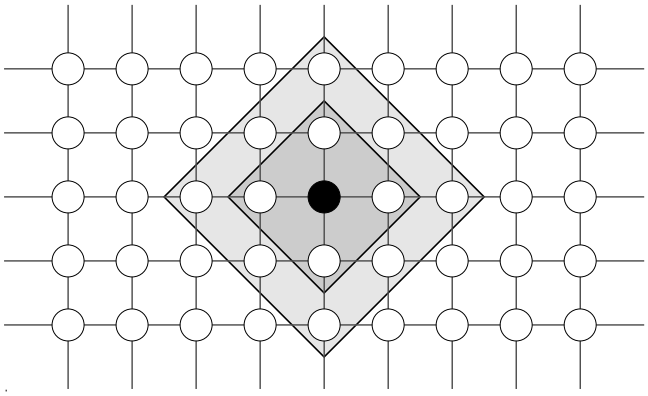


FIG. 3.2. *Compact neighborhoods for a node of a four-torus topology. The nodes in the inner shaded area are included for* $r = 1$. *The nodes in both the shaded areas are included for* $r = 2$.

subpopulations for the size $(r_s + r_m)$, then the set of subpopulations available for the migration operation is defined by $(K_{S+M} - K_S)$. Figure 3.3 illustrates a migration neighborhood for $r_s = 1$ and $r_m = 1$. For a zero-valued selection radius, the migration neighborhood reduces to the compact neighborhood of size r_m, sans the reference node.

4. Experimental Design. Many empirical studies reported in the computer literature compare one method with another or the effect of one factor. This is in contrast to the natural sciences in which it is common to simultaneously study the effects of a number of factors and their interactions. The study here is closer in spirit to the principles of statistical experiment design as first used in agricultural, and then manufacturing, settings. At least one previous study [39] of GAs has adopted a similar strategy.

Specifically, we employed an $(m_1 \times m_2 \times m_3 \times m_4)-$factorial design with complete randomization of trials. That is to say, there were four *factors* (independent variables) with m_1, m_2, m_3, and m_4 levels, respectively. A *level* is a quantitative or qualitative value at which a factor is tested. An *experiment* consists of a set of trials. A *trial* is a test involving each factor set at one of its levels. (In this experiment, a trial consists of one complete run.) Each trial corresponds to a single measurement of a *response variable* (dependent variable). Trials at the same factor levels are repeated several times with different initial conditions in order to obtain meaningful statistical properties.

Trials in an experiment must be conducted using a specific problem. An appropriate problem is one that is both generic and which provides insight into the behavior of the algorithm. Holland's revised version of Royal Road functions meets these criteria [24]. The Royal Road function chosen is particularly large and difficult to optimize. It is sufficiently difficult to exhibit meaningfully different behavior under different experimental

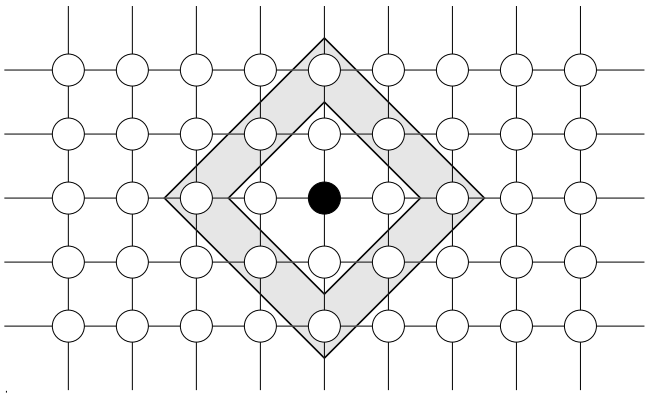


FIG. 3.3. *Migration neighborhood for a node in four-torus topology, given* $r_s = 1$. *The nodes in the shaded area are included by a migration radius of* $r_m = 1$.

conditions as is needed in studies such as this. The Royal Road functions were designed to reward the discovery of lowest level building blocks of the optimal solution as well as the successful recombination of such blocks into higher level blocks in a hierarchical fashion. Royal Road functions have been widely used in a number of areas, for example, crossover variations [42], dual GAs [50] and applications such as learning of neural networks [23].

Here we briefly describe the function adopting the symbols and terminology used by Holland [28] and Jones [24]. For this problem, populations are formed from individuals over the binary alphabet. The structure of the solutions in the population is uniformly specified by the triple $\langle b, g, k\rangle$. Each genome is composed of 2^k basic blocks of length $(b + g)$. Thus, each individual consisted of 240 genes using the parameter values shown in Table 4.1. Within each block, allele values at pre-specified b positions (block bits) are relevant for fitness computation and the allele values of the remaining g positions (gap bits) do not affect the fitness. Such basic blocks are called level-0 blocks. A level-0 block is considered complete when all of its block bits take the value 1. Even-odd pairs of level-0 blocks comprise level-1 blocks. Level-i blocks, for $1 \leq i \leq k$, are comprised of even-odd pairs of level-(i-1) blocks. Further, any level-i block is considered complete if the two adjacent blocks forming its constituent level-(i-1) even-odd pair are both complete.

The computation of the Royal Road function used itself can be described in two parts, typically referred to as the PART and the BONUS computations. PART computes a value with respect to level-0 blocks using two parameters, m^* and v. Let j stand for the number of 1s present in the b block bits for a given level-0 block in the genome. Now, the PART contribution for that block is given by $(j \times v)$, if $j \leq m^*$, and by $((m^* - j) \times -v)$, if $m^* < j < b$. The contribution is zero (0) if $j = b$. The full PART value is computed by summation over all the level-0 blocks. The intentional penalties for near-optimal blocks at the lowest level make this problem harder for hill-climbing algorithms.

The BONUS computation rewards the presence of complete blocks at each level i, $0 \leq i \leq k$, and is specified by the parameters u^* and u. Recall that only the presence of a corresponding even-odd pair of complete blocks of level-(i-1) can generate a complete block at level-i. Let j stand for the number complete blocks present in the genome for a given level-i. The BONUS contribution for level-i is zero, if $j = 0$ and is given by $(u^* + (j - 1) \times u)$ for $j > 0$. The complete BONUS value for the genome is computed by summation over all the levels. The fitness function itself is again a sum of PART and BONUS values. The parameter values used in this study are shown in Table 4.1 and they are similar to the values suggested by Holland.

4.1. Factors. The four factors studied are given in Table 4.2. Each was tested at four levels, leading to a $4 \times 4 \times 4 \times 4$ factorial experiment. Two factors—migration radius and migration probability—are quantitative. Two—topology and migration operator—are qualitative. Thus the experiment consisted of 256 combinations of different levels of factors, or "cells". Each such cell was sampled ten times resulting in a total of 2560 runs of the parallel GA using the same set of ten random seeds for each cell. The sequence of runs was determined in a fully randomized fashion. The parameters held constant over the experiment are listed in Table 4.3. The values in Table 4.3 were chosen following an extensive study of the SIMD algorithm literature. They appear to be robust and generally accepted in applications. Each subpopulation exported one individual (by copy) during each generation. When a subpopulation chose to import, it imported one individual from the set of candidates

TABLE 4.1
Parameters for the Royal Road function.

Parameters	Value
k	4
b	8
g	7
m^*	4
v	0.02
u^*	1.0
u	0.3

TABLE 4.2
Experimental Factors

Factor	Symbol	Levels
Topology	G	{ G_0 = Ring, G_1 = Tree, G_2 = 4-Torus, G_3 = 8-Torus }
Migration Radius	R	{ $R\_0 = 1$, $R\_1 = 3$, $R\_2 = 5$, $R\_3 = 7$ }
Migration Operator	O	{ O_0 = Import-random & Replace-random, O_1 = Import-random & Replace-worst, O_2 = Import-best & Replace-random, O_3 = Import-best & Replace-worst }
Migration Probability	P	{ $P\_0 = 0.0$, $P\_1 = 0.33$, $P\_2 = 0.66$, $P\_3 = 1.0$ }

available. The experiments were performed on the MasPar MP-216 at NASA/GSFC using MPGA [36], a massively parallel GA package developed by the authors.

4.2. Response Variables. The effect of the factors was tested using response variables corresponding to population diversity, schemata propagation, and search efficiency. More than one method was employed [35], but here we restrict the discussion to one representative method for each.

An objective in a GA, parallel or serial, is to obtain a balance between population diversity and the speed of convergence, *i. e.,* exploration and exploitation. Therefore studying the effect of the factors on the population diversity is important. To measure diversity we use a metric similar in spirit to the one described by Collins [13, pages 118–120].

$$\delta = \frac{1}{l} \times \sum_{i=1}^{l} \left(1 - 2 \times \left| 0.5 - \frac{\sum_{j=1}^{m} \sum_{k=1}^{n} \mathrm{Bit}(i,k,j)}{m \times n} \right| \right) \tag{4.1}$$

where, l is the length of the chromosome in bits, m is the number of subpopulations, and n is the number of chromosomes in each subpopulation. $\mathrm{Bit}(i,k,j)$ denotes the value of the ith bit of the kth member in the jth subpopulation. δ is in the range $[0,1]$ where zero (0) is the response for a global population consisting of one individual replicated and one (1) is the response when the global population is maximally diverse, *i. e.,* at each position all alleles are uniformly distributed.

Schemata propagation is used here to mean the extent to which a fit schema is present among all the subpopulations. A priori choice of a single schema to be monitored is not practical because there are a number of search paths from different schemata that lead to an optimum. Taking advantage of the problem symmetry, eight complementary schemata were chosen for monitoring. These schemata are shown in Figure 4.1. At selected generations the subpopulations were examined for presence of the schemata. In the results of Section 5 the final measurement is used.

The response value was not simply the fraction of subpopulations in which a schema was found. It was computed as the number of subpopulations containing instance(s) of the schema less the subset of those demes

TABLE 4.3
GA parameters used in the experiment.

Parameters	**Value**
Number of subpopulations	16,384
Subpopulation size	10
Number of maximum generations	100
Export operator	Export-current-best
Selection radius	0
Selection operator	Ranking
Elitism	No
Sampling operator	Stochastic universal sampling
Mutation operator	Bit-mutation
Mutation probability	0.005 per bit
Crossover operator	Two-point crossover
Crossover probability	0.7

having no neighbors (with respect to the topology) also containing instance(s). That is, only those subpopulations in which the schema could plausibly have occurred as a result of migration are counted. The maximum for the eight candidate schemata was chosen as the basis for the response. This accounts for the various search paths the GA might traverse. The response variable was normalized to be the fraction of the number of applicable subpopulations meeting the propagation criteria.

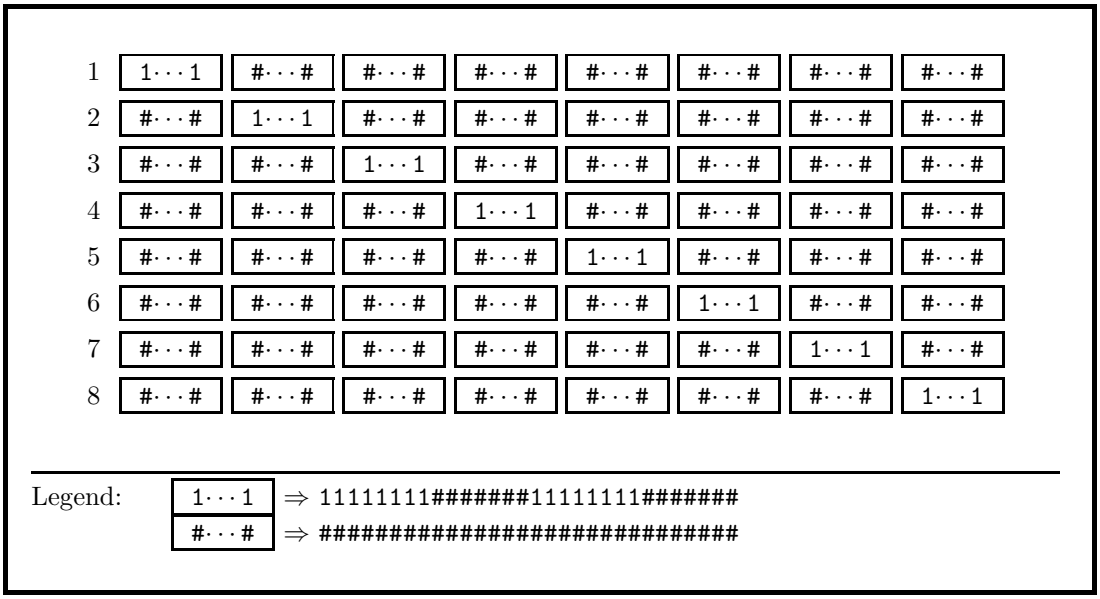


FIG. 4.1. *The list of schemata monitored.*

Finally, search efficiency is measured in terms of the time taken to reach the optimum. In this case, the number of generations taken to find the globally optimal solution is recorded. If a particular trial failed to locate the optimum, the preset value of maximum number of generations was used. Typically, the value for efficiency tends to be domain-specific. However, it is hoped that the premise underlying the design of the Royal Road functions—*building-block hypothesis*—renders the results a significance sufficiently independent of specific applications.

5. Results and Analysis. An important aspect of this paper is the experimentation and analysis of variance methods employed. It is more common to see less formal experiments having a different objective—performance prediction. The formal experimental design approach used here discovers which factors affect

performance. A factor may affect performance by itself, in combination with other factors, or both. The most important result here, given that each factor investigated affected performance as might be expected, is that there were no unusual effects that occurred when factors were used in combination. That is, the effects in combination are a linear combination of the effects of each single factor.

5.1. ANOVA. An illustrative ANOVA is shown in Table 5.1. A complete table for each response variable is in the appendix. The source column lists the factor (e.g., R or migration radius) or factor combination (e.g., $O \times P$ or migration operator with migration probability) that was tested for its effects. The last column (Computed F) is a statistic that is compared to standard tables to extract the degree of confidence that the source factor or source combination has a significant influence on the response. Further description of the ANOVA tables and the columns can be found in the appendix.

TABLE 5.1
Abbreviated ANOVA for global allelic diversity

#	Source	DF	Sum Sq	Mean Sq	Computed F
1	R	3	3.8493	1.2831	40437.7601
2	P	3	81.5102	27.1701	856282.9984
3	$R \times O$	9	0.4826	0.0536	1689.9835
4	$O \times P$	9	7.1838	0.7982	25155.7570
5	$G \times R \times O$	27	0.1644	0.0061	191.8747
6	$G \times O \times P$	27	1.0641	0.0394	1242.1203
7	$G \times R \times O \times P$	81	0.3613	0.0045	140.5775

Since a large computed F-statistic value relative to the corresponding critical value indicates that the effect under scrutiny is significant under normal circumstances, the ANOVA tables shown would lead one to conclude that all the factors and their interactions have significant effect on the response variables. This is, in fact, conclusive with respect to the single factor rows in Table 5.1 and the tables in the appendix. However, it is not conclusive with respect to the factor interaction effects. Their appearance of significance may be a carry-over effect of the individual factor effects. The standard recourse in such instances is to scrutinize in greater detail the effects. Generally, a graphical approach is used and that approach is taken in the following sections. For completeness, we examine the individual factors this way as well as the factor interactions.

5.2. Effects of individual factors. Note that in the ANOVA analysis, Tables A.1, A.2, and A.3, the computed F-statistic values for migration probability factor are the greatest. Migration probability, in fact, clearly affects all three responses as is evident from Figure 5.1. In the single factor graphs, the feature one looks for in order to confirm visually that a factor affects performance is a curve with a slope other than zero. The response values (Y-axis) for these and succeeding plots are described in Section 4.2. To summarize, the Y-axis of the top graph of Figures 5.1-5.4 is the value from Eq. 4.1. The Y-axis of the middle graph of the figures is the fraction of subpopulations in which a key schema occurs (plausibly) by means of migration. Finally, the Y-axis of the bottom graph is the number of generations required for convergence. For single-factor plots, each data point is an average over all possible combinations of other factors, *i. e.,* each point is an average of 640 values. The error bars depict the standard deviation.

Figure 5.1 emphatically demonstrates that subpopulations when interconnected behave differently than when isolated, *i. e.,* $P = 0$. Further, up to a point, the level of intercommunication, as reflected by the magnitude of migration probability, makes a difference. In the absence of migration, *i. e.,* zero probability, there is little exploitation (*i. e.,* maximal diversity), schema proliferation only by chance rediscovery, and poor search efficiency. In contrast, moderate to high migration probabilities result in significantly different behavior with respect to all response variables. As is seen in the following subsections, migration probability also exhibits significant interaction with other factors.

In Figure 5.2, it can be seen that the migration operator also affects all three responses. The effects, however, are less dramatic compared to the effect of migration probability. Recall that the migration operator components include an import policy and a replacement policy. Figure 5.2 suggests that the former is more important than the latter. In each plot the first data point is similar to the second and the third data point is similar to the last. See again Table 4.2 to correlate the data points with migration policies.

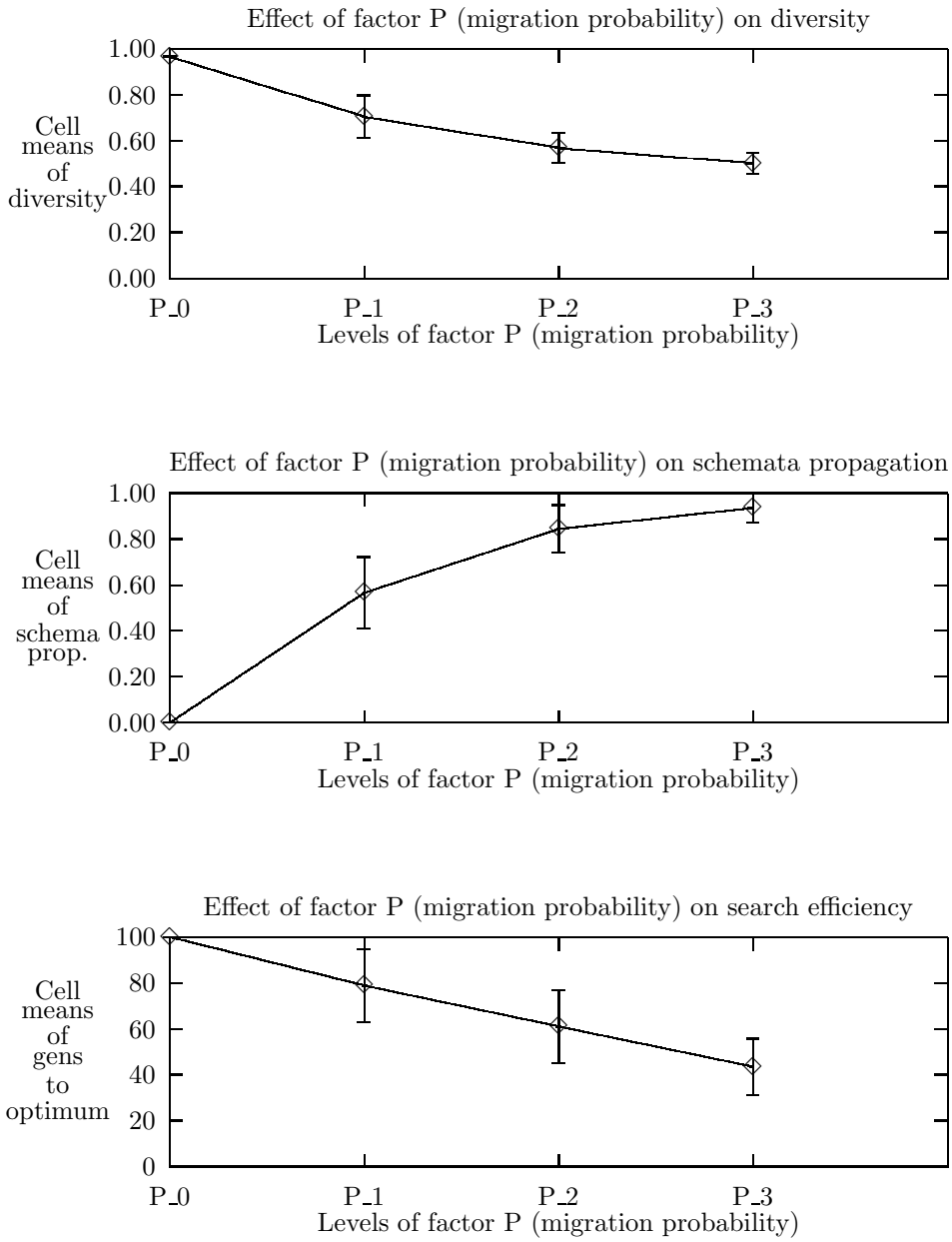


FIG. 5.1. *Effect of factor P (Migration Probability) on the response variables*

Figures 5.3 and 5.4 indicate that the independent effects of topology and migration radius are minimal. The changes in responses between the first and the second data points of each plot might best be explained in terms of changes in the degree of connectivity. The similarity among the remaining data points suggest that further increases in the degree of connectivity have little effect.

5.3. Effects of multiple factor interactions. One method for examining in more detail the higher order interactions is to visually compare different components of the effect under scrutiny, using two factors at a time. The cell means of the response variable are plotted against one of the factors, for each level of the second factor. All the cell means are averaged over all the combinations of levels of the remaining factors.

To illustrate the approach, the effects of the two factor interactions—migration radius vs. migration operator and migration operator vs. migration probability—on the response variable measuring diversity are shown in Figures 5.5 and 5.6 respectively. The null hypothesis here is: The effects of factor level combinations are an

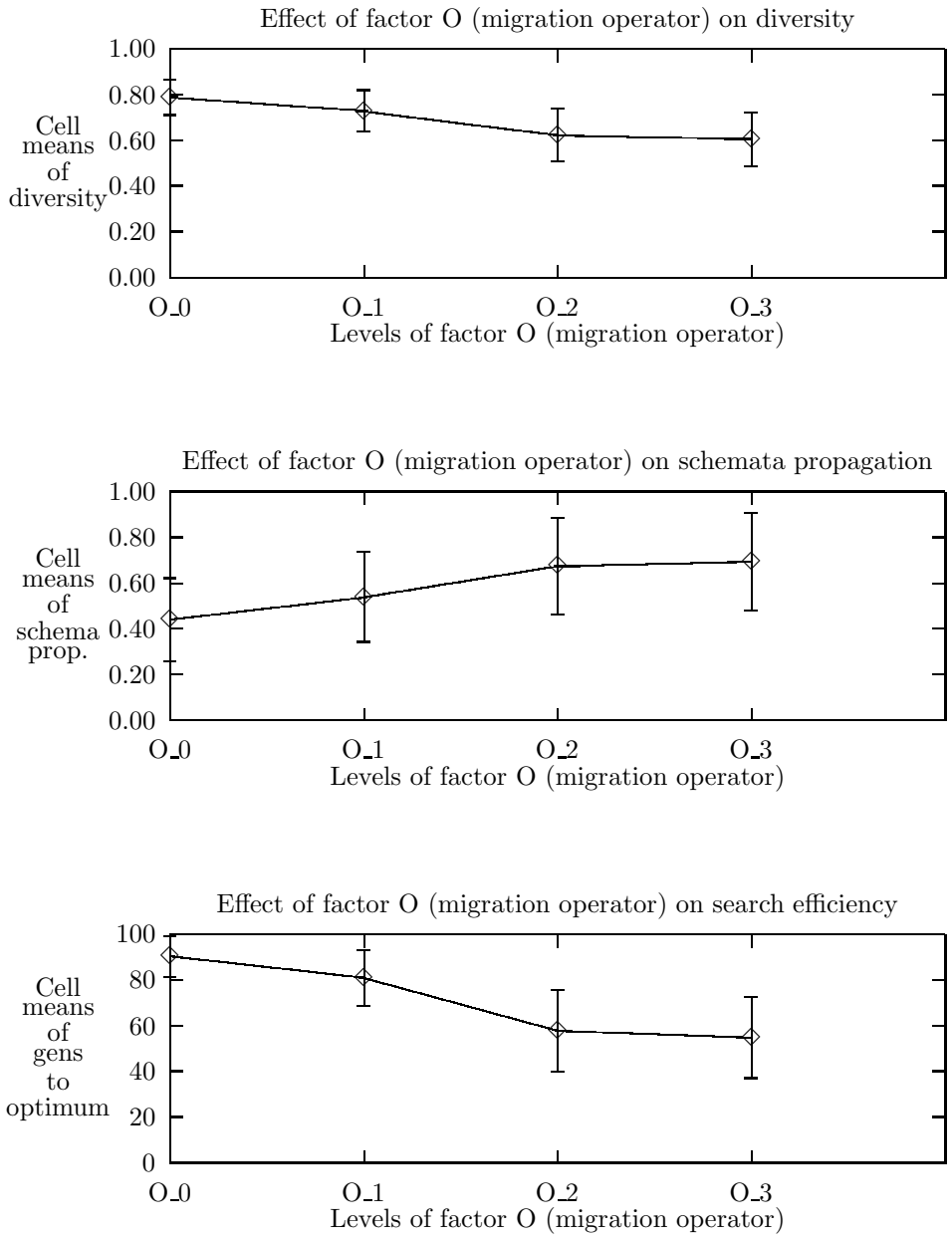


FIG. 5.2. *Effect of factor O (Migration Operator) on the response variables*

additive (linear) combination of the individual factor levels. The visual feature one looks for to "not reject the null hypothesis" is roughly parallel lines. The four plots in Figure 5.5 are roughly parallel to each other, thus the interaction effect is not significant with respect to diversity. Figure 5.6 shows an example that might suggest some factor interaction but the lines are parallel statistically speaking.

When we consider any three-factor interaction, one should examine all the plots that can be generated by choosing two factors at a time for each level of the third factor. For any response variable there are twelve graphs for each of the four possible three-factor interactions. If most of those graphs show parallel plots, we can conclude the effect of the interaction is insignificant. A similar procedure can be carried out to determine the significance of the four-factor interaction.

The complete analysis for three response variables of a $4 \times 4 \times 4 \times 4$ experiment uses more than 450 such graphs. Here we selectively present those which best illuminate the behavior. More details and a complete set of graphs can be found in [35].

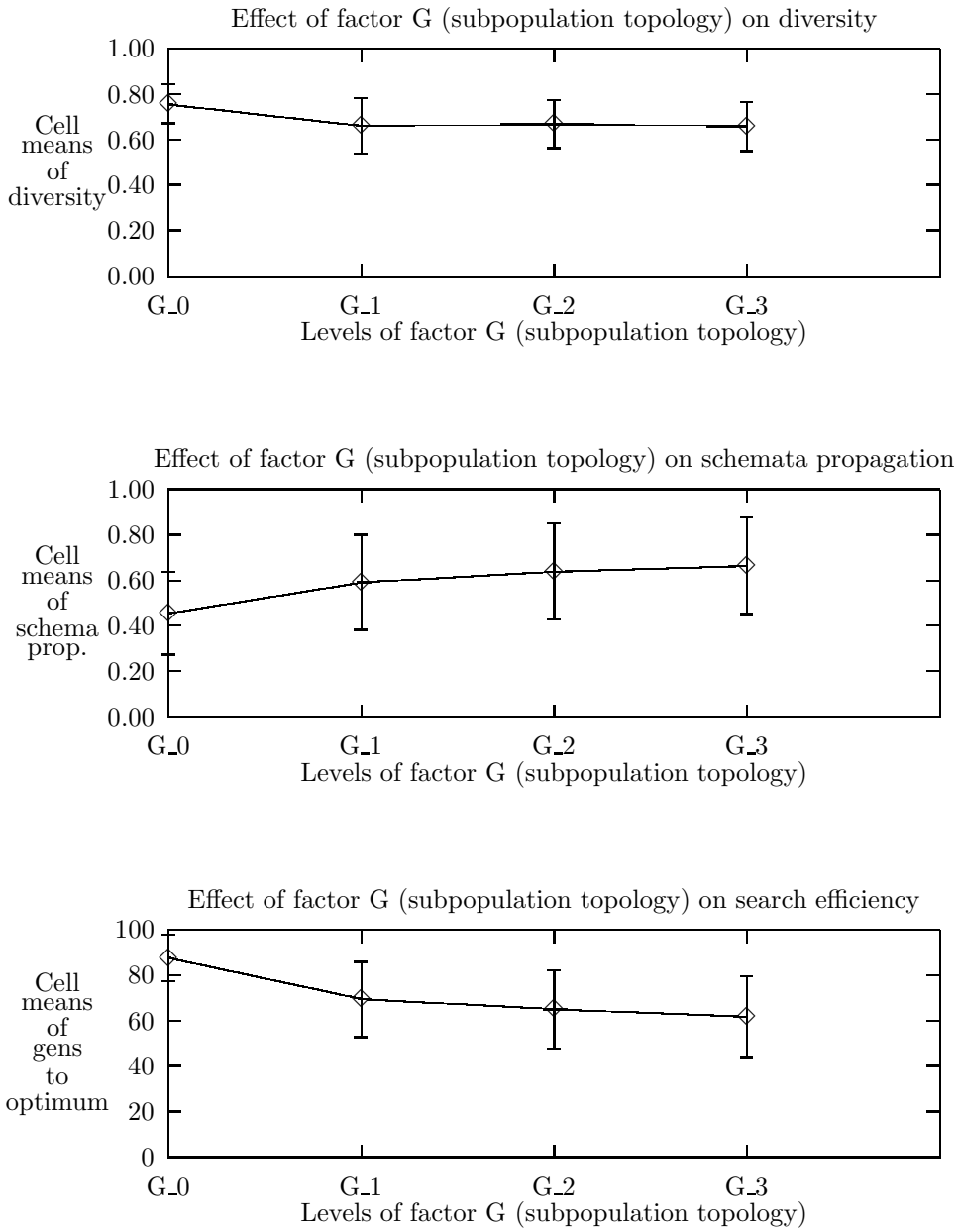


FIG. 5.3. *Effect of factor G (Subpopulation Topology) on the response variables*

Since the migration probability and migration operator have the greatest single factor effects, as can be expected, they exhibit the most pronounced second-order effects. The second order effect of $O \times P$ on diversity is shown in Figure 5.6. Similar behavior can be observed with respect to the other response variables. In Figure 5.6, there is evidence explaining the interaction of $O \times P$. Namely, the import policy impacts the migration effect to a greater extent than the replacement policy. Also, reinforcing our expectations, a migration probability of zero produces the most distinctively different results.

Given that $O \times P$ is the most significant second order interaction, we show first an example of a three-factor interaction not involving both simultaneously. Figure 5.7 depicts some of the effects of the interaction among migration radius, migration operator, and topology, *i. e.,* $R \times O \times G$, on the response variable measuring schemata propagation. It compares the first factor against the second while keeping the third factor constant at some level. In this figure, only the first graph, which corresponds to ring topology, indicates some significance.

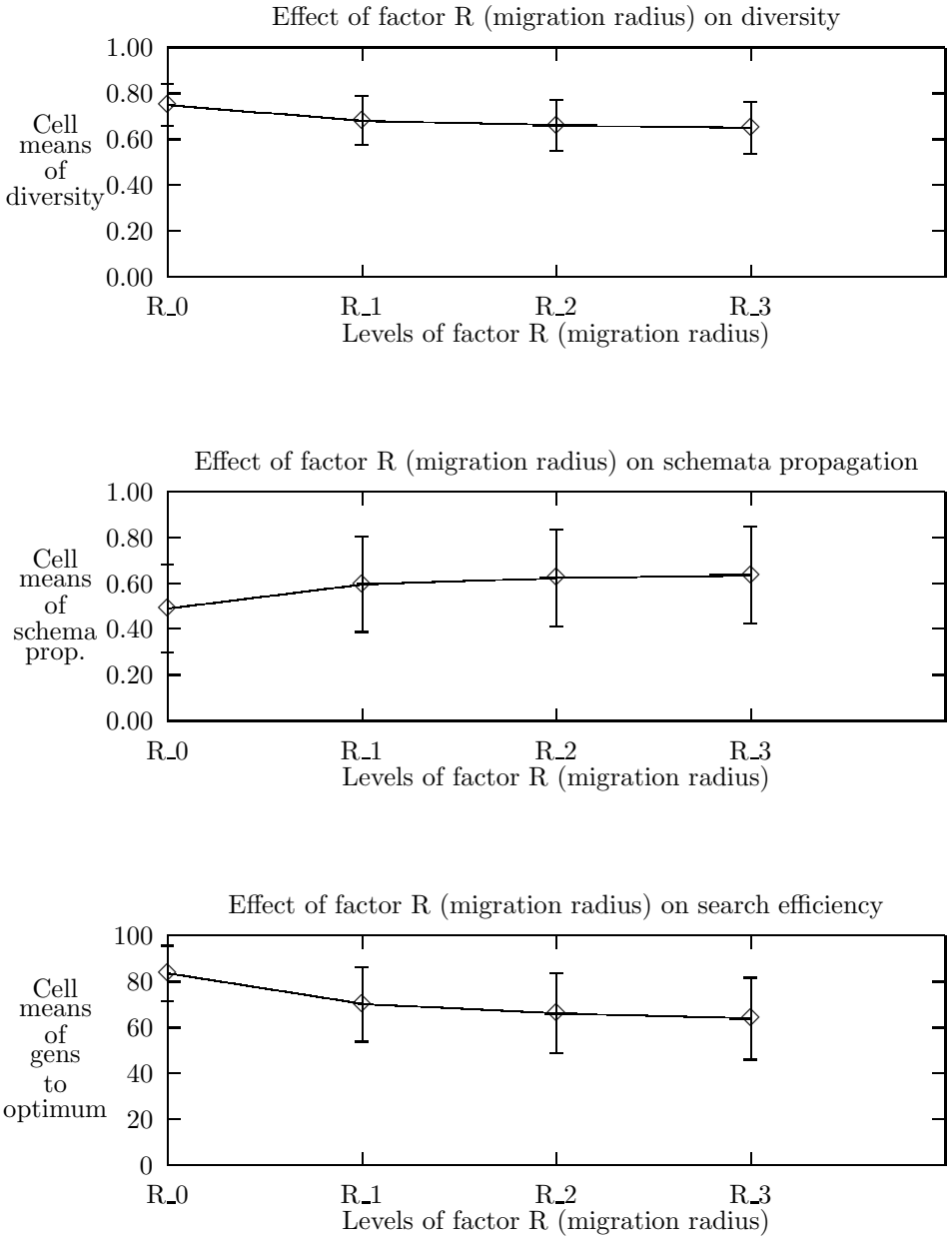


FIG. 5.4. *Effect of factor R (Migration Radius) on the response variables*

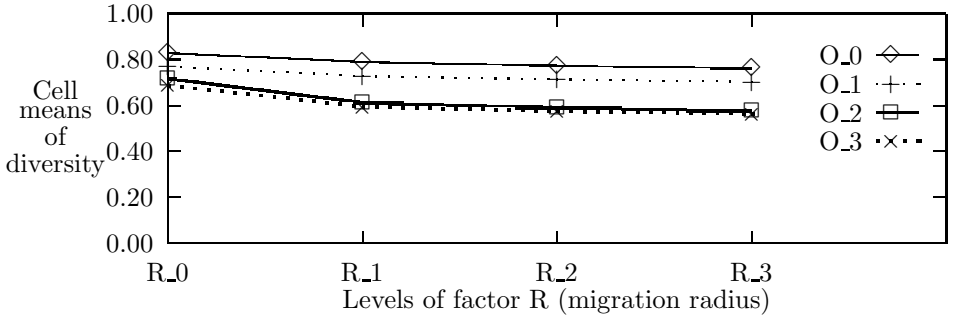


FIG. 5.5. *Effect of RxO interaction on diversity*

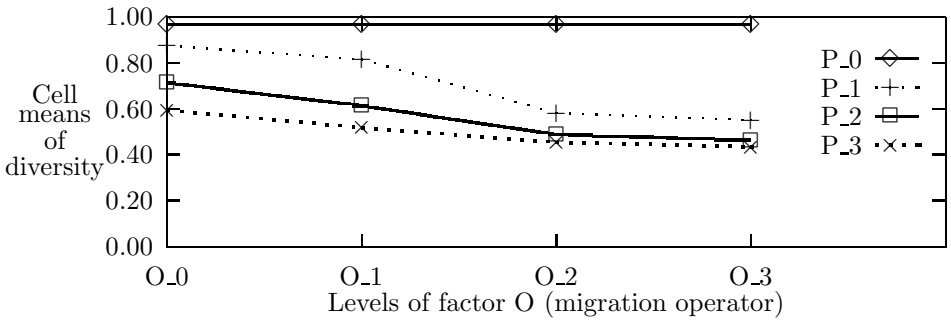


FIG. 5.6. *Effect of OxP interaction on diversity*

This is probably a reflection of the effect migration radius has in a low-connectivity architecture. The plots in terms of other response variables indicate essentially the same behavior.

Contrast Figure 5.7 with Figure 5.8 in which the effects of the interaction among migration radius, migration operator, and migration probability on schemata propagation are shown. While the $R \times O \times P$ interaction effect is not highly pronounced in the figure, it does assert itself in the second plot. This again is further corroboration of the importance of the import policy component of the migration operator.

The computed F-statistic for the fourth order interaction is low in a relative sense in each of the ANOVA Tables A.1, A.2, and A.3 shown in the appendix. This leads one to suspect that the interaction effect is minimal. Figure 5.9, which illustrates the fourth order interaction partially, supports this observation. The plots are in terms of the search efficiency response but the graphs of other response variables lead one to the same conclusion.

5.4. Global Behavior. Recall that in the experiments, the number of cells was 256 ($4 \times 4 \times 4 \times 4$). Since we took 10 samples for each cell (using the same set of 10 random seeds across all the cells) a total of 2,560 runs were made. Of the 2,560 runs, the algorithm found the maximum royal road (RR) level of 4 in 1,316 runs, reached RR level 3 in 1, 834 runs, and reached RR level 2 in 1,920 runs. It reached the first level in all runs. (While we are speaking here of global behavior, let us remind the reader the 640 runs that it failed to climb above level 1 are the same runs for which the migration probability was zero, i. e., the subpopulations were isolated. Table 5.2 summarizes the global behavior. We suggest that the measure "Evaluations (Avg.)"

TABLE 5.2
Summary across experiments.

RR Level	Reached in ...	That is ...	Fuction Evaluations (Avg.)	Generations (Avg.)
1	2,560 runs	100% of runs	163,840	1.00
2	1,920 runs	75% of runs	2,723,840	16.63
3	1,834 runs	72% of runs	6,102,459	37.25
4	1,316 runs	51% of runs	7,809,790	47.67

(the number of function evaluations required to reach a specific level) is not as informative as the number of generations given that 16,384 evaluations are done in parallel. That is 16,384 evaluations occur between points at which the level noted.

In general, suppose someone were to budget or use the same computational effort using a non-parallel GA. An example would be a GA with population size of 100 running for 81,920 generations. Would that setup reeach RR level 4 much faster than the 8,192,000 evaluations used in the MPGA of Table 5.2? That is, would the algorithmic performance of such a setup, as measured by machine cycles needed to find the optimal fitness vlaue, be much better? We think that is extremely likely to be true. For example, you might reach the same fitness value by 5,000 generations. However, the wall-clock time for 5,000 generations will be prohibitive, especially for complex fitness functions. On the other hand, the actual number of fitness function evaluations is not that important for massively parallel GAs because the required wall-clock time is small.

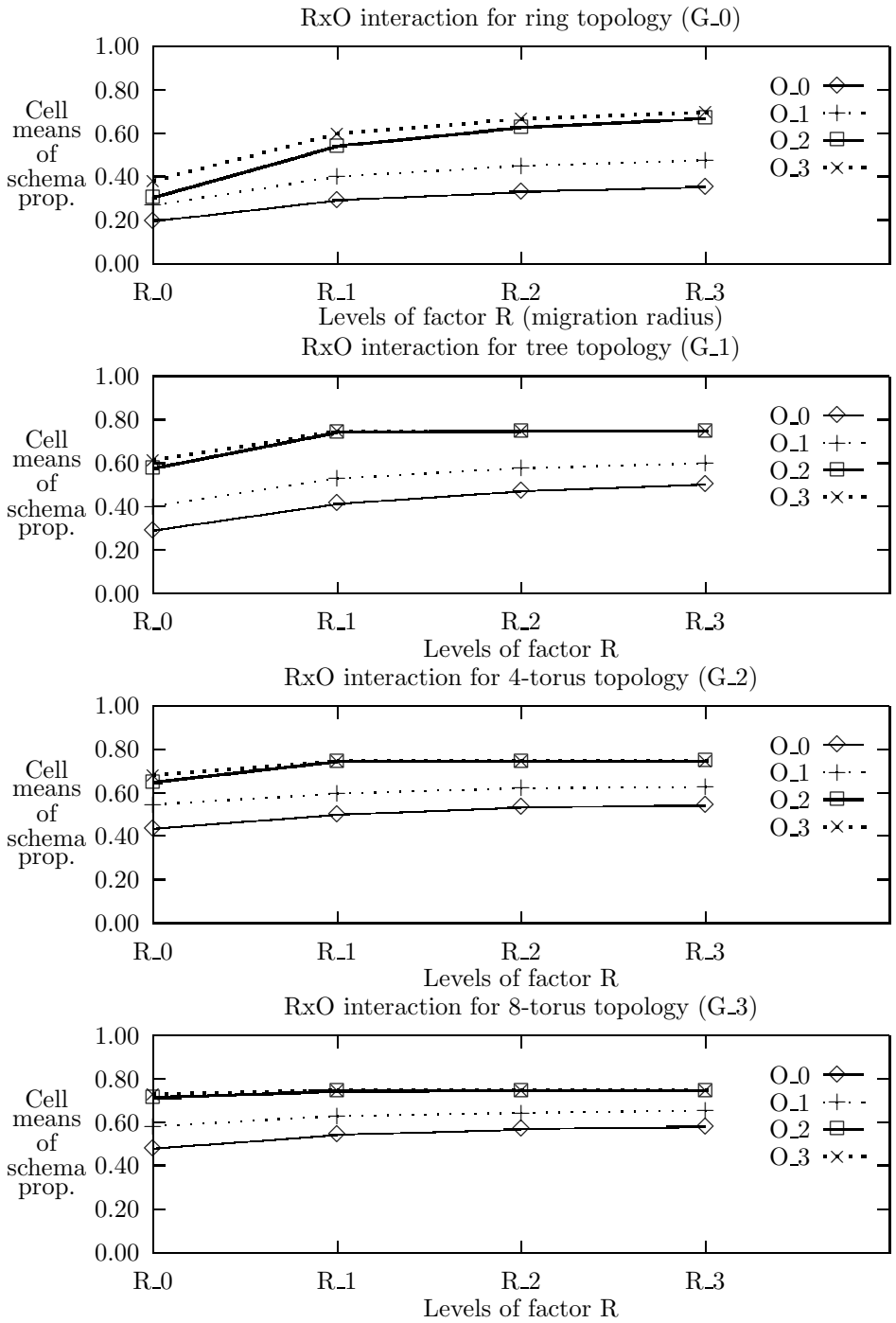


FIG. 5.7. *Partial effect of RxOxG interaction on schemata propagation*

6. Conclusions. The motivation of this study was to rigorously but empirically examine the behavior of GAs in a SIMD environment. Such an environment imposes constraints on the parallelization including uniformity of representation, commonality of operators, and synchronous execution. With this objective in mind, a reasonable set of control parameters and response metrics were devised and a rigorous randomized experiment was performed. Approaching an empirical study from a statistical experiment design is highly effective with respect to the objectives of this study. One purpose of this paper is to demonstrate the methodology.

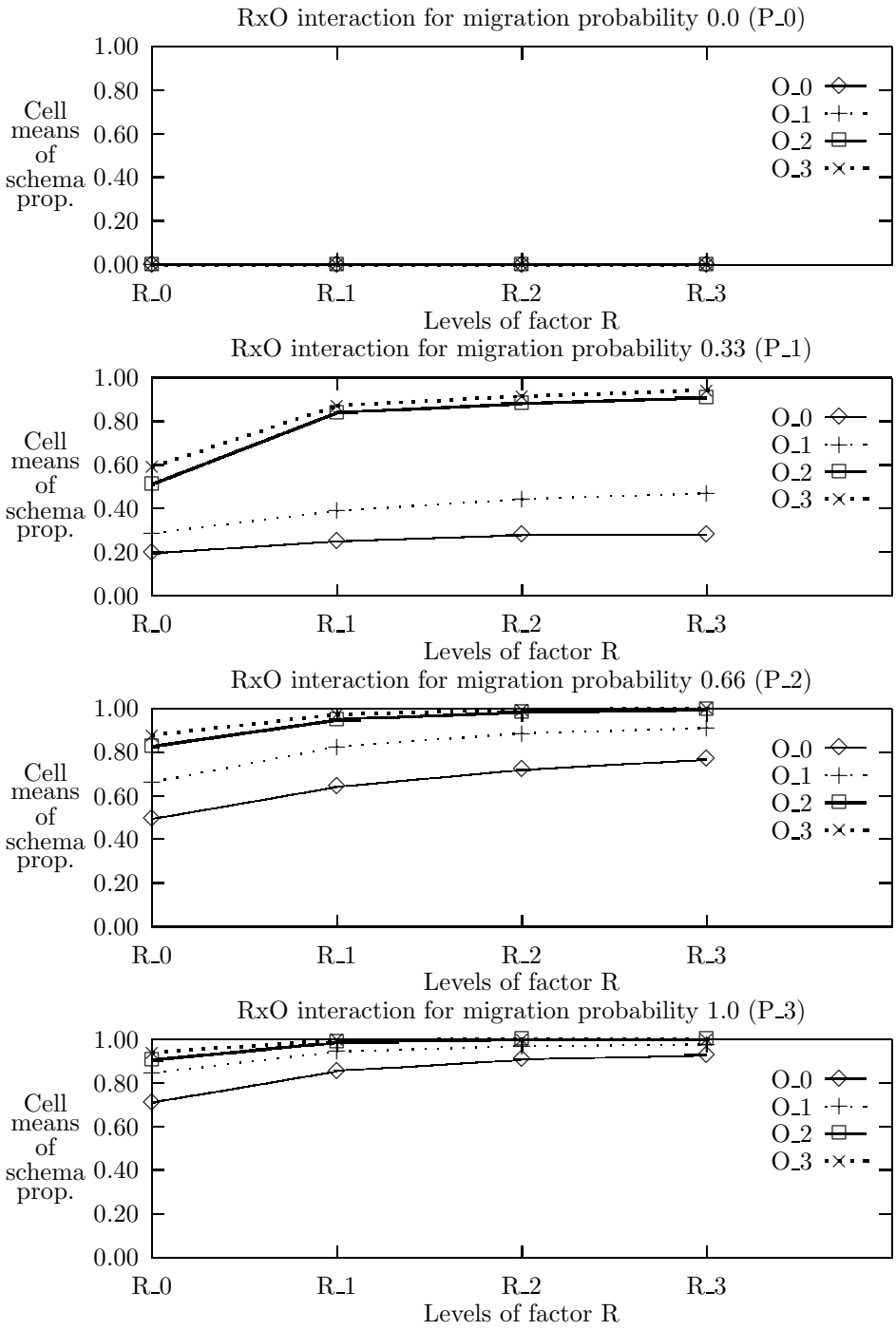


FIG. 5.8. *Partial effect of RxOxP interaction on schemata propagation*

The results demonstrate that migration probability and the choice of migration operator have the greatest impact. This applies to both individual and interaction effects. With respect to the migration operator, the import policy (import random vs. import best) appears more important than the replacement policy. Concerning the other factors, (topology and migration radius), the real effect is due to the degree of connectivity and both factors contribute to this implicit factor. Looking beyond the individual factors and at the degree of connectivity, it seems that above moderate levels, additional increases in connectivity have little additional effect. Finally, the

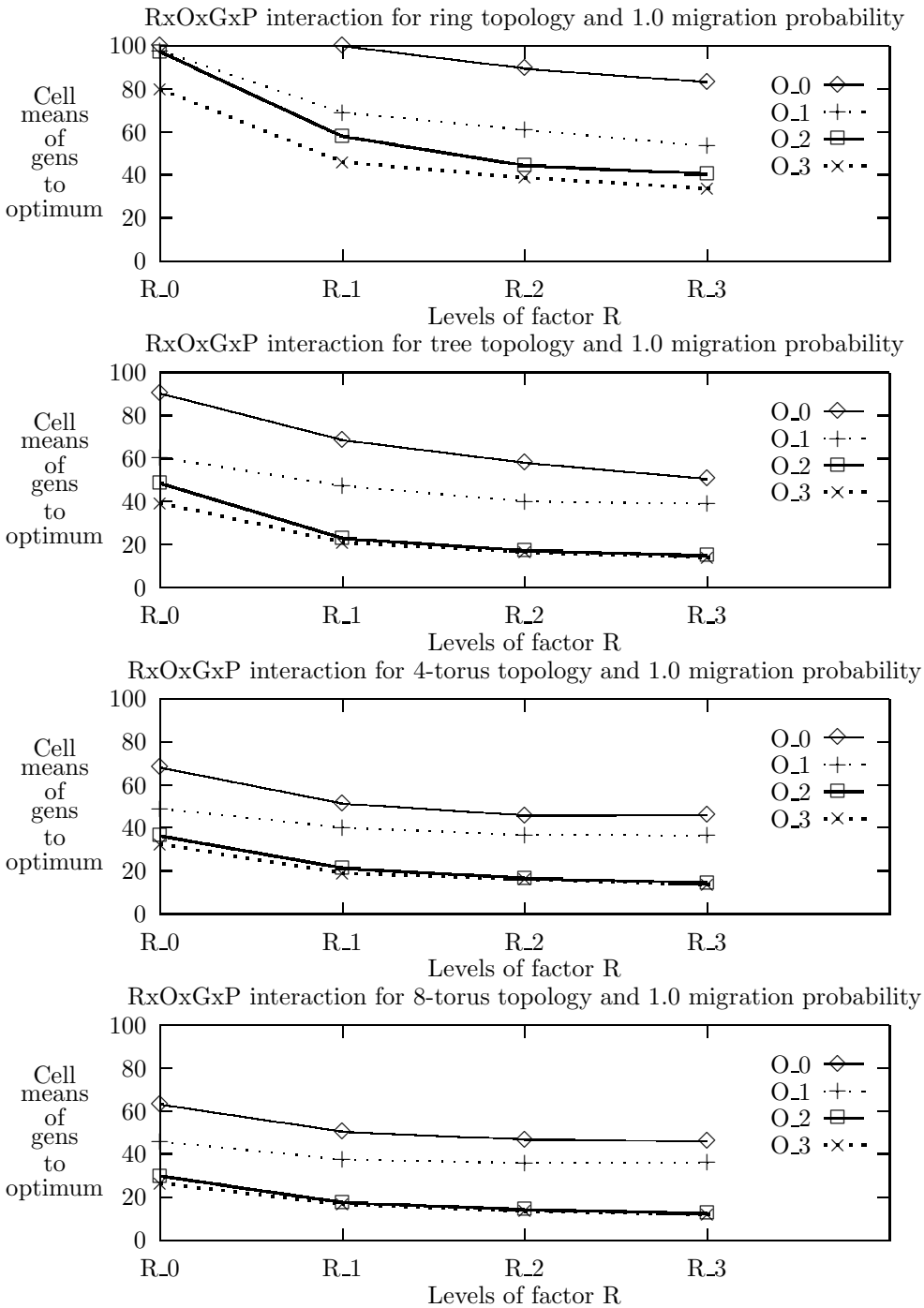


FIG. 5.9. *Partial effect of RxOxGxP interaction on search efficiency.*

results lend strong credence to a conclusion already largely accepted: There is a distinct behavioral difference between isolated subpopulations and those that communicate.

There are issues that are open to further study. Selection radius is one of them. Varying the selection radius has the effect of controlling the number of breeding pools to which an individual can belong. The migration operator, treated here as a single factor, deserves more detailed analysis. It consists of an import policy and a replacement policy. Each was instantiated using two options in the experiment. Other options are possible and the policies could be treated as stand-alone control parameters. Additionally, one might consider an export

policy. On the other side of the ledger, the choice of topology does not seem to be an important one as long as the communication is bidirectional. The native topologies of SIMD architectures, such as a 4- or 8-torus, can be used without impacting the performance.

Acknowledgement. The authors wish to thank Dr. Janet Rice of the Tulane School of Public Health for invaluable guidance in both design of the experiment and interpretation of the results. This work was supported in part by a grant from Naval Oceanographic and Atmospheric Research Laboratory, Stennis Space Center, MS, Grant # N00014-89-J-6003, in part by the NASA High Performance Computing and Communications Program (Earth and Space Sciences Project) Grant NAG 5-2216, and in part by EPSCoR grant NSF/LEQSF-ADP-04.

REFERENCES

[1] E. Alba and C. Cotta, *Evolution of complex data structures*, Informatica y Automatica, 30 (1997), pp. 42–60.
[2] E. Alba and J. M. Troya, *A survey of parallel distributed genetic algorithms*, Complexity, 4 (1999), pp. 31–52.
[3] S. L. amd W. Punch and E. Goodman, *Coarse-grain parallel genetic algorithms: Categorization and new approach*, in 6th IEEE Symp. On Parallel and Distributed Processing, IEEE Press, 1994, pp. 28–37.
[4] R. Belew and L. Booker, eds., *Proc. of Fourth Intern. Conf. on Genetic Algorithms*, San Diego, CA, April 1991, Morgan Kaufmann.
[5] E. Cantú-Paz, *A survey of parallel genetic algorithms*, Calculateurs Paralleles, Reseaux et Systems Repartis, 10 (1998), pp. 141–171.
[6] ———, *Efficient and Accurate Parallel Genetic Algorithms*, Kluwer Academic Publishers, 2000.
[7] ———, *Migration policies, selection pressure, and parallel evolutionary algorithms*, Journal of Heuristics, 7 (2001), pp. 311–334.
[8] E. Cantú-Paz and D. Goldberg, *On the scalability of parallel genetic algorithms*, Evolutionary Computation, 7 (1999), pp. 429–449.
[9] M. Capcarrere, M. Tomassini, A. Tettamanzi, and M. Sipper, *A statistical study of a class of cellular evolutionary algorithms*, Evolutionary Computation, 7 (1999), pp. 255–274.
[10] Y. Censor and S. Zenois, *Parallel Optimization: Theory, Algorithms and Applications*, Oxford University Press, 1998.
[11] ———, *Parallel algorithms in optimization*, in Handbook of Applied Optimization, M. Resende and P. Pardalos, eds., Oxford University Press, 2002, pp. 544–559.
[12] J. P. Cohoon, S. U. Hedge, W. M. Martin, and D. Richards, *Punctuated equilibria: a parallel genetic algorithm*, in Grefenstette [19], pp. 148–154.
[13] R. J. Collins, *Studies in Artificial Evolution*, PhD thesis, University of California, Los Angeles, 1992.
[14] R. J. Collins and D. R. Jefferson, *Selection in massively parallel genetic algorithms*, in Belew and Booker [4], pp. 249–256.
[15] G. Folino, C. Pizzuti, and G. Spezzano, *A parallel hybrid method for SAT that couples genetic algorithms and local search*, IEEE Trans. on Evolutionary Computation, 5 (2001), pp. 323–333.
[16] S. Forrest, ed., *Proc. of Fifth Intern. Conf. on Genetic Algorithms*, Urbana, IL, 1993.
[17] S. Forrest and M. Mitchell, *The performance of genetic algorithms on Walsh polynomials: Some anomalous results and their explanation*, in Belew and Booker [4], pp. 182–189.
[18] M. Gorges-Schleuter, *An analysis of local selection in evolution strategies*, in Proc. of Genetic and Evolutionary Computation Conf., Morgan Kaufmann, 1999, pp. 847–854.
[19] J. J. Grefenstette, ed., *Proc. of Second Intern. Conf. on Genetic Algorithms*, Lawrence Erlbaum Associates, 1987.
[20] G. Harik, E. Cantú-Paz, D. Goldberg, and R. Miller, *The gambler's ruin problem genetic algorithms and the sizing of populations*, Evolutionary Computation, 7 (1999), pp. 231–253.
[21] C. R. Hicks, *Fundamental Concepts in the Design of Experiments*, Holt, Rinehart and Winston, 1973.
[22] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor MI, 1975.
[23] T. Ichimura and Y. Kuriyama, *Learning of Neural Networks with Parallel Hybrid GA Using a Royal Road Function,* IEEE International Joint Conference on Neural Networks, 1998, ppp. 1131–1136
[24] T. Jones, *A description of Holland's Royal Road Function*, Evolutionary Computation, 2 (1994), pp. 409–415.
[25] J. Lienig, *A parallel genetic algorithm for performance-driven VLSI routing*, IEEE Trans. On Evolutionary Computation, 1 (1997), pp. 329–339.
[26] B. Manderick and P. Spiessens, *Fine-grained parallel genetic algorithms*, in Schaffer [38], pp. 428–433.
[27] W. Martin, J. Lienig, and J. Cohoon, *Island (migration) models*, in Handbook of Evolutionary Computation, T. Bäck, D. Fogel, and Z. Michalewicz, eds., Oxford University Press, 1997, pp. C6:5–8–15.
[28] M. Mitchell, S. Forrest, and J. Holland, *The Royal Road for genetic algorithms: Fitness landscapes and GA performance*, in Proc. 1st European Conf. on Artificial Life, MIT Press, 1992, pp. 245–254.
[29] D. C. Montgomery, *Design and Analysis of Experiments*, John Wiley & Sons, 1996.
[30] H. Mühlenbein, *Parallel genetic algorithms, population genetics and combinatorial optimization*, in Schaffer [38], pp. 416–421.
[31] H. Mühlenbein, M. Schomich, and J. Born, *The parallel genetic algorithm as function optimizer*, Parallel Computing, 17 (1992), pp. 619–632.
[32] M. Oussaidene, B. Chopard, O. Pictet, and M. Tommassini, *Parallel genetic programming and its application to trading model induction*, Parallel Computing, 23 (1997), pp. 1183–1198.
[33] C. Pettey, *Population structures: diffusion (cellular) models*, in Handbook of Evolutionary Computation, T. Bäck, D. Fogel, and Z. Michalewicz, eds., Oxford University Press, 1997, pp. C6:4–1–6.
[34] C. Pettey, M. Leuze, and J. Grefenstette, *A parallel genetic algorithm*, in Grefenstette [19], pp. 155–161.

[35] D. Prabhu, *A study in massively parallel genetic algorithms with application to image interpretation*, Tech. Rep. Tech Report 99-5, EECS Dept., Tulane University, New Orleans, 1999.
[36] D. Prabhu, B. P. Buckles, and F. E. Petry, *MPGA: User's guide*, tech. rep., Department of Computer Science, Tulane University, New Orleans, 1995.
[37] W. F. Punch, R. C. Averill, E. D. Goodman, S. C. Lin, and Y. Ding, *Using genetic algorithms to design laminated composite structures*, IEEE Expert, (1995), pp. 42–49.
[38] J. D. Schaffer, ed., *Proc. of Third Intern. Conf. on Genetic Algorithms*, Fairfax, VA, 1989, Morgan Kaufmann.
[39] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das, *A study of control parameters affecting online performances of genetic algorithms for function ptimization*, in Schaffer [38], pp. 51–60.
[40] M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*, Springer Verlag, 1997.
[41] J. Stender, *Parallel genetic algorithms theory and applications*, in Frontiers in Artificial Intelligence and Applications, J. Stender, ed., IOS Press, 1993.
[42] H. Suzuki and H. Sawai, *Crossover accelerates evolution in GAs with a Royal Road function*, in Proc. Genetic and Evolutionary Computation Conference, 2001, pp. 405–412.
[43] R. Tanese, *Parallel genetic algorithms for a hypercube*, in Grefenstette [19], pp. 177–183.
[44] ———, *Distributed genetic algorithms*, in Schaffer [38], pp. 434–440.
[45] M. Tomassini, *Parallel and distributed evolutionary algorithms: A review*, in Evolutionary Algorithms in Engineering and Computer Science, K. Miettien, M. Mäkekä, P. Neittaanmäki, and J. Périaux, eds., J. Wiley and Sons, 1999, pp. 113–133.
[46] S. Tsutsu and Y. Fujimoto, *Forking genetic algorithm with blocking and shrinking modes (fGA)*, in Forrest [16], pp. 206–213.
[47] D. Whitley, *Cellular genetic algorithms*, in Forrest [16], pp. 658–666.
[48] D. Whitley, S. Rana, and R. Heckendorn, *Exploiting seperability in search: The island model genetic algorithm*, Journal of Computing and Information Technology, 7 (1999), pp. 33–47.
[49] S. Wong, C. Wong, and C. Tong, *A parallelized genetic algorithm for the calibration of the Lowry Model*, Parallel Computing, 27 (2001), pp. 1523–1536.
[50] S. Yang, *Primal-dual genetic algorithms for Royal Road functions*, in Proc. of the 15th IFAC World Congress, Barcelona, Spain, 21-26 July 2002, pp. 439–446.

Appendix A.

Analysis of Variance (ANOVA) for various experimental designs is adequately covered in numerous texts (e.g. [21, 29]). The experiment design used here is known as *randomized factorial*. There are four factors and four levels of each, thus 4^4 level combinations. For each level of each factor, g, r, o, p, the outcome of a single repetition, k, is represented by y_{gropk}. For this experiment, $g \in \{1,2,3,4\}$, $r \in \{1,2,3,4\}$, and $o \in \{1,2,3,4\}$, and $p \in \{1,2,3,4\}$, and $k \in \{1,2,\ldots,10\}$, yielding 10×4^4 individual trials. The *degrees of freedom* are (one less) than than the number of choices possible for a component. For total trials (line 17 in Tables A.1-A.3) there are 2,559 ($10 \times 4^4 - 1$) degrees of freedom. For any given factor (lines 1–4 in Tables A.1–A.3) there are three (four levels – 1) degrees of freedom. For combinations of factors, the degrees of freedom are the product of the degrees for individual factors (*i. e.*, nine for pairs, 27 for triples, and 81 for the combination of all four factors).

Table A.1
ANOVA for global allelic diversity

#	Source	DF	Sum Sq	Mean Sq	Computed F
1	G	3	4.2971	1.4324	45141.8887
2	R	3	3.8493	1.2831	40437.7601
3	O	3	14.6118	4.8706	153500.4667
4	P	3	81.5102	27.1701	856282.9984
5	$G \times R$	9	0.6086	0.0676	2131.1502
6	$G \times O$	9	0.9484	0.1054	3321.1950
7	$G \times P$	9	1.5872	0.1764	5558.0741
8	$R \times O$	9	0.4826	0.0536	1689.9835
9	$R \times P$	9	1.6751	0.1861	5865.8867
10	$O \times P$	9	7.1838	0.7982	25155.7570
11	$G \times R \times O$	27	0.1644	0.0061	191.8747
12	$G \times R \times P$	27	0.4341	0.0161	506.6511
13	$G \times O \times P$	27	1.0641	0.0394	1242.1203
14	$R \times O \times P$	27	0.8969	0.0332	1046.8983
15	$G \times R \times O \times P$	81	0.3613	0.0045	140.5775
16	Errors	2304	0.0731	0.0000	
17	Totals	2559	119.7481		

TABLE A.2
ANOVA for schemata propagation

#	Source	DF	Sum Sq	Mean Sq	Computed F
1	G	3	16.6509	5.5503	25888.3304
2	R	3	8.4362	2.8121	13116.4494
3	O	3	27.5620	9.1873	42852.7267
4	P	3	340.6299	113.5433	529602.2027
5	$G \times R$	9	2.4828	0.2759	1286.7461
6	$G \times O$	9	0.7596	0.0844	393.6835
7	$G \times P$	9	6.8043	0.7560	3526.3902
8	$R \times O$	9	0.1485	0.0165	76.9467
9	$R \times P$	9	3.6515	0.4057	1892.4426
10	$O \times P$	9	22.7254	2.5250	11777.6365
11	$G \times R \times O$	27	0.8185	0.0303	141.4041
12	$G \times R \times P$	27	1.5875	0.0588	274.2432
13	$G \times O \times P$	27	3.7258	0.1380	643.6473
14	$R \times O \times P$	27	2.2732	0.0842	392.7038
15	$G \times R \times O \times P$	81	1.7956	0.0222	103.3959
16	Errors	2304	0.4940	0.0002	
17	Totals	2559	440.5459		

TABLE A.3
ANOVA for search efficiency

#	Source	DF	Sum Sq	Mean Sq	Computed F
1	G	3	254008.7824	84669.5941	9667.5576
2	R	3	149389.6855	49796.5618	5685.7616
3	O	3	579318.1168	193106.0389	22048.8096
4	P	3	1124711.5105	374903.8368	42806.4464
5	$G \times R$	9	7361.0035	817.8893	93.3864
6	$G \times O$	9	46817.6848	5201.9650	593.9593
7	$G \times P$	9	89067.4410	9896.3823	1129.9670
8	$R \times O$	9	42775.2691	4752.8077	542.6746
9	$R \times P$	9	50002.2004	5555.8000	634.3602
10	$O \times P$	9	242684.8566	26964.9841	3078.8566
11	$G \times R \times O$	27	3590.1730	132.9694	15.1824
12	$G \times R \times P$	27	25966.8418	961.7349	109.8107
13	$G \times O \times P$	27	68730.3230	2545.5675	290.6524
14	$R \times O \times P$	27	41988.8262	1555.1417	177.5658
15	$G \times R \times O \times P$	81	34339.4254	423.9435	48.4058
16	Errors	2304	20178.7000	8.7581	
17	Totals	2559	2780930.8402		

The fundamental idea in design of experiments is that if a factor has no significance, then the distribution of all responses is the same as the distribution of responses for a given level of the factor. That is, we could check the total mean against the mean for a specific factor level. Parametric comparisons in statistics more frequently employ an equivalent method: Determine if there are differences in the variances (mean squares) among factor levels. Let $\overline{y}_{g....}$ be the average of responses for a particular option, g, for the topology across all other factors and repetitions. Similarly, let $\overline{y}_{.....}$ be the overall average for all (in this case, 2,560) responses. To describe via example, the sum of squares for line 1 of Tables A.1-A.3 is

$$\sum_{g} n(\overline{y}_{g....} - \overline{y}_{.....})^2$$

where n is the number of replications (10 in our case). The mean square is the sum of squares divided by the degrees of freedom. The total sum of squares is $\sum_g \sum_r \sum_o \sum_p \sum_p (y_{gropk} - \overline{(y)}_{.....})^2$. The "easy" way to compute the error sum of squares is subtract the individual factor sums from the total sum.

Variances are compared by dividing the factor mean square by the error mean square (the F-statistic). Standard tabulations of critical F-statistic values can then be consulted to determine if the effect of a factor or an interaction on the given response variable is significant. ANOVA results for the response variables measuring population diversity, schemata propagation, and search efficiency are shown in Tables A.1, A.2, and A.3, respectively.

Edited by: Marcin Paprzycki.
Received: February 10, 2003.
Accepted: June 16, 2003.

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 87–91. http://www.scpe.org
ISSN 1895-1767

PARALLEL STANDARD ML WITH SKELETONS

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

Abstract. We present an overview of our system for automatically extracting parallelism from Standard ML programs using *algorithmic skeletons*. This system identifies a small number of higher-order functions as sites of parallelism and the compiler uses profiling and transformation techniques to exploit these.

Key words. automated parallelization, higher-order functions, algorithmic skeletons.

1. Introduction. The exploitation of parallelism in programs is greatly eased by tools based on implicit parallelism. The PMLS compiler realises higher order functions (HOFs) in Standard ML (SML) programs as parallel algorithmic skeletons. An SML program is treated as a prototype of the final parallel implementation. Static analysis and dynamic instrumentation, combined with performance models for skeletons, enable the identification of useful parallelism. Prototype transformation is employed to try and optimise parallelism. Where exploitable parallelism cannot be identified, program synthesis is used to introduce new instances of HOFs. Here, we describe the compiler and the methodology it is intended to support.

2. A Skeletons Methodology. The compiler was originally motivated by extensive exploration of algorithmic skeletons for the construction of parallel computer vision systems from functional prototypes [12, 17]. Since that time our compiler has matured into a more general-purpose parallel programming language based on SML, a mature functional language with a stable formal definition [13]. Although our methods are applicable to low-level programming such as image processing they are most useful for high-level programming with complex algorithmic structures. For prototyping, SML provides closeness to formalisms for proof and transformation. For parallel prototyping, SML's strictness is better suited than the laziness of Haskell [10] as it results in more predictable behaviour.

We focus on common low-level HOFs such as `map` and `fold` which are ubiquitous in functional programs. Explicit HOF names are used as the basis for identification:

```
fun map f [] = []
  | map f (h::t) = f h::map f t
fun fold (f:'a*'a->'a) b [] = b
  | fold f b (h::t) = f (h,fold f b t)
fun filter p [] = []
  | filter p (h::t) = if p h then filter p t else h::filter p t
fun compose (f:'b list -> 'c list) (g:'a list -> 'b list) x = f (g x)
fun tuple2 (fa,fb) = (fa (),fb ())
```

`map` and `fold` HOFs may be synthesised from arbitrary recursive functions [7], and implemented in parallel in a variety of ways [9]. `filter` can be implemented constructively from `map` with minimal overhead. Function composition can be realized in parallel provided the argument is a decomposable datatype. Tuple parallelism can be implemented by turning tuples into suspensions:

```
(a,b)  ⇔  tuple2 (fn _ => a,fn _ => b)
```

Transformations may be defined over these base functions to implement simple identification, distribution and equivalences. For example, we can mitigate communications costs by aggregation:

```
(map f) o (map g)  ⇔  map (f o g)
```

If a `fold` function argument is not associative, we can sometimes extract partial parallelism using `map`:

```
fold (fn (h,t) => f (g h) t) b l  ⇔  fold f b (map g l)
```

*School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa, Japan, 923-1292 (norman@jaist.ac.jp).
†Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, (greg@cee.hw.ac.uk).
‡School of Information Science, Japan Advanced Institute of Science and Technology, Tatsunokuchi, Ishikawa, Japan 923-1292 (hori@jaist.ac.jp)

We can switch between alternative implementations of the same skeleton:

```
map f  ⇔  fold (fn (h,t) => f h::t) []
```

Finally, we can move sites of argument instantiation to allow pre-computation before distribution:

```
hof (f x)  ⇔  let val x' = f x in hof x' end
```

Given the high computational complexity of program transformation systems we need a fast method of assessing the impact of transformations upon parallel performance. Using the SML definition in conjunction with an SML interpreter we can accurately summarise the semantic behaviour of an executing SML program. This behaviour can then be related to the execution times of compiled programs.

Cost models for algorithmic skeletons have been well-studied [16, 14]. Combining the profiling information with data size measurements results in the ability to instantiate these cost models giving predictions of the effect of transformations.

Partitioning transformations into; identification, optimisation, and restructuring, we propose a methodology for parallel functional prototyping which can be summarised as follows:

1. A sequential prototype is transformed using restructuring and identification transformations to lift as many HOF instances from arbitrary code as possible.
2. The computational loads for HOF instance functions and communication costs for their arguments and results are determined.
3. This data is applied to models for the equivalent algorithmic skeletons to determine the viability of potential parallel implementations.
4. If no parallelism is predicted, the prototype is transformed using optimisation and retsructuring transformations. to optimise the computational versus communication costs in the models.
5. Where useful parallelism is predicted the HOFs should be realised as, possibly nested, algorithmic skeleton instantiations.

3. The Parallel SML with Skeletons (PMLS) Compiler.

3.1. Design. Our experience with the PUFF [4] and SkelML [3] compilers, combined with that gained in developing parallel computer vision systems led to the design of a more general parallelizing compiler for SML [11], with the following properties:

- The full SML Core language is supported.
- Dynamic profiling provides parallel performance prediction.
- Heuristics-guided transformations drive performance optimisation.
- Transformations may be generated using automated proof synthesis.
- Skeletons can be nested but are not first-class objects.
- The compiler targets a broad range of general-purpose parallel computers.

The spine of the PMLS compiler was constructed from 1997 to 2000, incorporating `map` and `fold` skeletons. The performance prediction has been developed to a state where usably accurate predictions are provided for these two skeletons. Further skeletons are under development but are not currently modelled. At present we can present results for the manual analysis of exemplars and are currently automating this process.

3.2. Implementation. The host compilers are the ML Kit [2] for the front end, profiling and transformation systems, and Objective Caml [5] for backend compilation and execution. This combination allows the transformational compiler to be implemented on a separate machine so that the backend can be kept small and retargetable.

3.3. Analysis. During our analysis, the SML syntax tree is mapped onto a static network of processors. We generate an *abstract network* description of the program which defines the relationship between the HOFs in the sequential prototype and between executing skeletons in the parallel version. A two-phase algorithm is implemented whereby the skeleton information is added to the existing types and then the type information is stripped out leaving the network description:

```
val ff3 = map (fn x => x + 1) [1,2,3]
val ff3 :: node(map,base list,[(int->base),base list])
```

The handling of free values complicates our analysis. We wish to avoid the transmision of functional data at runtime as large free value bindings can overload the communications. For non-functional free values, lambda-lifting [8] is sufficient. Functions are augmented with additional formal parameters for their free variables and calls to those functions are extended with the corresponding free variables as the actual parameters. For skeleton instances, the free values are registered with the runtime system and transmitted to the point of function application.

For free functionals we use defunctionalization [1]. In this technique, closures are lifted to the top level of the language and represented by datatypes. This allows free functionals to be handled in the same way as free data but creates a global overhead of a datatype dereference for every function application. The defunctionalization of SML is problematical, however. The published algorithms all require forward code-references which would necessitate runtime registration of functions with attendant jump tables for SML. We adopt a solution whereby the entire program is turned into a single mutually-recursive block.

Finally, we generate launch-code for the skeletons which replace HOFs. This involves detecting free values, replacing the HOF with the skeleton call and reconstructing the type information. Our analysis thus converts the following code:

```
fun ff (y,z) = x + y + z
val result = fold ff (~x) [1,2,3]
```

into:

```
fun ff1 (x) (y,z) = x + y + z
val _ = register ‘‘ff1" ff1
val result =
  (fn _ => (pfold : string -> int -> int list -> int) ‘‘ff1" (~x) [1,2,3])
  (register ‘‘ff1_fvs" (x))
```

3.4. Dynamic profiling. PMLS includes an integrated *dynamic* profiling mechanism [15]. The ML Kit interpreter has been modified to annotate the syntax tree with counts of rules in the semantics which are fired during execution. Given a suitable set of training programs it is possible to assign a weight to each rule in the semantics. The test programs cause as many of the different rules to be fired as possible and the actual execution times for the test programs form the dependent variable for weight determination. This system can be expressed in matrix form as $Pw = x$, where P is an $r \times n$ matrix where r is the number of rules and n the number of program executions, w is a vector of weights and x is the vector of execution times. We solve this equation for the training-set, giving a set of weights which can be applied to a new, unknown profile of rule counts to give a predicted execution time. It is important to select a suitable set of test programs whereby each rule has a significant representation and no rules dominate the entire data set.

Currently, we can use either numerical analysis methods or genetic algorithm techniques to solve the above equation for our training-set. The numerical methods give more accurate fits but suffer from numerical instability which limits the range of validity of the generated weights. The genetic solution is extremely slow and much less accurate but has less numerical instability. At best this technique only gives a *rule-of-thumb* estimate of the sequential performance of an arbitrary section of code, typically within about a 200% margin. This is sufficient, however, to drive a performance-improving transformation system.

3.5. Transformation. PMLS transformations are defined by associating equivalent SML constructs (either *expressions* or *declarations*). The transformations are elaborated and type information is preserved on transformation application. The resulting system allows simple identification and optimization:

```
dtrans T2008 (FF,F,H,T,TAIL) = fun FF [] = TAIL
                                 | FF (H::T) = F::FF T
                           ==> fun FF l = (map (fn H => F) l)@TAIL
end
trans T202 (F,G,L) = fn (F,G,L) => map F (map G L)
                 ==> fn (F,G,L) => map (fn x => F (G x)) L
```

We are currently constructing the transformation engine using the performance prediction results to rank improving transformations and prune non-useful ones.

3.6. HOF Synthesis. To support the programmer with automatic detection of HOFs, Cook investigated automatic extraction of HOFs using proof-planning techniques [7]. The λ-CLAM proof-planner is employed to locate instances of `map`, `fold` and `scan` using higher-order unification and middle-out reasoning. Currently, we use the HOF synthesizer as a pre-processor. For example, the following functions:

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

yield the following equivalent programs:

```
fn x => map (fn y => map (fn (z:int) => z*z) y) x
fn x => map (fn y => foldr (fn (z:int,u) => z*z::u) [] y) x
fn x => map (fn y => squares y) x
```

4. Performance. We have employed PMLS to parallelise a wide range of SML programs. Substantial exemplars include island model genetic algorithms, arbitrary length integer matrix multiplication, linear equation solving and ray tracing. In general, skeleton parallelism tends to be coarse grain. For such exemplars, PMLS offers useful, scalable speedup, typically on up to 16 processors. These applications are all regular parallelism but we hope to tackle more complex irregular problems with future enhancements to the compiler. The performance of our compiler compares favourably with other similar approaches [9]. The following table summarizes the comparison on a Beowulf class workstation cluster (SRT=sequential runtime, PRT=parallel runtime, SP=speedup):

	Eden			GpH			PMLS		
	SRT	PRT	SP	SRT	PRT	SP	SRT	PRT	SP
matmult	38.5s	13.2s	2.9	30.3s	8.9s	3.4	22.8s	4.3s	5.2
linsolv	491.7s	35.1s	14.0	307.9s	25.9s	11.9	190.8s	16.1s	11.9
raytracer	177.4s	13.4s	13.3	163.3s	24.1s	6.8	172.1s	11.4s	15.2

PMLS can generate native code for a variety of CPUs. Consistent cross-platform performance as been shown on the Cray T3E, Fujitsu AP3000, Sun Enterprise, IBM SP2 and Beowulf-class workstation clusters.

5. Concluding Remarks. PMLS demonstrates the potential of exploiting implict parallelism through the combination of a variety of static and dynamic program analysis techniques targeted at skeleton-oriented parallel implementation. This results in a simple method of introducing parallelism with minimal burdens upon the programmer. Our approach is novel in; matching parallel topology to algorithm rather than algorithm to topology, basing profiling on semantic entities rather than absolute or simulated times, and closely coupling instrumentation, analysis and transformation. PMLS is primarily a research vehicle. Current challenges include; combining speed with stability in performance prediction, broadening the range of exploitable HOFs, and exploiting parallelism in the presence of conditionals.

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

REFERENCES

[1] J. M. Bell, F. Bellegarde, and J. Hook, *Type-driven defunctionalization,* In Proceedings of the ACM SIGPLAN ICFP '97, pages 25–37. ACM, Jun 1997.

[2] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner, *The ML Kit (Version 1),* Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[3] T. Bratvold, *Skeleton-based Parallelisation of Functional Programmes,* PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[4] D. Busvine, *Detecting Parallel Structures in Functional Programs,* PhD thesis, Heriot-Watt University, Riccarton, Edinburgh, 1993.

[5] E. Chailloux, P. Manoury, and B. Pagano, *Développement d'applications avec Objective Caml,* O'Reilly, Paris, Apr 2000.

[6] A. Cook, A. Ireland, and G. Michaelson, *Higher-order Function Synthesis through Proof Planning,* In Proceedings of 16th Annual International Conference on Automated Software Engineering (ASE 2001), pages 307–310, San Diego, USA, Nov 2001. IEEE Computer Society.

[7] A. Cook, A. Ireland, G. Michaelson and N. Scaife, *Discovering Applications of Higher Order Functions Through Proof Planning,* Formal Aspects of Computing, V. 17(1), pp. 38–57,2005.
[8] T. Johnsson, *Lambda Lifting: Transforming Programs to Recursive Equations,* In J.-P. Jouannaud, editor, Functional Programming Languages and Computer Architecture, volume 201 of LNCS, pages 190–302. Springer, 1985.
[9] H-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, Á. J. Rebón Portillo, S. Priebe, and P. W. Trinder, *Comparing Parallel Functional Languages: Programming and Performance,* Higher-order and Symbolic Computation, 16(3), pp. 203–251, 2003.
[10] H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L Peyton Jonem es, *Engineering Parallel Symbolic Programs in GPH,* Concurrency—Practice and Experience, 11:701–752, 1999.
[11] G. Michaelson, N. Scaife, P. Bristow, and P. King, Nested Algorithmic Skeletons from Higher-Order Functions. *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, 16(2–3):181–206, 2001.
[12] G. J. Michaelson and N. R. Scaife, Prototyping a parallel vision system in Standard ML. *Journal of Functional Programming*, 5(3):345–382, Jul 1995.
[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised),* MIT Press, 1997.
[14] R. Rangaswami, *A Cost Analysis for a Higher-Order Parallel Programming Model,* PhD thesis, University of Edinburgh, 1995.
[15] N. Scaife, S. Horiguchi, G. Michaelson and P. Bristow, *A Parallel SML Compiler Based on Algorithmic Skeletons,* Journal of Functional Programming, V. 15(4), pp. 615–650, 2005.
[16] D. B. Skillicorn and W. Cai, *A Cost Calculus for Parallel Functional Programming,* Journal of Parallel and Distributed Programming, 28(1):65–84, 1995.
[17] A. M. Wallace, G. J. Michaelson, N. Scaife, and W. J. Austin, *A Dual Source, Parallel Architecture for Computer Vision,* The Journal of Supercomputing, 12(1/2):37–56, Jan/Feb 1998.

Edited by: H. Shen.
Received: June 3, 2002.
Accepted: December 19, 2002.

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 93–106. http://www.scpe.org
ISSN 1895-1767

A CLASS OF PARALLEL MULTILEVEL SPARSE APPROXIMATE INVERSE PRECONDITIONERS FOR SPARSE LINEAR SYSTEMS

KAI WANG *, JUN ZHANG†, AND CHI SHEN‡

Abstract. We investigate the use of the multistep successive preconditioning strategies (MSP) to construct a class of parallel multilevel sparse approximate inverse (SAI) preconditioners. We do not use independent set ordering, but a diagonal dominance based matrix permutation to build a multilevel structure. The purpose of introducing multilevel structure into SAI is to enhance the robustness of SAI for solving difficult problems. Forward and backward preconditioning iteration and two Schur complement preconditioning strategies are proposed to improve the performance and to reduce the storage cost of the multilevel preconditioners. One version of the parallel multilevel SAI preconditioner based on the MSP strategy is implemented. Numerical experiments for solving a few sparse matrices on a distributed memory parallel computer are reported.

Key words. Sparse matrices, parallel preconditioning, sparse approximate inverse, multilevel preconditioning, multistep successive preconditioning.

1. Introduction. Large sparse unstructured matrices arise from various computer simulation and modeling problems. For example, the discretization of systems of partial differential equations by finite difference, finite element, or finite volume methods leads to large systems of simultaneous linear equations, whose coefficient matrix is sparse. In current industrial and engineering applications, the size of the sparse linear systems of practical interest is between a few thousands to a few millions. The solution computation of such large problems typically consumes a major portion of CPU time of many supercomputers used in large scale simulations.

To be more specific, we consider the solution of linear systems of the form $Ax = b$, where b is the right-hand side vector, x is the unknown vector, and A is a large sparse nonsingular matrix of order n. For solving this class of problems, preconditioned Krylov subspace methods are considered to be one of the most promising candidates [2, 36]. A preconditioned Krylov subspace method consists of a Krylov subspace solver and a preconditioner. It is believed that the quality of the preconditioner influences and in many cases dictates the performance of the preconditioned Krylov subspace solver [30, 52]. As the order of the sparse linear systems of interest continues to grow, parallel iterative solution techniques that can utilize the computing power of multiple processors have to be employed. Although the parallel implementations of most Krylov subspace methods have been studied for years and very good software packages are available [5, 34, 46], the research on robust parallel preconditioners that are suitable for distributed memory architectures is being actively pursued [11, 13, 21, 48].

The incomplete LU (ILU) factorizations have been used as general purpose preconditioners for solving general sparse matrices [28]. Since the ILU preconditioners are based on various Gauss elimination procedures, they are inherently sequential in both the construction and the application phases. The ILU factorizations may be used as localized preconditioners to extract parallelism when domain decomposition methods are used to solve large sparse linear systems [31, 47]. However, the computed preconditioners are approximations to a block Jacobi preconditioner. The convergence rate (performance) of such domain decomposition preconditioners deteriorates as the number of processors increases [45]. For many difficult problems, the localized ILU (block Jacobi) preconditioners are not robust.

Using a multilevel structure, the performance of the ILU preconditioners can be improved. There are several variants of multilevel ILU preconditioners [3, 10, 11, 32, 39, 50, 54]. One class of multilevel preconditioners is based on exploiting the idea of successive (block) independent set orderings, which afford parallelism in both the preconditioner construction and application phases [35, 38, 39, 40, 42, 43, 44].

Sparse approximate inverse (SAI) is another class of preconditioning techniques which can be used for solving large sparse linear systems on parallel systems [6, 7]. Several versions of SAI techniques have been developed [8, 14, 19, 21, 55]. These preconditioners possess high degree of parallelism in the preconditioner application phase and are shown to be effective for certain type of problems. Parallel implementations of SAI preconditioners are available [4, 12, 13, 22, 48]. For difficult problems, the SAI preconditioners may be less robust, compared to the ILU preconditioners. Based on the success achieved by applying the multilevel structure

* kwang0@cs.uky.edu, URL: http://www.csr.uky.edu/~kwang0
†The corresponding author. jzhang@cs.uky.edu, http://www.cs.uky.edu/~jzhang
‡ Laboratory for High Performance Scientific Computing and Computer Simulation, Department of Computer Science, University of Kentucky, Lexington, KY 40506–0046, USA, cshen@cs.uky.edu

to ILU preconditioners, the idea of combining strengths of the multilevel methods and the SAI techniques looks attractive. In fact, some authors have already proposed to improve the robustness of SAI techniques by using multilevel structures or to enhance the parallelism of multilevel preconditioners by using SAI [9, 29, 49, 53]. But none of these studies is done on a distributed memory computer system.

Recently, a multistep successive preconditioning strategy (MSP) was proposed in [48] to compute robust preconditioners based on SAI. MSP computes a sequence of low cost sparse matrices to achieve the effect of a high accuracy preconditioner. The resulting preconditioner has a lower storage cost and is more robust and more efficient than the standard SAI preconditioners.

In this paper, we investigate the use of the MSP strategy to construct a class of multilevel SAI preconditioners. Because of the inherent parallelism provided by MSP, we need not use an independent set ordering. We use forward and backward preconditioning strategy to improve the performance of the multilevel preconditioner. In addition MSP provides a convenient approach to creating approximate Schur complement matrices with different accuracy. We implement a two Schur complement matrix preconditioning strategy to reduce the storage cost of the multilevel preconditioner.

This paper is organized as follows. Section 2 outlines the procedure for constructing a multilevel preconditioner based on MSP. Section 3 discusses some implementation details and strategies to improve the performance of our multilevel preconditioner. Section 4 reports some numerical experiments with the multilevel preconditioners on a distributed memory parallel computer. A brief summary is given in Section 5.

2. Preconditioner Construction. We recount the multistep successive preconditioning (MSP) strategy introduced in [48], and explain the concept of multilevel preconditioning techniques briefly. We then discuss the idea of using MSP in the multilevel structure to construct a multilevel SAI preconditioner. Our aim is to build a hybrid preconditioner with increased robustness and inherent parallelism.

2.1. Multistep successive preconditioning. In order to speed up the convergence rate of the iterative methods, we may transform the original linear system into an equivalent one $MAx = Mb$, where M is a nonsingular matrix of order n. If M is a good approximation to A^{-1} in some sense, M is called a sparse approximate inverse (SAI) of A [6, 7]. Several techniques have been developed to construct SAI preconditioners [8, 7, 14, 17, 21, 55]. Each of them has its own merits and drawbacks. In many cases, the inverse of a sparse matrix may be a dense matrix, a high accuracy SAI preconditioner may have to be a dense matrix. The basic idea behind MSP is to find a multi-matrix form preconditioner and to achieve a high accuracy sparse inverse step by step. In each step we compute an SAI inexpensively and hope to build a high accuracy SAI preconditioner in a few steps. MSP can be applied to almost any existing SAI techniques [48]. The following is an MSP algorithm with a static sparsity pattern based SAI.

ALGORITHM 2.1. Multistep Successive SAI Preconditioning [48].

0. Given the number of steps $l > 0$, and a threshold tolerance ϵ
1. Let $A_1 = A$
2. For $i = 1, \ldots, (l-1)$, Do
3. Sparsify A_i with respect to ϵ
4. Compute an SAI according to the sparsified sparsity pattern of A_i, $M_i \approx A_i^{-1}$
5. Drop small entries of M_i with respect to ϵ
6. Compute $A_{i+1} = M_i A_i$
7. EndDo
8. Sparsify A_l with respect to ϵ
9. Compute an SAI according to the sparsified sparsity pattern of A_l, $M_l \approx A_l^{-1}$
10. Drop small entries of M_l with respect to ϵ
11. $\prod_{i=1}^{l} M_i$ is the desired preconditioner for $Ax = b$

There are a few heuristic strategies to choose the sparsity pattern for an SAI preconditioner. Both static and dynamic sparsity pattern approaches have been investigated [14, 15, 25]. Usually the dynamic sparsity pattern strategies can compute better SAI preconditioners with a given storage cost. But they may be more expensive and more difficult to implement on parallel computers.

The static sparsity pattern strategy is attractive to implement on distributed memory parallel computers [12, 48]. A particularly useful and effective strategy is to use the sparsified pattern of the matrix A (or $A^2, A^3, \cdots$)

to achieve higher accuracy [12]. Here "sparsified" refers to a preprocessing phase in which certain small entries of the matrix are removed before its sparsity pattern is extracted. In order to keep the computed matrix sparse, small size entries in the computed matrix M_i are dropped (postprocessing phase) at each step of MSP. We note here that Algorithm 2.1 is slightly different from the one developed in [48], in which different parameters are used for the preprocessing and postprocessing phases. Since these parameters are usually chosen to be of the same value [48], only one parameter is used in Algorithm 2.1.

Algorithm 2.1 generates a sequence of matrices $M_1, M_2, \cdots, M_l$ inexpensively. They together form an SAI for A, i. e., $M_l M_{l-1} \cdots M_1 \approx A^{-1}$. From the numerical results in [48] we know that in addition to enhanced robustness, MSP outperforms standard SAI in both the computational and storage costs.

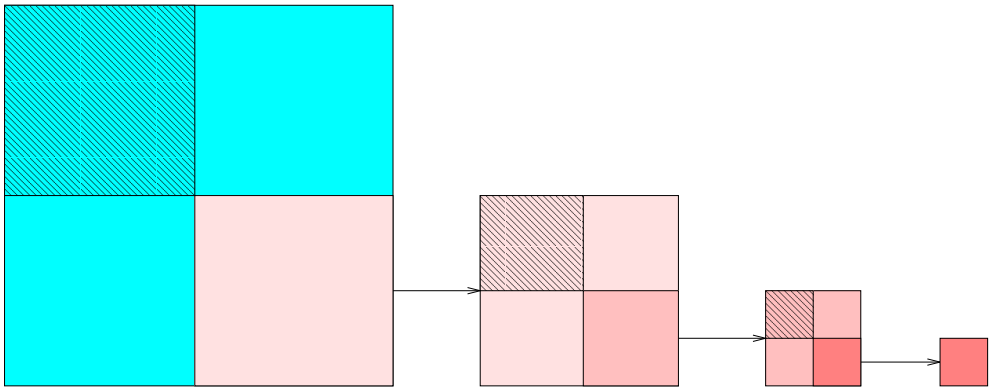


FIG. 2.1. *Recursive matrix structure of a 4 level preconditioner.*

2.2. Multilevel preconditioning. For an illustration purpose, we show in Fig. 2.1 the recursive matrix structure of a 4 level preconditioner. Usually, the construction of a multilevel preconditioner consists of two phases. First, at each level the matrix is permuted into a two by two block form, according to some criterion or ordering strategy,

$$A_\alpha \sim P_\alpha A_\alpha P_\alpha^T = \begin{pmatrix} D_\alpha & F_\alpha \\ E_\alpha & C_\alpha \end{pmatrix}, \tag{2.1}$$

where P_α is the permutation matrix and α is the level reference. For simplicity, we denote both the permuted and the unpermuted matrices by A_α. Second, the matrix is decomposed into a two level structure by a block LU factorization,

$$\begin{pmatrix} D_\alpha & F_\alpha \\ E_\alpha & C_\alpha \end{pmatrix} = \begin{pmatrix} I_\alpha & 0 \\ E_\alpha D_\alpha^{-1} & I_\alpha \end{pmatrix} \begin{pmatrix} D_\alpha & F_\alpha \\ 0 & A_{\alpha+1} \end{pmatrix}, \tag{2.2}$$

where I_α is the generic identity matrix at level α. $A_{\alpha+1} = C_\alpha - E_\alpha D_\alpha^{-1} F_\alpha$ is the Schur complement matrix, which forms the reduced system. The whole process, permuting matrix and performing block LU factorization, can be repeated with respect to $A_{\alpha+1}$ recursively to generate a multilevel structure. The recursion is stopped when the last reduced system $A_{\mathcal{L}}$ is small enough to be solved effectively.

The preconditioner application process consists of a level by level forward elimination, the coarsest level solution, and a level by level backward substitution. Suppose the right hand side vector b and the solution vector x are partitioned according to the permutation in (2.1), we have, at each level,

$$x_\alpha = \begin{pmatrix} x_{\alpha,1} \\ x_{\alpha,2} \end{pmatrix}, \qquad b_\alpha = \begin{pmatrix} b_{\alpha,1} \\ b_{\alpha,2} \end{pmatrix}.$$

The forward elimination is performed by solving a temporary vector y_α, i. e., for $\alpha = 0, 1, \ldots, \mathcal{L}-1$, by solving

$$\begin{pmatrix} I_\alpha & 0 \\ E_\alpha D_\alpha^{-1} & I_\alpha \end{pmatrix} \begin{pmatrix} y_{\alpha,1} \\ y_{\alpha,2} \end{pmatrix} = \begin{pmatrix} b_{\alpha,1} \\ b_{\alpha,2} \end{pmatrix}, \qquad \text{with} \qquad \begin{cases} y_{\alpha,1} &= b_{\alpha,1}, \\ y_{\alpha,2} &= b_{\alpha,2} - E_\alpha D_\alpha^{-1} y_{\alpha,1}. \end{cases}$$

The last reduced system may be solved to a certain accuracy by a preconditioned Krylov subspace iteration to get an approximate solution $x_{\mathcal{L}}$. After that, a backward substitution is performed to obtain the preconditioning solution by solving, for $\alpha = \mathcal{L}-1, \ldots, 1, 0$,

$$\begin{pmatrix} D_\alpha & F_\alpha \\ 0 & A_{\alpha+1} \end{pmatrix} \begin{pmatrix} x_{\alpha,1} \\ x_{\alpha,2} \end{pmatrix} = \begin{pmatrix} y_{\alpha,1} \\ y_{\alpha,2} \end{pmatrix}, \quad \text{with} \quad \begin{cases} x_{\alpha,2} &= A_{\alpha+1}^{-1} y_{\alpha,2}, \\ x_{\alpha,1} &= D_\alpha^{-1}(y_{\alpha,1} - F_\alpha x_{\alpha,2}), \end{cases}$$

where $x_{\alpha,2}$ is actually the coarser level solution.

2.3. Multilevel preconditioner based on MSP. A straightforward way to build a multilevel SAI preconditioner is to compute an SAI matrix M_α for the submatrix D_α, and to use M_α to substitute D_α^{-1} in Eq. (2.2). We have

$$\begin{pmatrix} D_\alpha & F_\alpha \\ E_\alpha & C_\alpha \end{pmatrix} \approx \begin{pmatrix} I_\alpha & 0 \\ E_\alpha M_\alpha & I_\alpha \end{pmatrix} \begin{pmatrix} D_\alpha & F_\alpha \\ 0 & A_{\alpha+1} \end{pmatrix},$$

The approximate Schur complement matrix is computed as $A_{\alpha+1} = C_\alpha - E_\alpha M_\alpha F_\alpha$. Continue doing this for $A_{\alpha+1}$ at the next level, a multilevel preconditioner based on SAI can be constructed. Correspondingly, the forward and backward substitutions in the preconditioner application phase change to

$$\begin{cases} y_{\alpha,1} &= b_{\alpha,1}, \\ y_{\alpha,2} &= b_{\alpha,2} - E_\alpha M_\alpha y_{\alpha,1}, \end{cases} \quad \text{and} \quad \begin{cases} x_{\alpha,2} &= A_{\alpha+1}^{-1} y_{\alpha,2}, \\ x_{\alpha,1} &= M_\alpha(y_{\alpha,1} - F_\alpha x_{\alpha,2}). \end{cases} \tag{2.3}$$

Because M_α is only an approximation to D_α^{-1}, $C_\alpha - E_\alpha M_\alpha F_\alpha$ is not the exact Schur complement matrix, but an approximation of it. The computed value x_α according to (2.3) will deviate from the true value, even if $A_{\alpha+1}^{-1}$ can be computed exactly. The larger the difference between M_α and D_α^{-1}, the more the deviation of x_α will have. Thus we prefer an accurate SAI of D_α during the construction of the multilevel preconditioner.

Through suitable permutation, it is possible to find a D_α with some special structure so that a sparse inverse of D_α can be computed inexpensively and accurately. A (block) independent set strategy is used in [35, 39, 41, 40] for building the multilevel ILU preconditioners, in which D_α consists of small block diagonal matrices. Thus an accurate (I)LU factorization can be applied to these blocks independently. An independent set related strategy to find a well-conditioned D_α is also used in [49] to construct a multilevel factored SAI preconditioner. Unfortunately block independent set algorithms may be difficult to implement on distributed memory parallel computers. Most published parallel multilevel ILU preconditioners are two level implementations [27, 37, 43], truly parallel multilevel implementations have been reported only recently [24, 44].

For SAI based multilevel preconditioners, there is no need to exploit independent set ordering to extract parallelism, although a block diagonal matrix is certainly easy to invert [53]. What we want is to form a well-conditioned D_α. A diagonally dominant matrix is well-conditioned and may be inverted accurately. This suggests us to find a D_α matrix with a good diagonal dominance property so that D_α^{-1} can be computed inexpensively and accurately. In our implementation, at each level we use a diagonal dominance based strategy to force the rows with small size diagonal entries into the next level system and keep the relatively large diagonal entries in the current level. At the next level another well-conditioned subsystem is found by pushing the rows with unfavorable property into its next level system. This diagonal dominance based strategy is more like a divide and conquer strategy. Each time a difficult to solve problem is divided into two parts. One part is easier to solve than the other. We solve the easier part and employ the Schur complement strategy to deal with the other part.

We can improve the approximation of D_α^{-1} by using MSP. At each level, we compute a series of sparse matrices such that

$$M_{\alpha l} M_{\alpha l-1} \cdots M_{\alpha 1} \approx D_\alpha^{-1}, \tag{2.4}$$

where l is the number of steps. The corresponding Schur complement matrix can be formed as

$$C_\alpha - E_\alpha M_{\alpha l} M_{\alpha l-1} \cdots M_{\alpha 1} F_\alpha. \tag{2.5}$$

3. Implementation Details. To solve a sparse linear system on a parallel computer, the coefficient matrix is first partitioned by a graph partitioner and is distributed to different processors (approximately) evenly. Suppose the matrix is distributed to each processor according to a row-wise partitioning [26], each processor holds k rows of the global matrix to form a local matrix.

Matrix permutation. We give a simple diagonal dominance based strategy to find a well-conditioned D_α matrix. This can be accomplished by computing a diagonal dominance measure for each row of the matrix based on the diagonal value and the sum of the absolute nonzero values of the row [50], i. e., $t_i = |a_{ii}|/\sum_{j\in \mathrm{Nz}(i)} |a_{ij}|$. Here $\mathrm{Nz}(i)$ is the index set of the nonzeros of the ith row. If the ith row is a zero row (locally) in a processor, we set $t_i = 0$. Then the rows with the largest diagonal dominance measures are permuted to form the upper block matrices D_α.

Let ϕ be a parameter between 0 and 1, which is referred to as the reduction ratio. We keep the $k \cdot \phi$ rows with the largest diagonal dominance measures at the current level and let $k \cdot (1 - \phi)$ rows go to the next level. When ϕ is close to 1, the reduced system (next level matrix) will be small. We can maintain load balancing by using the same ϕ in each processor.

We should also point out that in our implementation, the number of levels is not an input parameter like in the other multilevel methods, e.g., BILUM [39]. The multilevel setup algorithm builds the multilevel structure automatically, using ϕ as the constraint. One option is to let the construction phase stop when each processor has only 1 unknown. The last reduced system may be easy to solve. But this may generate too many levels.

To improve the performance of the diagonal dominance based permutation, a local pivoting strategy can be used before we compute the diagonal dominance measures. The local pivoting strategy finds the largest entry in each row of the local matrix, and permutes this entry to the main diagonal. So that most of the main diagonal entries in the local matrix will be larger than the offdiagonal entries in the same row. The submatrix D_α after the diagonal dominance based permutation is more diagonally dominant and better conditioned.

Forward and backward preconditioning. When examining the forward and backward steps in (2.3), we find that the operation $\tilde{x}_\alpha = M_\alpha b_\alpha$ appears twice. In exact form, this operation should be $x_\alpha = D_\alpha^{-1} b_\alpha$. So the value $\tilde{x}_\alpha$ is only an approximation of the true value x_α. The more accurately that $\tilde{x}_\alpha$ approximates x_α, the better a preconditioner we have. We can improve the computed value $\tilde{x}_\alpha$ by a preconditioned GMRES iteration on $M_\alpha D_\alpha x_\alpha = M_\alpha b_\alpha$ and using $\tilde{x}_\alpha$ as the initial guess. We call this preconditioning iteration as a forward and backward preconditioning (FBP) iteration.

Because the reduced systems (Schur complement matrices) are not computed exactly, there is no need to perform many FBP iterations to obtain a very accurate value of $\tilde{x}_\alpha$. A few sweeps are sufficient to make the approximate inverse of D_α comparably accurate with respect to other parts of the preconditioning matrix.

Schur complement preconditioning. When using MSP to compute the SAI of a matrix, a larger number of steps will produce a better approximation [48]. The final form of the preconditioner is a multi-matrix form and these matrices are stored individually. The combined storage cost of MSP is not too large if each matrix is sparse. This is one of the advantages of MSP over the standard SAI [48]. When using MSP to generate a multilevel preconditioner, these sparse matrices have to be multiplied out to compute the reduced system $A_{\alpha+1}$ as in (2.5). This may result in a dense Schur complement matrix.

A compromise can be reached in this situation by computing two Schur complement matrices with different accuracy by using different drop tolerances [29]. The more sparse one is used as the coarse level system to generate the coarse level preconditioner, and is discarded after serving that purpose. The more accurate and denser Schur complement matrix is kept as a part of the preconditioning matrix and is used in the preconditioner application phase. In our multilevel MSP preconditioner, we use a similar strategy to control the storage cost. Here the two Schur complement matrices are not computed by using different drop tolerances but by using different steps in MSP.

Suppose that MSP generates a series of matrices as in (2.4). We construct the explicit Schur complement matrix (for the reduced system) by using only the first few steps of (2.4), e.g., only $M_{\alpha 1}$, we have $C_\alpha - E_\alpha M_{\alpha 1} F_\alpha$. Because $M_{\alpha 1}$ is usually very sparse according to [48], this Schur complement matrix may be sparse (at least more sparse than the Schur complement matrix (2.5)) and can be computed inexpensively. In the preconditioning phase, we may use the more accurate Schur complement matrix (2.5) in an implicit form. To further improve the accuracy of the Schur complement solution, we may iterate on the implicit Schur complement matrix (2.5) with the lower level preconditioner. This strategy is called Schur complement preconditioning [51]. During the Schur complement preconditioning phase, we only perform a series of matrix vector products. We can see that if each of these matrices is sparse, the combined storage cost is not too high.

Stored preconditioning matrices. At each level α of the multilevel preconditioner, we should store E_α, F_α, and the computed MSP matrices $M_{\alpha l} M_{\alpha l-1} \cdots M_{\alpha 1}$ for the forward and backward substitutions in the preconditioning process. In addition, the matrix D_α is needed in the FBP iterations. If the Schur complement

preconditioning is implemented, the matrix C_α should also be kept. Therefore, the sparsity ratio, which is the storage cost of the preconditioning matrices divided by the storage cost of the original matrix, is at least 1. Some strategies may reduce the storage cost, e.g., the matrices D_0, E_0, F_0 and C_0 do not need to be stored, they can be recovered by a permutation from the original matrix [51]. In our current prototype implementation, we do not use this strategy. At each Krylov subspace iteration, the permutation to recover these four submatrices may be expensive on distributed memory parallel computers.

4. Experimental Results. We implement our parallel multilevel MSP preconditioner (MMSP) based on the strategies outlined in the previous sections. At each level, we use a diagonal dominance measure based strategy to permute the matrix into a two by two block form. A static sparsity pattern based MSP is used to compute an SAI of D_α. During the preconditioning phase, we perform forward and backward preconditioning (FBP) iterations to improve the performance of MMSP. The last level reduced system is solved by a GMRES iteration preconditioned by MSP. We use the MSP code developed in [48] to build our MMSP code, which is written in C with a few LAPACK routines [1] written in Fortran. The interprocessor communications are handled by MPI [20]. We conduct a few numerical experiments to show the performance of MMSP. We also compare MMSP with MSP to show the improved robustness and efficiency due to the introduction of the multilevel structure.

The computations are carried out on a 32 processor (750 MHz) subcomplex of an HP superdome (supercluster) with distributed memory at the University of Kentucky. Unless otherwise indicated explicitly, four processors are used in our numerical experiments.

For all preconditioning iterations, which include the outer (main) preconditioning iterations, FBP iterations, Schur complement preconditioning iterations, and the coarsest level solver, we use a flexible variant of restarted parallel GMRES (FGMRES) [33, 34].

In all tables containing numerical results, "ϕ" is the reduction ratio; "step" indicates the number of steps used in MSP; "iter" shows the number of outer iterations for the preconditioned FGMRES(50) to reduce the 2-norm residual by 8 orders of magnitude. We also set an upper bound of 2000 for the FGMRES iteration, a symbol "-" in a table indicates lack of convergence; "density" stands for the sparsity ratio; "setup" is the total CPU time in seconds for constructing the preconditioner; "solve" is the total CPU time in seconds for solving the given sparse linear system; "total" is the sum of "setup" and "solve"; "ϵ" is the parameter used in MMSP and MSP to sparsify the computed SAI matrices.

4.1. Test problems. We first introduce the test problems used in our experiments. The right hand sides of all linear systems are constructed by assuming that the solution is a vector of all ones. The initial guess is a zero vector.

Convection-diffusion problem. A three dimensional convection-diffusion problem (defined on a unit cube)

$$u_{xx} + u_{yy} + u_{zz} + 1000\,(p\,u_x + q\,u_y + r\,u_z) = 0 \tag{4.1}$$

is used to generate some large sparse matrices to test the scalability of MMSP. Here the convection coefficients are chosen as $p = x(x-1)(1-3y)(1-2z)$, $q = y(y-1)(1-2z)(1-2x)$, $r = z(z-1)(1-2x)(1-2y)$. The Reynolds number for this problem is 1000. Eq. (4.1) is discretized by using the standard 7 point central difference scheme and the 19 point fourth order compact difference scheme [23]. The resulting matrices are referred to as the 7 point and 19 point matrices respectively.

General sparse matrices. We also use MMSP to solve the sparse matrices listed in Table 4.1.
The BARTHT1A matrix is from a 2D high Reynolds number airfoil problem with turbulence modeling. The WIGTO966 matrix comes from an Euler equation model and was supplied by L. Wigton from Boeing. (Both BARTHT1A and WIGTO966 matrices are available from the corresponding author). The FIDAP matrices are extracted from the test problems provided in the FIDAP package [18]. They arise from coupled finite element discretization of Navier-Stokes equations modeling incompressible fluid flows. The UTM matrices are real nonsymmetric matrices arising from nuclear fusion plasma simulations in a tokamak reactor. The UTM matrices and the FIDAP matrices can be downloaded from the MatrixMarket of the National Institute of Standards and Technology.[1] We remark that, based on our experience, most of these matrices are considered difficult to solve by standard SAI preconditioners.

[1] `http://math.nist.gov/MatrixMarket`.

TABLE 4.1

Information about the general sparse matrices used in the experiments (n is the order of a matrix, nnz is the number of nonzero entries).

matrices	n	nnz	description
BARTHT1A	14075	481125	Navier-Stokes flow at high Reynolds number
FIDAP012	3973	80151	Flow in lid-driven wedge
FIDAP024	2283	48733	Nonsymmetric forward roll coating
FIDAP028	2603	77653	Two merging liquids with one external interior interface
FIDAP031	3909	115299	Dilute species deposition on a tilted heated plate
FIDAP040	7740	456226	3D die-swell (square die $Re = 1$, $Ca = \infty$)
FIDAPM03	2532	50380	Flow past a cylinder in free stream ($Re = 40$)
FIDAPM08	3876	103076	Developing flow, vertical channel (angle = 0, Ra = 1000)
FIDAPM09	4683	95053	Jet impingment cooling
FIDAPM11	22294	623554	3D steady flow, heat exchanger
FIDAPM13	3549	71975	Axisymmetric poppet value
FIDAPM33	2353	23765	Radiation heat transfer in a square cavity
UTM1700A	1700	21313	Nuclear fusion plasma simulations
UTM1700B	1700	21509	Nuclear fusion plasma simulations
UTM3060	3060	42211	Nuclear fusion plasma simulations
WIGTO966	3864	238253	Euler equation model

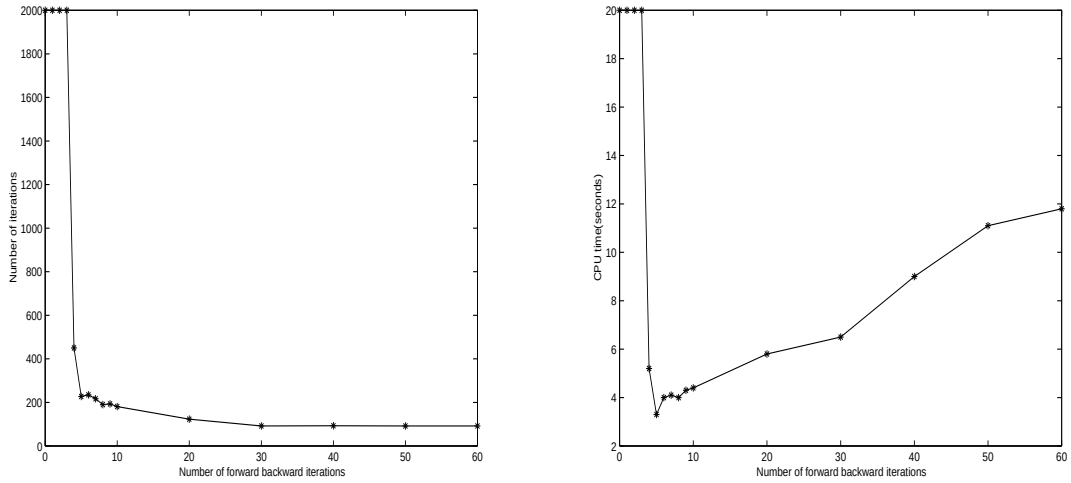


FIG. 4.1. *Convergence behavior of MMSP using different number of FBP iterations for solving the UTM1700B matrix* ($\phi = 0.67, \text{step} = 2, \epsilon = 0.05, \text{density} = 3.48, \text{level} = 7$). *Left: the number of outer iterations versus the number of FBP iterations. Right: the total CPU time versus the number of FBP iterations.*

4.2. Performance of MMSP.

Forward and backward preconditioning. Fig. 4.1 depicts the convergence behavior and the CPU time with respect to the number of FBP iterations for solving the UTM1700B matrix. Here, the FBP iteration performs a few FGMRES(50) iterations to reduce the 2-norm of the relative residual. The number of iterations is an input parameter. From Fig. 4.1, we can see that when we increase the number of FBP iterations from 0 to 5, the number of outer FGMRES iterations decreases rapidly from more than 2000 to around 200. Correspondingly the total CPU time decreases from more than 20 seconds to around 3 seconds. However, we find that doing more than 5 FBP iterations does not result in significant difference in the convergence of MMSP, the number of outer iterations only decreases to around 100. The CPU time actually increases from 3 to 11 seconds. We conclude that the FBP iteration can improve the convergence of MMSP. But a large number of FBP iterations is not cost effective, since the other parts of the preconditioner are not computed exactly. In Fig. 4.1, the best result is obtained with 5 FBP iterations. In the following tests, we fix the number of FBP iterations at 5. We

TABLE 4.2
Solving the BARTHT1A matrix with different MMSP levels ($\phi = 0.67, \text{step} = 3, \epsilon = 0.02$).

level	size	density	iter	setup	solve	total
2	4692	5.13	1843	22.5	417.1	439.6
4	524	6.88	238	18.0	71.8	89.8
6	60	6.86	140	17.3	41.8	59.1
8	8	6.86	137	17.3	40.1	57.4

TABLE 4.3
Solving the WIGTO966 matrix with different values of ϕ ($\text{step} = 2$*).*

ϕ	level	ϵ	density	iter	setup	solve	total
0.67	7	0.05	2.61	-	2.1	-	-
0.50	10	0.05	5.67	184	3.8	9.7	13.5
0.40	14	0.05	9.03	81	7.8	5.2	13.1
0.33	17	0.05	12.14	45	13.3	3.5	16.8
0.25	23	0.05	17.72	33	26.4	3.1	29.5
0.25	23	0.50	10.74	1083	12.6	86.8	99.4

point out that the optimum value of this parameter may be problem dependent, and 5 FBP iterations may not be the best for all problems.

Reduction ratio and number of levels.. The sizes of the current level matrix and the next level (reduced) matrix are controlled by the reduction ratio ϕ. ϕ is an important parameter for deciding the number of levels and influences the performance of MMSP. Here we give some experimental results concerning the reduction ratio and the number of MMSP levels.

The data in Table 4.2 are from solving the BARTHT1A matrix using $\phi = 0.67$. We let the multilevel construction stop when the number of levels reaches a predefined value. The column "size" in the table indicates the size of the last reduced (coarsest) system, which is solved by a preconditioned FGMRES(5) iteration when the 2-norm residual is reduced by a factor of 10^8 or the maximum number of 5 iterations is reached.

It can be seen that a 2 level MMSP, with the last reduced system of 4692 unknowns, needs 1843 iterations and 439.6 seconds to converge. An 8 level MMSP, with the last reduced system of only 8 unknowns, converges in 137 iterations and in 57.4 seconds. In particular, we observe that both the setup time and the solution time are reduced with more levels. The smaller setup time with more levels is due to the fact that a less expensive SAI is constructed for a smaller last level reduced system with more levels.

This experiment indicates that an MMSP with more levels is advantageous for this test problem. In our following experiments, the construction of MMSP stops when there is only one unknown left in each processor. So the number of levels controlled by the reduction ratio ϕ is $-\log_{(1-\phi)} n$, where n is the subproblem size in each processor. For the same problem, different reduction ratio may result in different number of MMSP levels.

Next we use the WIGTO966 matrix to show the influence of ϕ value on the performance of MMSP. The results are given in Table 4.3. We can see that when $\phi = 0.67$, a 7 level MMSP is constructed in 2.1 seconds but does not converge. When ϕ decreases from 0.50 to 0.25, the corresponding number of MMSP levels increases from 10 to 23 and the number of MMSP iterations decreases from 184 to 33, which means that MMSP is more robust when a small ϕ value is used. Unfortunately, a small ϕ value also incurs a large storage cost because more matrices are stored in MMSP.

In the last two rows of Table 4.3, we use the same $\phi = 0.25$ but different ϵ values (0.05 and 0.5). The computed two MMSPs have the same number of levels. The storage cost (density) of the second one is 10.74, compared to 17.72 of the first one. However, the second one needs more iterations (1083) and more solution time (86.8 seconds) to converge. Its performance is worse than that reported in the row 2, where the number of levels is 10, the density is 5.67, and MMSP only needs 184 iterations and 9.7 seconds to converge.

The previous two tests imply that it is not advantageous to set the value of ϕ to be too large or too small. In the following tests, we use $\phi = 0.67$.

In Table 4.4 we show the diagonal dominance and the 2-norm condition number of the matrices A_α and D_α at the first four levels of MMSP for the FIDAP031 matrix. "ddiag" in the table is the ratio of the number of diagonally dominant rows in a given matrix. "cond" is the condition number. We can see that a comparably

TABLE 4.4
The diagonal dominance ratio and the condition number of the matrices at each level of MMSP for the FIDAP031 matrix ($\phi = 0.67, \text{step} = 2, \epsilon = 0.05, \text{density} = 2.48$).

	A_α			D_α		
level	size	ddiag	cond	size	ddiag	cond
1	3909	0.05	$1.0 * 10^6$	2606	0.36	$6.3 * 10^4$
2	1303	0.06	$7.9 * 10^3$	868	0.17	62.9
3	435	0.01	$5.3 * 10^3$	290	0.36	33.59
4	145	0.43	84.79	96	0.81	26.78

TABLE 4.5
Comparison of MMSP with different MSP steps for solving two FIDAP matrices ($\phi = 0.67, \epsilon = 0.01$).

matrices	level	step	density	iter	setup	solve	total
FIDAPM09	8	1	3.40	-	0.9	-	-
	8	2	6.61	-	4.5	-	-
	8	3	9.82	248	11.1	13.8	24.9
FIDAPM33	7	1	3.21	-	0.6	-	-
	7	2	8.14	43	2.0	0.7	2.6
	7	3	14.93	31	5.4	0.7	6.2

well-conditioned and diagonally dominant matrix D_α can be found at each level by the diagonal dominance based strategy. E.g., at the first level, the condition number of the original matrix is $1.0 * 10^6$ and the diagonal dominance ratio is 0.05. After the permutation we can get a matrix D_1 with a condition number $6.3 * 10^4$ and a diagonal dominance ratio 0.36. The matrix D_1 is easier to solve than the matrix A. This is how the multilevel preconditioner works. Instead of preconditioning an ill-conditioned matrix directly, it transforms the matrix into some well-conditioned parts and preconditions these matrix parts level by level.

Number of steps. The data in Table 4.5 show the influence of different MSP steps on the performance of MMSP. For the FIDAPM09 matrix, MMSP does not converges with 1 and 2 MSP steps. It converges with 3 MSP steps in 248 iterations. For the FIDAPM33 matrix, MMSP converges with 2 and 3 MSP steps, but fails in the 1 MSP step case. Just as we expected, a larger number of MSP steps builds a more robust MMSP preconditioner.

Schur complement preconditioning. In Table 4.5, we see that the storage cost of MMSP with 3 MSP steps is large and the implementation may be impractical in large scale applications. The Schur complement preconditioning strategy may alleviate this problem to some extent [51]. We rerun the two test problems in Table 4.5 using the two Schur complement matrix strategy. The two Schur complement matrix strategy is only implemented at the first level. Here we use the FIDAPM09 matrix as an example to explain how the strategy works. In the setup phase, a 3 step MSP is used to form the SAI of D_1, i. e., $M_3M_2M_1 \approx D_1^{-1}$. Then the explicit Schur complement matrix $C_1 - E_1M_1F_1$ is computed as the next level matrix. In the preconditioning phase, we iterate on the implicit Schur complement matrix $C_1 - E_1M_3M_2M_1F_1$ by FGMRES(50) preconditioned by the lower level part of MMSP constructed from $C_1 - E_1M_1F_1$. The test results are shown in Table 4.6, where "step" is the number of MSP steps in the implicit Schur complement matrix. We only allow at most 50 Schur complement preconditioning iterations.

From Tables 4.5 and 4.6 we can see that the two Schur complement matrix strategy reduces the sparsity ratio of MMSP for solving the FIDAPM09 matrix from 9.82 to 4.79. For solving the FIDAPM33 matrix, the sparsity ratio of the 2 MSP step case is reduced from 8.14 to 4.51 and that of the 3 MSP step case is reduced from 14.93 to 6.65. In addition, the setup (construction) time is also reduced to some extent with the two Schur complement matrix strategy. We consider the two Schur complement matrix strategy as an effective way to reduce the memory cost of MMSP. However, the solution time increases because the Schur complement preconditioning strategy utilizes a lot of matrix vector products in the preconditioning phase. We provide the Schur complement preconditioning strategy as an option in our MMSP code in case we have to use a large number of MSP steps for some difficult problems and if the memory cost is more critical than the CPU time.

4.3. Comparison of MSP and MMSP. In Table 4.7, we compare MSP and MMSP for solving a few sparse matrices. For MSP, we adjust the parameter ϵ and the number of steps and try to give the best

TABLE 4.6
Results of the two Schur complement matrix strategy, compared to Table 4.5.

matrices	level	step	density	iter	setup	solve	total
FIDAPM09	8	3	4.79	94	2.5	66.6	69.1
FIDAPM33	7	2	4.51	19	0.7	9.1	9.9
	7	3	6.65	15	1.8	7.7	9.5

performance results for solving these matrices. For MMSP we fix $\epsilon = 0.05$ and step $= 2$. The number in the parentheses of MSP is the number of steps, and the number in the parentheses of MMSP is the number of MMSP levels.

TABLE 4.7
Comparison of MSP and MMSP for solving a few sparse matrices.

matrices	preconditioner	ϵ	density	iter	setup	solve	total
FIDAP024	MSP(3)	0.01	4.87	188	14.4	1.8	16.3
	MMSP(7)	0.05	3.05	39	0.8	0.6	1.4
FIDAPM08	MSP(3)	0.01	3.28	729	48.3	3.4	51.7
	MMSP(8)	0.05	3.02	192	1.1	4.5	5.6
FIDAP012	MSP(-)	-	-	-	-	-	
	MMSP(8)	0.05	3.38	57	1.1	1.2	2.3
FIDAP040	MSP(-)	-	-	-	-	-	
	MMSP(8)	0.05	3.46	39	4.3	4.0	8.3
FIDAPM03	MSP(-)	-	-	-	-	-	-
	MMSP(7)	0.05	3.35	62	0.8	1.0	1.8
FIDAPM11	MSP(-)	-	-	-	-	-	-
	MMSP(9)	0.05	6.81	200	16.3	85.1	101.4
FIDAPM13	MSP(-)	-	-	-	-	-	-
	MMSP(8)	0.05	3.58	86	1.1	1.7	2.8
UTM1700A	MSP(-)	-	-	-	-	-	-
	MMSP(7)	0.05	3.45	145	0.7	1.9	2.6
UTM3060	MSP(-)	-	-	-	-	-	-
	MMSP(8)	0.05	4.02	474	1.0	7.9	8.9

Only 2 of the 9 tested matrices can be solved by MSP. The MMSP can solve these two matrices with smaller sparsity ratios and only 10 percent of the CPU time. In addition, MSP fails to solve the other 7 matrices, which can be solved by MMSP effectively.

Fig. 4.2 shows the convergence behavior of the first 100 MMSP and MSP iterations for solving the FIDAP028 matrix. We can see that MSP reduces the relative residual norm by almost 8 orders of magnitude in 100 iterations. But MMSP reduces the relative residual norm by almost 16 orders of magnitude. From the results of Table 4.7 and Fig. 4.2, we conclude that MMSP is more efficient and more robust than MSP.

4.4. Scalability tests. The main computational costs in MMSP are the matrix-matrix product and matrix-vector product operations. These operations can be performed in parallel efficiently on most distributed memory parallel architectures.

We use the 3D convection-diffusion problem (4.1) to test the implementation scalability of MMSP. The results in Fig. 4.3 are from solving a 7-point matrix with $n = 100^3$ and $nnz = 6940000$ using different number of processors. Due to the local memory limitation of our parallel computer, we can only run the test with at least 4 processors. For easy visualization, we set the speedup in the 4 processor case to be 4. From Fig. 4.3 we can see that MMSP scales well. In particular, we point out that the convergence behavior of MMSP is different from that of MSP. We know that the number of MSP iterations is not influenced by the number of processors when the problem size is fixed [47, 48]. The number of MMSP iterations is affected by the number of processors. This is because the permutation of the matrix at each level depends on the ordering of the unknowns. Different number of processors results in different ordering of the unknowns for the same problem. As it is well known, the performance of (ILU type) preconditioners is affected by the matrix ordering [16]. Fortunately, the number of MMSP iterations does not seem to be strongly influenced by the number of processors.

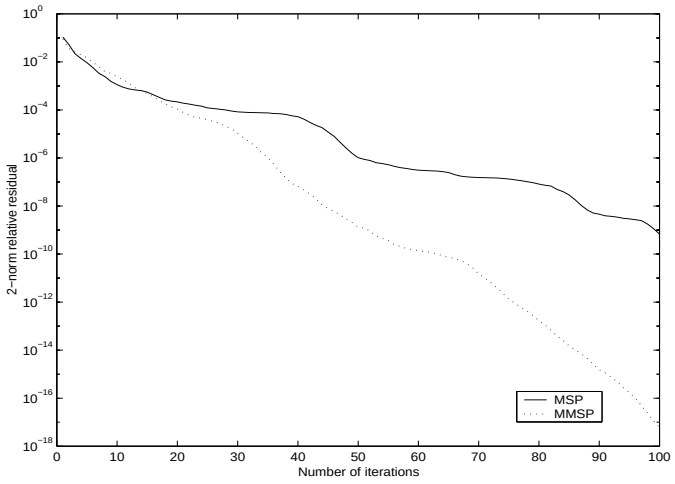


FIG. 4.2. *Convergence behavior of MMSP and MSP for solving the FIDAP028 matrix in* 100 *iterations (MMSP:* density $= 2.83, \epsilon = 0.05, \text{level} = 7, \text{step} = 2$*; MSP:* density $= 5.34, \text{step} = 3, \epsilon = 0.005$*).*

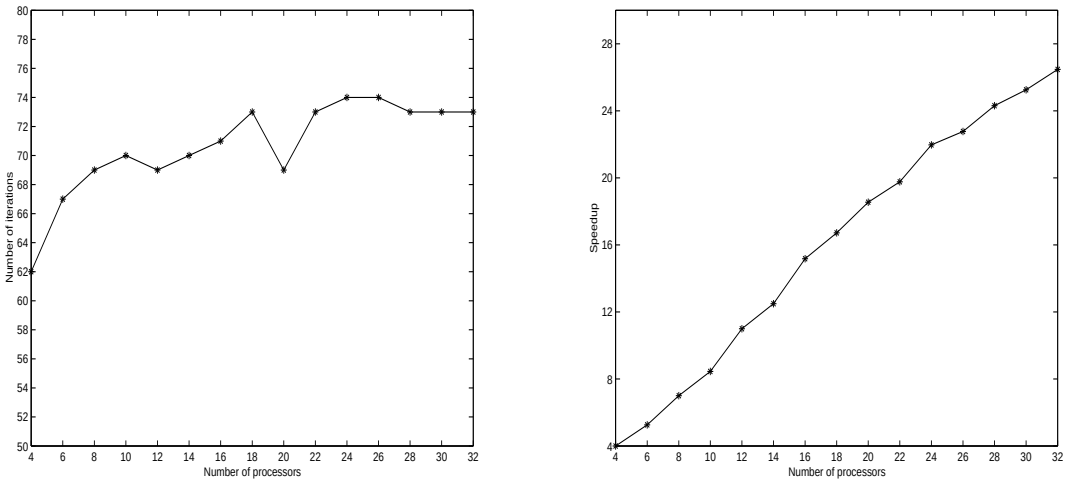


FIG. 4.3. *Scalability test of MMSP when solving a* 7 *point matrix with* $n = 100^3$*,* $nnz = 6940000$ *(*$\epsilon = 0.1, \text{step} = 2, \text{level} = 10, \text{density} = 2.03$*). Left: the number of MMSP iterations versus the number of processors. Right: the speedup of MMSP as a function of the number of processors.*

In Fig. 4.4, the scaled scalability of MMSP is tested by solving a series of 19-point matrices. We try to keep the number of unknowns in each processor to be approximately 25^3. When we change the number of processors, the problem size increases at the same time. To be comparable, we also give the scaled scalability of MSP in the same figure. The parameters used are step $= 1, \text{level} = 10, \epsilon = 0.1$ for MMSP, and step $= 2, \epsilon = 0.05$ for MSP. From Fig. 4.4, We find that MMSP shows better scaled scalability than MSP for this test problem. The behavior of MMSP are more stable than that of MSP.

5. Summaries. We have developed a class of parallel multilevel sparse approximate inverse (SAI) preconditioners based on MSP for solving general sparse matrices. A prototype implementation is tested to show the robustness and computational efficiency of this class of multilevel preconditioners.

From the numerical results presented, we can see that the forward and backward preconditioning (FBP) iteration is an effective strategy for enhancing the performance of MMSP. A few FBP iterations improve the convergence of MMSP. A suitable number of FBP iterations makes MMSP converge fast.

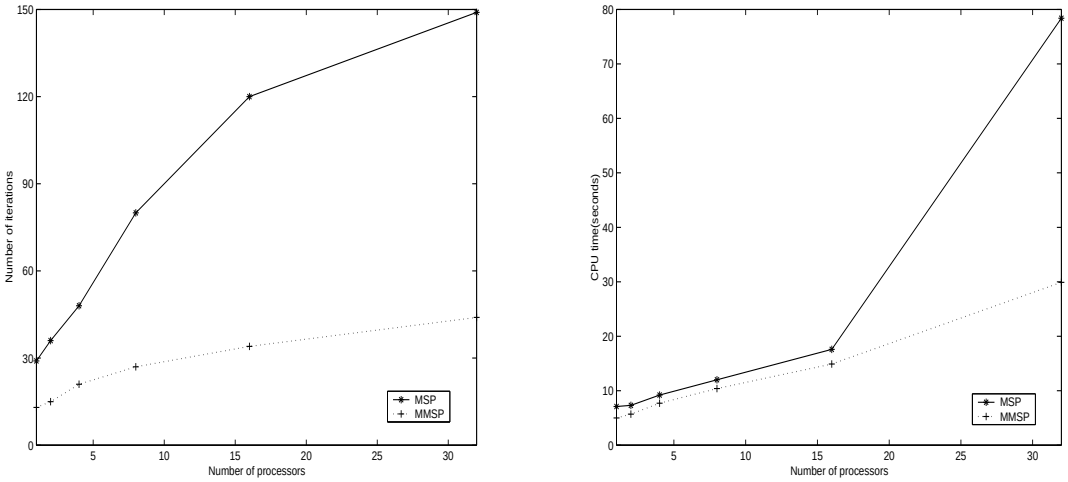


FIG. 4.4. *Scaled scalability test of MMSP and MSP for solving a series of* 19 *point matrices with* $n \approx 25^3$ *in each processor. Left: the number of iterations versus the number of processors. Right: the total CPU time versus the number of processors.*

The number of MMSP levels influences the convergence and storage cost of the preconditioner. A large number of levels results in a fast MMSP preconditioner with a high storage cost. A small number of levels results in an inexpensive preconditioner with a low storage cost. The same statement is valid with respect to the number of MSP steps used at each level of MMSP. We can use a two Schur complement matrix strategy to reduce the storage cost.

Compared with MSP, MMSP is more robust and costs less to construct. The scalability of MMSP seems to be good. But the convergence of MMSP may be affected by the number of processors employed, due to the local matrix reordering implemented to enhance the factorization stability.

Acknowledgements. Kai Wang's research work was funded by the U. S. National Science Foundation under grants CCR-9902022 and ACI-0202934. Jun Zhang's research work was supported in part by the U.S. National Science Foundation under grants CCR-9902022, CCR-9988165, CCR-0092532, and ACI-0202934, by the U. S. Department of Energy Office of Science under grant DE-FG02-02ER45961, by the Japanese Research Organization for Information Science & Technology, and by the University of Kentucky Research Committee. Chi Shen's research work was funded by the U.S. National Science Foundation under grants CCR-9902022 and CCR-0092532.

REFERENCES

[1] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND D. SORENSEN, *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 2 edition, 1995.
[2] O. AXELSSON, *Iterative Solution Methods*. Cambridge Univ. Press, Cambridge, 1994.
[3] R. E. BANK AND C. WAGNER, Multilevel ILU decomposition. *Numer. Math.*, 82(4):543–576, 1999.
[4] S. T. BARNARD, L. M. BERNARDO, AND H. D. SIMON, An MPI implementation of the SPAI preconditioner on the T3E. *Int. J. High Perf. Comput. Appl.*, 13:107–128, 1999.
[5] R. BARRETT, M. BERRY, T. F. CHAN, J. DEMMEL, J. DONATO, J. DONGARRA, V. EIJKHOUT, R. POZO, C. ROMINE, AND H. VAN DER VORST, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Publications, Philadelphia, PA, 1993.
[6] M. W. BENSON AND P. O. FREDERICKSON, Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems. *Utilitas Math.*, 22:127–140, 1982.
[7] M. W. BENSON, J. KRETTMANN, AND M. WRIGHT, Parallel algorithms for the solution of certain large sparse linear systems. *Int. J. Comput. Math.*, 16:245–260, 1984.
[8] M. BENZI AND M. TUMA, A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 19(3):968–994, 1998.
[9] M. BOLLHÖFER AND M. MEHRMANN, Algebraic multilevel methods and sparse approximate inverses. *SIAM J. Matrix Anal. Appl.*, 24(1):191–218, 2002.

[10] E. F. F. Botta and F. W. Wubs, Matrix renumbering ILU: an effective algebraic multilevel ILU preconditioner for sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):1007–1026, 1999.
[11] T. F. Chan and V. Eijkhout, ParPre: a parallel preconditioners package reference manual for version 2.0.17. Technical Report CAM 97-24, Department of Mathematics, UCLA, Los Angeles, CA, 1997.
[12] E. Chow, A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM J. Sci. Comput.*, 21(5):1804–1822, 2000.
[13] E. Chow, Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *Int. J. High Perf. Comput. Appl.*, 15:56–74, 2001.
[14] E. Chow and Y. Saad, Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Sci. Comput.*, 19(3):995–1023, 1998.
[15] J. D. F. Cosgrove, J. C. Diaz, and A. Griewank, Approximate inverse preconditionings for sparse linear systems. *Int. J. Comput. Math.*, 44:91–110, 1992.
[16] I. S. Duff and G. A. Meurant, The effect of reordering on preconditioned conjugate gradients. *BIT*, 29:635–657, 1989.
[17] A. C. N. van Duin, Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices. *SIAM J. Matrix Anal. Appl.*, 20:987–1006, 1999.
[18] M. Engelman, FIDAP: Examples Manual, Revision 6.0. Technical report, Fluid Dynamics International, Evanston, IL, 1991.
[19] N. I. M. Gould and J. A. Scott, Sparse approximate-inverse preconditioners using norm-minimization techniques. *SIAM J. Sci. Comput.*, 19(2):605–625, 1998.
[20] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT, Boston, 2 edition, 1999.
[21] M. Grote and T. Huckle, Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18:838–853, 1997.
[22] M. Grote and H. D. Simon, Parallel preconditioning and approximate inverse on the Connection machines. In R. F. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 519–523, Philadelphia, PA, 1993. SIAM.
[23] M. M. Gupta and J. Zhang, High accuracy multigrid solution of the 3D convection-diffusion equation. *Appl. Math. Comput.*, 113(2-3):249–274, 2000.
[24] G. Karypis and V. Kumar, Parallel threshold-based ILU factorization. Technical Report 96-061, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1996.
[25] L. Y. Kolotilina, Explicit preconditioning of systems of linear algebraic equations with dense matrices. *J. Soviet Math.*, 43:2566–2573, 1988.
[26] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*. Benjamin/Cummings Pub. Co., Redwood City, CA, 1994.
[27] Z. Li, Y. Saad, and M. Sosonkina, pARMS: a parallel version of the algebraic recursive multilevel solver. Technical Report UMSI 2002-100, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001.
[28] J. A. Meijerink and H. A. van der Vorst, An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.
[29] G. Meurant, A multilevel AINV preconditioner. *Numer. Alg.*, 29(1-3):107–129, 2002.
[30] N. M. Nachtigal, S. C. Reddy, and L. N. Trefethen, How fast are nonsymmetric matrix iterations? *SIAM Matrix Anal. Appl.*, 13(3):778–795, 1992.
[31] K. Nakajima and H. Okuda, Parallel iterative solvers with localized ILU preconditioning for unstructured grids on workstation clusters. *Int. J. Comput. Fluid Dynamics*, 12:315–322, 1999.
[32] A. Reusken, On the approximate cyclic reduction preconditioner. *SIAM J. Sci. Comput.*, 21(2):565–590, 1999.
[33] Y. Saad, A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Statist. Comput.*, 14(2):461–469, 1993.
[34] Y. Saad, Parallel sparse matrix library (P_SPARSLIB): The iterative solvers module. In *Advances in Numerical Methods for Large Sparse Sets of Linear Equations*, volume Number 10, Matrix Analysis and Parallel Computing, PCG 94, pages 263–276, Yokohama, Japan, 1994. Keio University.
[35] Y. Saad, ILUM: a multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, 17(4):830–847, 1996.
[36] Y. Saad, *Iterative Methods for Sparse Linear Systems*. PWS Publishing, New York, NY, 1996.
[37] Y. Saad and M. Sosonkina, Distributed Schur complement techniques for general sparse linear systems. *SIAM J. Sci. Comput.*, 21(4):1337–1356, 1999.
[38] Y. Saad and B. Suchomel, ARMS: an algebraic recursive multilevel solver for general sparse linear systems. *Numer. Linear Alg. Appl.*, 9(5):359–378, 2002.
[39] Y. Saad and J. Zhang, BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 20(6):2103–2121, 1999.
[40] Y. Saad and J. Zhang, BILUTM: a domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM J. Matrix Anal. Appl.*, 21(1):279–299, 1999.
[41] Y. Saad and J. Zhang, Diagonal threshold techniques in robust multi-level ILU preconditioners for general sparse linear systems. *Numer. Linear Algebra Appl.*, 6(4):257–280, 1999.
[42] Y. Saad and J. Zhang, Enhanced multilevel block ILU preconditioning strategies for general sparse linear systems. *J. Comput. Appl. Math.*, 130(1-2):99–118, 2001.
[43] C. Shen and J. Zhang, Parallel two level block ILU preconditioning techniques for solving large sparse linear systems. *Paral. Comput.*, 28(10):1451–1475, 2002.
[44] C. Shen, J. Zhang, and K. Wang, Distributed block independent set algorithms and parallel multilevel ILU preconditioners. Technical Report No. 358-02, Department of Computer Science, University of Kentucky, Lexington, KY, 2002.
[45] B. Smith, P. Bjørstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, NY, 1996.

[46] B. Smith, W. D. Gropp, and L. C. McInnes, PETSc 2.0 user's manual. Technical Report ANL-95/11, Argonne National Laboratory, Argonne, IL, 1995.
[47] K. Wang, S. B. Kim, J. Zhang, K. Nakajima, and H. Okuda, Global and localized parallel preconditioning techniques for large scale solid Earth simulations. *Future Generation Comput. Systems*, 19(4):443–456, 2003.
[48] K. Wang and J. Zhang, MSP: a class of parallel multistep successive sparse approximate inverse preconditioning strategies. *SIAM J. Sci. Comput.*, 24(4):1141–1156, 2003.
[49] K. Wang and J. Zhang, Multigrid treatment and robustness enhancement for factored sparse approximate inverse preconditioning. *Appl. Numer. Math.*, 43(4):483–500, 2002.
[50] J. Zhang, A multilevel dual reordering strategy for robust incomplete LU factorization of indefinite matrices. *SIAM J. Matrix Anal. Appl.*, 22(3):925–947, 2000.
[51] J. Zhang, On preconditioning Schur complement and Schur complement preconditioning. *Electron. Trans. Numer. Anal.*, 10:115–130, 2000.
[52] J. Zhang, Preconditioned Krylov subspace methods for solving nonsymmetric matrices from CFD applications. *Comput. Methods Appl. Mech. Engrg.*, 189(3):825–840, 2000.
[53] J. Zhang, Sparse approximate inverse and multilevel block ILU preconditioning techniques for general sparse matrices. *Appl. Numer. Math.*, 35(1):67–86, 2000.
[54] J. Zhang, A class of multilevel recursive incomplete LU preconditioning techniques. *Korean J. Comput. Appl. Math.*, 8(2):213–234, 2001.
[55] J. Zhang, A sparse approximate inverse technique for parallel preconditioning of general sparse matrices. *Appl. Math. Comput.*, 130(1):63–85, 2002.

Edited by: L. Brugnano.
Received: November 7, 2003.
Accepted: May 28, 2004.

Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 107–108. http://www.scpe.org
ISSN 1895-1767

BOOK REWIES

EDITED BY SHAHRAM RAHIMI

Parallel Scientific Computation: A Structured Approach using BSP and MPI
Rob H. Bisseling
Hardcover: 324 pages
Oxford University Press, USA (May 6, 2004)
Language: English
ISBN: 0198529392

In spite of many efforts, no solid framework exists for developing parallel software that is portable and efficient across various parallel architectures. The lack of such framework is mostly due to the absence of a universal model of parallel computation, which can play a role similar to that which Von Neumanns model plays for the sequential computing, and inhibit the diversity of the existing parallel architecture and parallel programming models.

Bulk Synchronous Parallel (BSP) is a parallel computing model proposed by Valiant in 1989, which provides a useful and elegant theoretical framework for bridging the gap between parallel hardware and software. This model comprises a computer archicture (BSP computer), a class of algorithms (BSP algorithm), and a performance model (BSP cost function). The attraction of BSP model lays in its simplicity. A BSP computer consists of collection of processors, each with private memory, and a communication network. A BSP algorithm consists of a sequence of supersteps. A superstep contains either a number of computation steps or a number of communication steps, followed by global barrier synchronization. A BSP performance cost function is based on four parameters: number of processors (p), processor computing rate (r), the ratio between the computation time and communication time (g), and the synchronization cost (l).

In Parallel Scientific Computation: A Structured Approach using BSP and MPI, Rob Bisseling provides a practical introduction to the area of numerical scientific computation by using BSPlib communication library in parallel algorithm design and parallel programming. Each chapter contains: an abstract; a brief discussion of sequential algorithm included to make the material self-contain; the design and analysis of a parallel algorithm; an annotated program text; illustrative experimental results of an implementation on a particular parallel computer; bibliographic notes; theoretical and practical exercises. The source files of the printed program texts, together with a set of test programs that demonstrate their use, form a package called BSPedupack, which is available at the official home page of the book. Researchers, students, and savvy professionals, schooled in hardware or software, will value Bisseling's self-study approach to parallel scientific programming. After all, this is the first textbook provides a comprehensive overview of the technical aspects of building parallel programs using BSP.

The book opens with an overview of BSP model and BSPlib, which tell you how to get started with writing BSP programs, and how to benchmark your computer as a BSP computer. Chapter 2 on dense LU decomposition presents a regular computation with communication patterns that are common in matrix computations. Chapter 3 on the FFT also treats a regular computation but one with a more complex flow of data. Chapter 4 presents the multiplication of a sparse matrix and dense vector. Appendix C presents MPI programs in the order of the corresponding BSP programs appear in the main text.

The book includes a reasonable amount of real world examples, which support the theoretical aspects of the discussions. It is easy to follow and includes logical and consistent exposition and clear descriptions of basic and advanced techniques.

Being a textbook, it contains various exercises and project assignments at the end of each chapter. However, sample solutions for these exercises are still not available. Perhaps an accompanying CD carrying the sample solutions and tutorials for use in the classroom would have added to the academic value of the book. However, the bibliographic notes given at the ends of each chapter, as well as the references at the end of the book, are quite useful for those interested in exploring the subject of BSP development further.

The book is contemporary, well presented, and balanced between concepts and the technical depth required for developing parallel algorithms. Although the book takes a simple performance view of parallel algorithms design, readers should have some basic knowledge of parallel computing, data structures, and C programming.

Overall, the book is suitable as a textbook for one-term undergraduate or graduate courses, as a self-study book, or as technical training material for professionals.

Ami Marowka,
Department of Software Engineering, Shenkar College of Engineering and Design,
Ramat-Gan, Israel.

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:

- programming environments,
- debugging tools,
- software libraries.

Performance:

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2$_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the PDCP WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.