More on JACE: New Functionalities, New Experiments

Jacques M. Bahi, Stéphane Domas and Kamel Mazouzi

Laboratoire d'informatique de l'université de Franche-Comté (LIFC)
IUT de Belfort-Montbéliard
Rue Engel Gros BP 527 90016 Belfort CEDEX France
bahi,mazouzi,sdomas@iut-bm.univ-fcomte.fr

Abstract

Java is often criticized for its poor performances compared to native codes. Nevertheless, this language provides lots of interesting functionalities to easily implement scientific applications on a widely distributed architecture (i.e. grid). The context of this paper is that of iterative algorithms. In order to increase the efficiency of the code, we suggest to use a special class of algorithms called AIACs (Asynchronous Iterations, Asynchronous Computations). This paper presents new results on our works to combine Java and asynchronism within a programming/execution environment called JACE. New functionalities have been added and interesting comparisons with C/MPI and on the impact of overlap techniques are given.

1 Introduction

These last years, parallel programming gave rise to distributed programming, in the sense of target architectures are more and more collections of distant computing resources. A lot of problems emerge using such architectures: heterogeneity, network bottlenecks, resources allocation, application deployment, secured accesses, etc. Research projects based on the "grid computing" try to address these difficult issues, in order to solve very large problems. For example, Globus [11] is a middleware for the grid that provides a lot of functionalities (resource management, data management, communication, fault detection, portability, ...), successfully used to program huge scientific applications. Project DIET [9] relies on the grid to do meta-computing: choosing the best computing site among a collection to solve a client problem. Unfortunately, these environments are heavy to install and it is hard to obtain good performances unless we use clusters of clusters, linked by high speed networks.

Simply using workstations, lower level environments like MPI are a good and easy solution to construct a virtual parallel machine (i.e. local), but not a distributed one. For example, it implies to have the same version of MPI installed on each machine, to deploy the computing code "by hand".

Keeping only advantages of these tools, we can state that an environment well adapted to solve scientific problems on the grid must:

- build an architecture composed of an unlimited number of distant and heterogeneous machines,
- compile and spawn the computing codes,
- provide communication primitives between machines,
- possibly secure accesses to grid resources.

The main key to reach these goals is portability. Nearly all problems come from the heterogeneity. If we use a Java based environment, portability is insured and a lot of problems disappear. By the way, few environments like MPJ [4] or JMPI [14] propose Java MPI clones. Other projects are based on active objects, like ProActive [8] which is more generic and can be used to solve scientific or classical concurrent problems, to implement applications relying on migration, etc.

Meanwhile, there is still the efficiency problem which is insufficiently addressed or skewed using big clusters and dedicated high speed networks. In fact, the classical computation libraries like ScaLAPACK [7] and PetsC [5] obtain very poor performances on heterogeneous and widely distributed architectures. We can evenly say that it is the common way of programming scientific applications that is not adapted

1-4244-0054-6/06/$20.00 ©2006 IEEE

to the grid. We must think at the opposite and try to adapt algorithms to the grid.

Our approach is based on **AIACs** (*Asynchronous Iterations-Asynchronous Communications*) algorithms. This class completely avoids the synchronization phases which are clearly a bottleneck on a widely distributed architecture. Furthermore, these algorithms are tolerant to communication deadlines and even message loss. Meanwhile, they use a communication semantic that no existing environment provides. For a detailed comparison between synchronous and asynchronous algorithms, one can refer to [1].

In order to reach the four goals given above and to implement AIACs algorithms, we have developed **JACE** (*Java Asynchronous Computing Environment*), a programming/execution environment.

JACE has largely evolved since its beginnings [3] [2] and many functionalities have been added. We have also done new experiments based on real scientific problems in chemistry, and studied the influence of overlapping the data domains.

In this paper, we present the new version of JACE and give comparative results based on an advection-diffusion problem.

In section 2, we do a recall on asynchronism and its exploitation in JACE. In section 3, we give an overview of the new functionalities and their goals. Finally, section 4 presents the new application and its experimental results.

2 JACE basics

2.1 Benefits of asynchronism on the grid

To summarize the study of asynchronous algorithms on the grid, presented in [1], we can say that direct methods using sparse matrices and the message passing paradigm are very hard to implement and may obtain poor performances on an heterogeneous grid. Iterative methods are easier to implement but suffer from the same performance problem when they use a synchronous execution scheme.

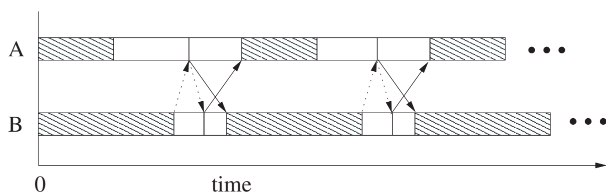

Figure 1. Synchronous iterations

As shown in Figure 1, each computing task must send and receive dependencies data (dotted arrows) at the end of each iteration (hashed boxes), and to retrieve the global convergence state (plain arrows), **before** going on to the next iteration. This implies lots of idle times (white boxes), especially when the machines are heterogeneous and widely distributed over Internet.

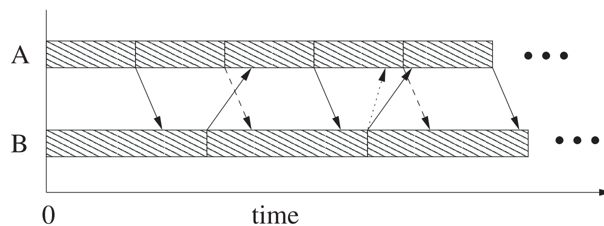

Figure 2. Asynchronous iterations

As shown on Figure 2, an asynchronous execution removes all these idle times by using non-blocking communications and convergence detection. Each task begins a new iteration with "old data" if no messages (dependences and convergence state) have been received. Obviously, using this technique on any iterative algorithm may lead to an endless execution, with no convergence. Meanwhile, the convergence may be ensured by checking some mathematical properties on the iteration functions. Fortunately these properties are satisfied for a large class of scientific problems such as those described by linear systems involving M-matrices or those modelized by partial differential equations and discretized by the finite difference method (e.g. [6]).

Since asynchronism tends to increase the number of iterations to reach convergence, it is useful to increase the convergence rate by always consuming the last data received. On Figure 2, A computes nearly two times faster than B. During iterations 2 and 3, B receives two dependencies messages from A. Convergence may be accelerated if the second message (plain arrow) is consumed by B and the first (dashed arrow) directly crushed.

To summarize, an asynchronous execution relies on a special communication semantic. Within a programming/execution environment, it implies that:

- computations and communications are in different processes,
- messages must be time stamped,
- communication buffers must allow direct crushing of messages.

Unfortunately, no environment provides such functionalities as internal mechanisms. Meanwhile, it is sometimes possible to implement them "by hand" but it is nor easy nor efficient. For example, [3] presents the drawbacks using classical MPI to manage asynchronous computations.

2.2 JACE overview

The idea that leads to the development of JACE was to design a grid-enabled programming and execution environment, allowing a simple and efficient implementation of asynchronous algorithms. In order to do experimental comparisons and to have a more generic environment, JACE should also provide primitives to implement synchronous algorithms and a simple mechanism to change from a synchronous to an asynchronous execution. The main features of the first version of JACE are:

- **Widely distributed architecture**

  JACE builds a virtual distributed machine, composed of heterogeneous machines scattered over several distant sites. Each machine launches a Java Virtual Machine, which executes a JACE daemon (as in MPI or PVM).

- **Thread-based computations and communications**

  Unlike MPI, the computation unit is a Java thread, launched within the daemon context. Therefore, a single daemon may execute multiple tasks. There are also communication threads which send and receive messages for every task thread. This division is mandatory to implement non-blocking communications needed for AIACs algorithms.

- **Easy spawning**

  JACE ensures the localization and a transparent access to each machine during the spawning phase. A simple text file describes how many and on which machines tasks are created, executing which Java byte-code.

- **Standard programming interface**

  The JACE programming interface is intentionally similar to that of MPI, but it only offers the basic set of communication, synchronization and information routines.

- **Explicit message-passing over RMI**

  Tasks cooperate by explicit send and reception of data, encapsulated in *Message* objects. Each message is sent and received using the RMI protocol (*Remote Method Invocation*). Since there are communication threads, the computation task is never blocked by a send, despite the fact that RMI calls are blocking.

- **Global convergence detection**

  JACE is designed to control the global convergence transparently. Tasks only compute their local convergence state and call the JACE API to retrieve the global state. The internal mechanisms of the convergence detection depend on the execution mode, synchronous or asynchronous.

As in classical environments like PVM [10], JACE relies on four components: the daemon, the computing task, the spawner and an application programming interface used to communicate between tasks.

2.2.1 Daemon.

The daemon is the core of the environment. It provides communication mechanisms, task localization and execution. It runs on all machines composing the virtual distributed machine. An XML file containing informations on all machines is given as a parameter to start the daemon. With these informations, a daemon can contact all other daemons and build a directory containing, for example, the stubs needed for RMI calls. This directory is also used to localize tasks and thus, to route messages between tasks.

As soon as all daemons are started JACE is ready to execute tasks.

2.2.2 Task.

A JACE application is a set of cooperating sequential tasks. As shown in figure 4, all tasks run as Java threads within the daemon. Thus, multiple tasks may execute within the same daemon and can share the system resources. When a task is spawned, an unique identification number (task ID) is assigned to it.

To write a JACE application, the user simply needs to extend the **Task** base class and to define a *run()* method containing its own code. In fact, the Task class represents the programming interface since it contains all methods to communicate and to retrieve informations on the execution context.

Figure 3 shows a code example of a task such as a programmer must write it. When the task is compiled, the byte-code can be stored on any point of the network, accessible via a valid URL, for example on a

```
public class Compute extends jace.Task {

  int DATA_TAG = 123;
  byte[] tab;

  public void jaceInitTask(){
    jaceInit(0);
    tab=new byte[1024];
  }

  public void run(){
    jaceBarrier();
    if (jaceMyId == 0) {
      tab=(byte[])jaceReceive(1,DATA_TAG);
    }
    else {
      jaceSend(tab,0,DATA_TAG);
    }
  }
};
```

Figure 3. A JACE Program Example

shared disc or a Web server. By using the reflexivity mechanisms and the dynamic class loading, the daemon downloads the byte-code, creates a new thread corresponding to the control flow of this byte-code, calls the initialization method `jaceInitTask()` and finally starts the thread `run()` method.

2.2.3 Spawner.

When all daemons are started, computation begins by launching the user task on the desired nodes. This functionality is supplied by the spawner. It supports only static spawning, that is, tasks cannot be spawned and deleted during program execution. The number of tasks to be created and an URL of the task byte-code must be specified. All tasks execute the same code, following the SPMD model. The task mapping is done by JACE, trying to balance the number of tasks over the number of machines. After termination of the user's task, the daemon returns the results and cleans the current session by calling a Java garbage collection.

2.2.4 Communications.

The API of JACE is very similar to that of MPI. There are primitives dedicated to initialize the environment, to retrieve informations, to synchronize tasks and to communicate between tasks. As said above, only the last ones completely differ from MPI since they use a special semantic and rely on RMI calls.

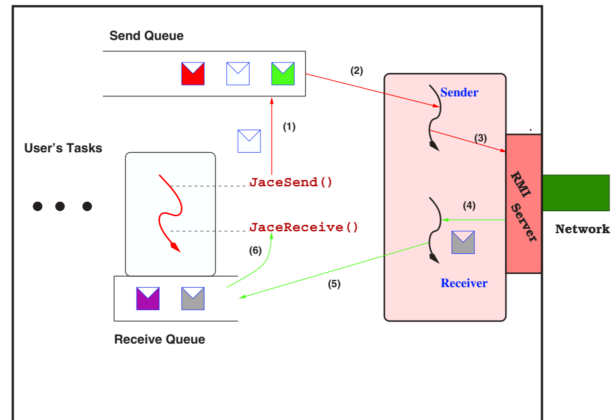

Figure 4. JACE message management.

An example of message management is shown on Figure 4. Whether the execution is asynchronous or not, message queues are not managed in the same way. In the asynchronous case:

1. To send data, the user's task calls the `JaceSend()` method of the API, giving the data, a destination and a tag as parameters. The whole is encapsulated in a Message object, represented by a white box, and put in the send queue. Since there is already a white box (i.e. same source, destination and tag for the two messages) in the queue, the message queued is replaced by the newest.

2. When the send thread (*Sender*) awakes, it checks the send queue.

3. the *Sender* sends the existing messages via a RMI call to the daemon where the destination task is located.

4. When the RMI server is contacted, the message to receive, represented by a grey box, is an argument of the called method `iSendYou()`. The server passes the message to the receiving thread (*Receiver*).

5. The *Receiver* stores the message in the receive queue. Since there is already a grey box in the queue, this last one is replaced by the one just received.

6. To receive data, the user's task calls the `JaceSend()` method, specifying the source and tag of the message.

In the synchronous case, the same steps occur but messages are never crushed in the queues. If the same message exists, the newest is simply stored at the end of the queue. Note that it is the basic behavior of existing environments.

This first version of JACE has given an entire satisfaction to obtain the first comparisons between synchronous and asynchronous executions of real scientific applications. Nevertheless, new developments in the domain of load balancing on networks with a dynamic topology, has bring us to include new functionalities and to define a better organization of the virtual distributed machine.

3 New Functionalities of JACE

3.1 Migration

JACE allows the weak migration (strong migration requires changes in the JVM) of tasks by using the serialization mechanism. When the task arrives on the destination daemon, a thread is created to take the control of the task and to execute its method `run()`. The thread starts its execution with the data of the task right before the migration. As the location of the task changed, a message containing the new location is sent to all daemons before the migration. Thus, other tasks can send their messages to the new location in a transparent way. Since JACE can do synchronous and asynchronous executions, the migration is not completed in the same way in both cases. Indeed, in the asynchronous case, no synchronization is required. The tasks continue their execution whereas a task migrates. In the synchronous case, when the migrant task starts again its execution, the other tasks must also restart to avoid deadlocks produced by explicit synchronizations in the code.

For now, the migration is launched by an explicit call to the JACE API. It is particularly useful to implement a load balancing policy. Supposing the data are regularly saved, it can even be used to restart a task when the hosting machine has crashed.

3.2 Console

The console is a stand-alone process that allows to configure the virtual machine, to receive informations concerning the tasks and to dynamically spawn tasks. It can be launched from any machine, even one not in the virtual machine. A console uses special RMI calls to contact daemons and to interact with the JACE environment. From a JACE console, one can:

- obtain informations on the machines (`conf`);
- add/delete a node to the virtual machine during the execution(`add/del`);
- view the content of task queues (`queue`);
- spawn a task on the environment during the execution (`spawn`);
- execute a set of commands in the form of script (`exec`).

Because it interacts dynamically with daemons, the console allows the execution of MIMD programs[1], using different byte-codes for the same execution. For example, one can execute a distributed application composed of 3 tasks A, B and C. Tasks A and B execute the same computation code and task C deals with the input/output of the application.

The script functionality has been successfully used to simulate mobile and dynamic networks. In conjunction with a little scheduler adding and deleting nodes along time, scripts allowed to check the behavior of load balancing algorithms on a network where nodes may appear and disappear at any time.

3.3 Site-view architecture

In order to minimize the most penalizing communications (those between distant machines), we have enforced an organization by geographical sites.

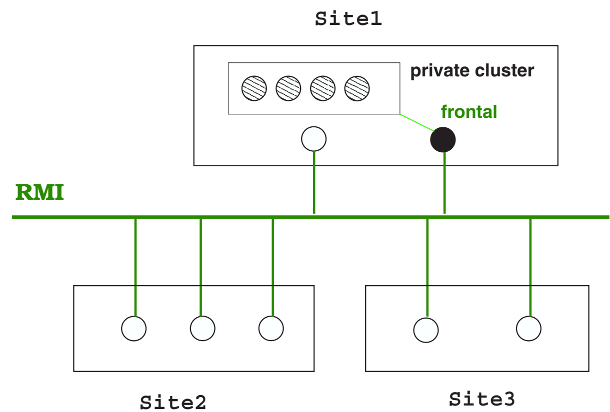

Figure 5. Multisite view of JACE architecture.

It means that the spawner takes care to put maximum number of tasks with following identifiers in the the same site. Indeed, when data are distributed over

[1]Multiple Instruction Multiple Data

a linear set of tasks, most applications imply communications between a task and its neighbors. If neighbor tasks are in the same site, there are no distant communications.

Figure 5 gives an example of a virtual distributed architecture composed of 3 sites. One of them contains a private cluster (hashed circles). In this case, a frontal machine (plain circle) must forward incoming/outgoing messages from/to other sites. In JACE, the daemon launched on the frontal do this automatically. Thus, it is possible to reach machines with private IPs in a transparent way for the user.

4 Experiments

The first experiments using JACE [3] [2] were relatively simple problems. The application presented in this paper modelizes the diffusion (in three dimensions) of chemical components in a little deep water, taking account the reaction between components. The system evolution follows an advection-diffusion model, based on a system of partial differential equations.

For each time step to simulate, we use a combination of multisplitting techniques and Newton's method to solve the problem. To summarize, it consists in computing a Jacobian matrix and to use it in a loop to solve a succession of linear systems. The loop ends when the linear system solution allows to compute a vector close enough (precision is fixed by the user) to the problem solution, at the chosen time step. Thus, the whole algorithm is composed of two nested loops. The outer iterates on time steps and the inner on linear system resolutions, which can be done asynchronously.

The experiments on these problem have three goals:

- to check that asynchronism is much efficient on an heterogeneous grid,
- to study the impact of the overlap techniques,
- to compare the JACE version with a C version, implemented with a multithreaded MPI.

4.1 Asynchronism vs. synchronism

Figure 6 gives a comparison of the execution time (in seconds) of the synchronous and the asynchronous versions. The problem size is given by dimension. Thus, 70 means that a vector of size 70^3 must be found for each time step. The program code is compiled with Java 1.5 (client version). The linear system resolutions are done using GMRES method from the MTJ [12] library. 31 time steps are computed.

32 machines of various power (from 2.4GHz to 3GHz Pentium) have been used. 16 was linked by a gigabit network and the others by a 100Mb network. We have limited (with QoS) the bandwidth between the two sets to simulate distant sites.

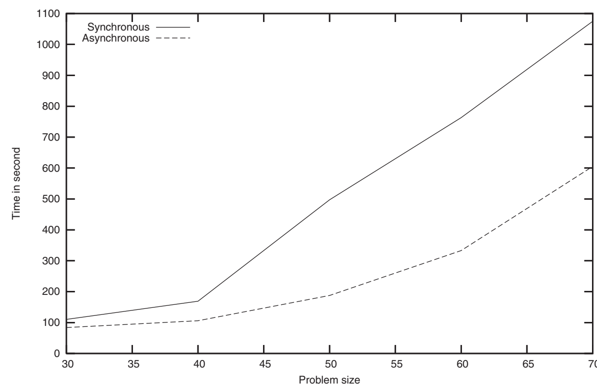

Figure 6. Comparison between synchronous and asynchronous version.

Even if the chosen architecture is quite homogeneous, the asynchronous version is more efficient than the synchronous. It is mainly due to the machine number and the low bandwidth between the two machine sets. By the way, experiments using only the 16 first machines give better results with the synchronous version since global communications and convergence detection are no more penalizing.

4.2 Overlap impact

Overlap is a way to distribute data over computing task such as some solution components are computed by two different tasks.

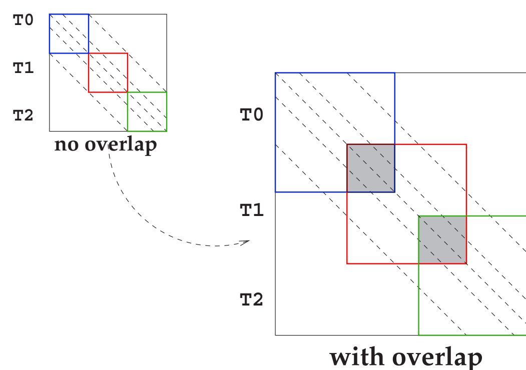

Figure 7. Overlapping distribution.

Figure 7 gives a comparison: on the left, no components are shared, on the right, grey blocks represent components shared by T_0-T_1, and T_1-T_2.

For some problems, which is the case for our one, overlap may accelerate the convergence, as we can see on Figure 8. The execution time is given for different overlap size. A size of 2 indicates that the domain of each task is extended by 2 components along each dimension, which may increase drastically the memory needed to store the data.

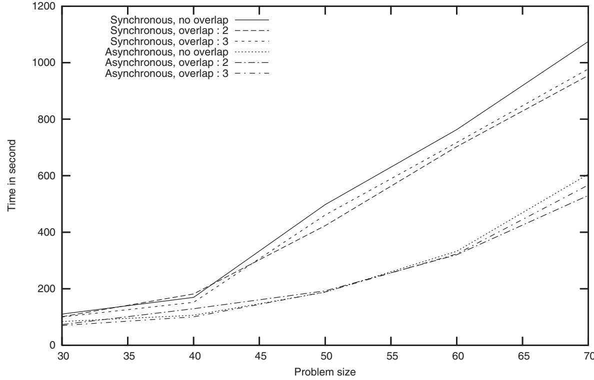

Figure 8. Comparison with and without overlap.

Whether the execution is asynchronous or not, overlap tends to decrease the execution time. But we note that the good size must be chosen since an overlap of 2 is often better than 3. For now, the impact of overlap is unpredictable because it depends on too many parameters, such as number of tasks, problem type and size, etc.

4.3 Raw performance

A frequent question concerns the performance ratio between JACE and other environments. Because it is written in Java and user's tasks too, the raw performances may be poor compared to a C version of the same application. In order to address this question, we have compared our code and a C version using a multithreaded MPI (MPICH/MADELEINE [13] with PM2), on an homogeneous cluster of 16 machines. Obviously, it is the worst case for Java since the communications are not really penalizing in this context and the computation time largely prevails.

Table 1 gives the ratio between the Java and C execution times. Surprisingly, the efficiency of the Java version is good compared to the C version. It is quite common to have a ratio of 10 for scientific applications and we obtain an average of 6. The worst case occurs for small problem sizes because communications are dominant but messages become very small, which is penalizing for RMI calls that have a big latency. As expected, the ratio increases with the problem size since the computations take a more and more important part in the total execution time. The better ratio for asynchronous execution is not really relevant because of the homogeneous context but it should be confirmed in a really heterogeneous and widely distributed context.

size	ratio in synch.	ratio in asynch.
20	16.7	6.7
40	5.8	4.22
60	6.3	5.3
70	6.9	7.8

Table 1. Comparison between JAVA and C execution.

As a conclusion of these results, we note that Java is a realistic way to do high performance computing. Furthermore, JACE completely hides the asynchronism mechanisms which is not the case of the C version. By the way, our application consists in 1500 lines of code and the C version 2200. The excess (nearly the third of the code), represents the lines needed to manage asynchronism, for which a very small part may be reused in another application.

5 Conclusion

JACE combines asynchronism and Java in a single environment. It was designed to implement and to execute efficient computing codes in a grid context. Like MPI, it builds a virtual distributed architecture but in a more hierarchical way in order to minimize the bandwidth use between distant machines. It allows to use machines behind firewall, with private IPs, with is often the case of clusters. JACE API offers primitives to implement message passing codes, but with a special semantic for communications. Indeed, non-blocking calls are mandatory for asynchronous executions and messages may be crushed in the communication buffers. These mechanisms are hidden and should not be implemented by the user, which is the case with MPI and C.

Once again, experiments have proved that asynchronism is perfectly adapted to a grid context. Furthermore, the efficiency may be optimized using an overlapping distribution. Nevertheless, this principle

gives unpredictable results. The comparison of our application with its C counterpart has shown that Java is a good candidate to implement scientific computing codes. In the future, progresses in "just in time" compilation techniques may reduce significantly the ratio.

For now, JACE has been used with a limited number of machines (< 100). In order to increase the scalability of our environment, it would be useful to adapt JACE to a global computing context. That is, a client willing to insert a daemon in the virtual distributed machine would register to a central server. This method allows to control accesses to the architecture but implies several problems:

- computing resources may appear and disappear in an unpredictable way,
- check point mechanisms are needed to store regularly the computed data,
- a resource manager is mandatory, which allocates and checks for available machines.

With this type of architecture, execution of algorithms, mainly the asynchronous ones, may be completely different. Thus, it is necessary to study the behavior of these algorithms on a grid with volatile nodes.

References

[1] J.M. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5):439–461, 2005.

[2] J.M. Bahi, S. Domas, and K. Mazouzi. Combination of java and asynchronism for the grid : a comparative study based on a parallel power method. In *18th IEEE and ACM Int. Conf. on Parallel and Distributed Processing Symposium, IPDPS 2004*, pages 158a, 8 pages, Santa Fe, USA, April 2004. IEEE computer society press.

[3] J.M. Bahi, S. Domas, and K. Mazouzi. Jace : a java environment for distributed asynchronous iterative computations. In *12th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'04*, pages 350–357, Coruna, Spain, February 2004. IEEE computer society press.

[4] Mark Baker and Bryan Carpenter. Mpj: A proposed java message passing api and environment for high performance computing. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 552–559, London, UK, 2000. Springer-Verlag.

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page, 2003. http://www-unix.mcs.anl.gov/petsc/petsc-2/index.html.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs NJ, 1989.

[7] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide.* SIAM, 1997.

[8] D. Caromel and al. ProactivePDC : Java library for parallel, distributed, and concurrent computing. http://www-sop.inria.fr/oasis/ProActive/.

[9] F. Desprez and al. DIET : Distributed Interactive Engineering Toolbox. http://graal.ens-lyon.fr/ diet/.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[11] S. Tuecke I. Foster, C. Kesselman. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal Supercomputer Applications*, 15:200–222, 2001.

[12] B. Jornoh and al. MTJ : Matrix Toolkit for Java. http://mtj.dev.java.net/.

[13] G. Mercier. MPICH-Madeleine III : An MPI Implementation for Heterogeneous Clusters of Clusters. http://dept-info.labri.u-bordeaux.fr/ mercier/mpi.html.

[14] S. Morin, I. Koren, and C. Krishna. Jmpi: Implementing the message passing standard in java. 2002.