Java and asynchronous iterative applications: large scale experiments

Jacques M. Bahi, Raphaël Couturier, David Laiymani, 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
name@iut-bm.univ-fcomte.fr

Abstract

This paper focuses on large scale experiments with Java and asynchronous iterative applications. In those applications, tasks are dependent and the use of distant clusters may be difficult, for example, because of latencies, heterogeneity, and synchronizations. Experiments have been conducted on the Grid'5000 platform using a new version of the Jace environment. We study the behavior of an application (the Poisson problem) with the following experimentation conditions: one and several sites, large number of processors (from 80 to 500), different communication protocols (RMI, sockets and NIO), synchronous and asynchronous model. The results we obtained, demonstrate both the scalability of the Jace environment and its ability to support wide-area deployments and the robustness of asynchronous iterative algorithms in a large scale context.

1 Introduction

Computational grids [7, 13] are now widely used around the world. They connect a large number of resources over multiple distributed organizations and run many kind of applications [11]. Parallel numerical algorithms, executed in a grid-like architecture, require usually several inter-processor synchronizations. These synchronizations are needed to update data and to start the next computation steps. In this case, synchronizations and global communications are an important drawback, often degrading the performances especially when the number of processors increases. This class of algorithms is well-known and unfortunately it has been the main class used in scientific applications so far. To overcome this problem, a solution is to use **AIACs** (*Asynchronous Iteration Asynchronous Communication*) algorithms [3, 6]. This class of algorithm is very suitable in a grid computing context because it suppresses all synchronizations between computation nodes, tolerates the messages loss and enables the overlapping of communications by computations. Interested readers might consult [3] for a precise classification and comparison of parallel iterative algorithms. In this way, several experiments [3] show the relevance of AIAC algorithms in a grid context. These works underline the good adaptability of AIAC algorithms to network and processor heterogeneity. Nevertheless, to the best of our knowledge no large scale experiments exist which study the scalability of this kind of algorithms.

Moreover, many works [14, 9] show how the Java language is a well suited language to develop grid applications. Even if its performances are not comparable to those of the C language for example, its portability makes it an interesting solution for the grid context. In [4] we propose a first version of Jace, a Java-based environment dedicated to AIAC algorithms. We also show how Jace allows to develop efficient Java parallel applications based on the AIAC model. Unfortunately, the tests we conducted, only concerned about 50 nodes.

The aim of this paper is twofold. First it describes the second version of Jace (Jace V2) and shows how the architecture of this new version is scalable. Second it studies the behavior of AIAC algorithms in a large scale context (i.e. with a number of geographically distributed nodes greater than 300). In order to perform our tests we have chosen to implement the Poisson problem. This problem seems to be a good candidate application since it presents an interesting computation/communication ratio well suited for grid platforms. The Jace environment and the Poisson application have been deployed over the Grid'5000 wide-area platform and the following parameters have been studied: the number of processors, the number of distant sites, the communication protocol, the problem size and the synchronization mode (synchronous or asynchronous). The results we obtained demonstrate the scalability of Jace and its ability to support wide-area deployments. They also show the robustness of AIAC algorithms in a large scale context.

This paper is organized as follows. Section 2 presents the different issues induced by large scale deployments of numerical applications. It describes how AIAC algorithms can overcome these difficulties. Section 3 presents the Jace V2 environment and more particularly how this platform is designed to face the scalability issue. In section 4 we present and comment the different experimentation results we obtained with the Poisson application on the Grid'5000 testbed. We underline the good behavior of AIAC algorithms and of the Jace V2 environment. We end in section 5 by some concluding remarks and open works.

2 Motivations

2.1 The drawback of synchronizations

Parallel iterative algorithms can be classified in three main classes depending on how iterations and communications are managed (for more details readers can refer to [2]):

- *Synchronous Iterations - Synchronous Communications (SISC)*. Here, data exchanges are performed at the end of each iteration by global synchronous communications. In this way, all the processors must begin the same iteration at the same time. This leads to idle times on processors when they wait for another processor to be ready to communicate or during the communication itself. Figure 1 illustrates this model. Here the grey blocks represent the computation phases, the white spaces the idle times and the arrows the communications.

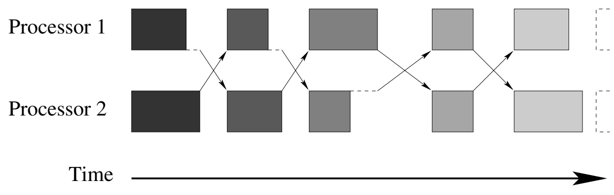

Figure 1. The Synchronous Iterations - Synchronous Communications model

- *Synchronous Iterations - Asynchronous Communications (SIAC)*. This model is similar to the previous one except that data required on another processor are sent without stopping current computations i.e. asynchronously. This technique allows to partially overlap communications by computations. Here, all the nodes may not begin the same iteration at the same time but at a time t processors are either idle or either computing the same iteration. Again, important idle times may occur since communication overlapping is only partial (see figure 2).

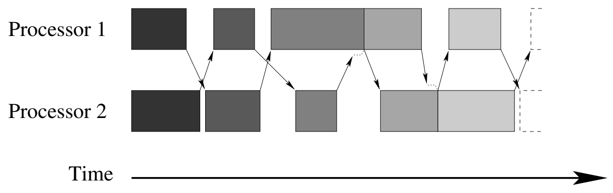

Figure 2. The Synchronous Iterations - Asynchronous Communications model

- *Asynchronous Iterations - Asynchronous Communications (AIAC)*. In this model, local computations do not need to wait for required data. Processors perform their iterations with the data present at that time. In this way, processors can compute different iterations at a given time avoiding useless idle times. Figure 3 illustrates this model.

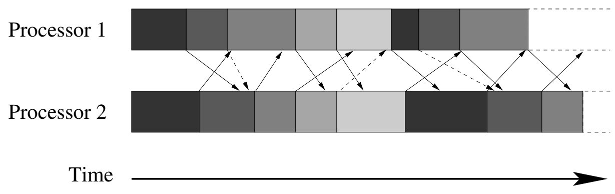

Figure 3. The Asynchronous Iterations - Asynchronous Communications model

For AIAC algorithms, the number of iterations required before the convergence is generally greater than for the two former classes. But by suppressing waiting times (due to synchronizations) on processors, AIAC algorithms are less sensitive to heterogeneity and long distance communications issues. That is why in a grid context, where nodes are heterogeneous and geographically distributed, the overall execution time can be considerably reduced. Several works show the good behavior of AIAC algorithms compared with SISC or SIAC ones [3].

2.2 The AIAC algorithms in a large scale context

It is difficult to define what a large scale context is. Nevertheless our practical experience shows us that using a platform which is "nationally" distributed and composed of more than about 300 computing nodes leads to several new major algorithmic and technical difficulties. It is clear that in such a context each choice must be very carefully taken. From an algorithmic point of view and as explained before, it seems that AIAC algorithms are well suited for a large scale grid context. The crucial point concerns the convergence detection where an important number of messages can be exchanged. In [5] a distributed solution of this problem is proposed. From a technical point of view, the execution environment must address the following issues:

- *Portability*. This characteristic is crucial whatever the size of the deployment platform.
- *Efficient Communication Layer*. Since communications may be large and may occur on long distance links, the communication layer must be carefully optimized and must propose several protocols.
- *Low Resources Consumption*. Since a large number of nodes occurs for a computation, the number of local resources involved (threads, buffers . . .) may grow abnormally and cause important bottlenecks.
- *Efficient Bootstrapping*. As for the previous point, the large number of nodes involved in a computation may induce a costly bootstrapping procedure. The technics used to load the execution environment and the computation code must be carefully designed.

In the following section, we present Jace V2, a scalable Java-based platform for AIAC algorithms. In particular, we underline how Jace V2 is designed to face these issues. Note that in [4] we show why existing environments and libraries, like MPI or Proactive for instance, are not suitable to implement efficient AIAC algorithms.

3 The Jace V2 platform

Jace [4] is a Java programming and executing environment that permits to implement efficient asynchronous algorithms as simply as possible. Jace builds a distributed virtual machine, composed of heterogeneous machines scattered over several distant sites. It proposes a simple programming interface to implement applications using the message passing model. The interface completely hides the mechanisms related to asynchronism, especially the communication manager and the global convergence control. In order to propose a more generic environment, Jace also provides primitives to implement synchronous algorithms and a simple mechanism to swap from one mode to another. Jace relies on three components: *the daemon, the computing task* and *the spawner*.

3.1 The daemon

The daemon is the entity responsible for executing user applications. It is a Java process running on each node taking part in the computation. Figure 4 shows the internal architecture of the daemon which is composed of three layers : *the RMI service, the application layer* and *the communication layer*.

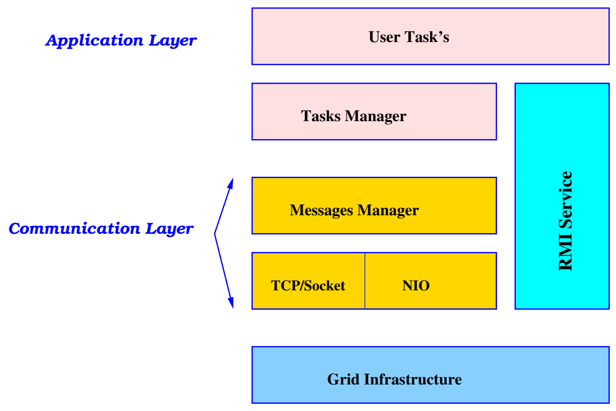

Figure 4. Jace daemon architecture.

The RMI Service. When a daemon is launched, an RMI server is started on it and continuously waiting for remote invocations. This server provides communications between the daemons and the spawner. It is used to manage the Jace environment like for example: initializing the daemons, monitoring and gathering the results.

The Application Layer. This layer provides tasks execution and global convergence detection. A daemon may execute multiple tasks, allowing to reduce distant communications. Jace is designed to control the global convergence process in a transparent way. 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 i.e. synchronous or asynchronous.

The Communication Layer. Communications between tasks are performed using the message/object passing model. Jace uses waiting queues to store incoming/outgoing messages and two threads (`sender` and `receiver`) to deal with communications. According to the kind of algorithm used, synchronous or asynchronous, queues managements are different. For a synchronous execution, all messages sent by a task must be received by the other tasks. Whereas on an asynchronous execution, only the most recent occurrence of a message, with the same source or destination and containing the same type of information is kept, in the queues. The older one, if existing, is deleted.

For scalability issues and to achieve better performances, the communication layer should use an efficient protocol to exchange data between remote tasks. For this reason Jace is based on several protocols : TCP/IP Sockets, NIO (New Input/Output) [1, 10] and RMI. NIO is a Java API (introduced in Java 1.4). It provides new features and improved performances in the areas of buffer management, scalable network and file I/O. The most important distinction between the original I/O library and NIO is how data is packaged and transmitted. Original I/O deals with data in streams, whereas NIO deals with data blocks and consumes a block of data in one step. Furthermore, previously for network applications, users would have had to deal with multiple socket connections by starting a thread for each connection. Inevitably, they would have encountered issues such as operating system limits, deadlocks, or thread safety violations more specially in a large scale context. With NIO, selectors are used to manage multiple simultaneous socket connections on a single thread.

3.2 The Computing Task

As in MPI-like environments, the programmer decomposes the problem to be solved into a set of cooperating sequential tasks. These tasks are executed on the available processors and invoke special routines to send or receive messages. A `task` is the computing unit in Jace, which is executed like a thread rather than a process. Thus, multiple tasks may execute in the same daemon and can share the system resources.

To write a Jace application, the user simply needs to extend the `Task` class and to define a `run()` method containing its program code. The `Task` class may be considered as the programming interface of Jace. It contains a limited set of methods and attributes dedicated to implement asynchronous/synchronous algorithms in a message passing style. To summarize, we can find:

- the non-blocking send/receive,
- the blocking send/receive (for synchronism),
- the global communications: barrier, broadcast, rendezvous
- the convergence control,
- the finalization.

We also point out here that Jace implementation relies on the Java object serialization to transparently send objects rather than raw data.

3.3 The Spawner

The spawner is the entity that effectively starts the user application. After starting daemons on all nodes, computations begin by launching the spawner program with the following parameters:

- the number of tasks to be executed
- the URL of the task byte-code
- the parameters of the application
- the list of target daemons
- the mapping algorithm (round robin, best effort)

Then, the spawner broadcasts this information to all the daemons. As Jace V2 is designed to execute applications on large scale architectures with a large number of nodes, that is achieved by using an efficient broadcast algorithm based on a binomial tree [8]. This algorithm provides better performances compared to a binary tree.

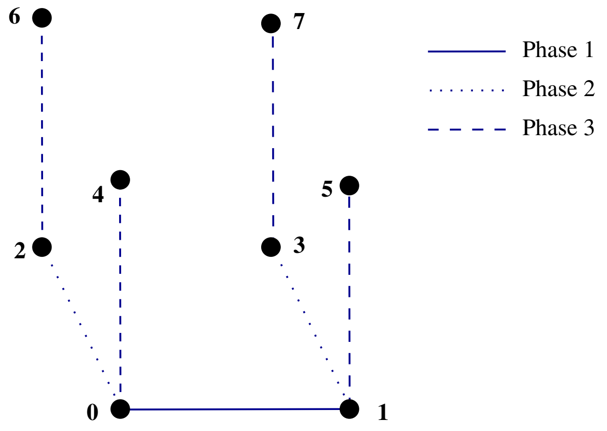

Figure 5. A binomial tree broadcast procedure with 2^3 elements.

Assuming a binomial tree of 2^d nodes and assuming that node 0 needs to send a message to all other nodes. This is done in d parallel phases where at phase k ($1 \leq k \leq d$) node i (with $i < 2^{d-1}$) sends a message to node $i + 2^{k-1}$. In the general case, the spawning procedure is achieved in $log_2(n)$ communication steps on n nodes. Figure 5 presents a binomial tree broadcast procedure with 2^3 nodes.

Now, when a task is spawned, an identification number (task ID) is assigned to it. This number is an integer whose value ranges from 0 to $p - 1$, with p being the global number of tasks in the Jace application. This task mapping is done by Jace and by default uses a round robin algorithm. Another method can be used (called best effort) trying to balance the number of tasks over the set of machines. To illustrate these two policies, let us assume 6 tasks $(0, 1 \dots 5)$ to be mapped on 3 processors. With a round robin algorithm tasks 0 and 3 are mapped on processor 0 and so on. With a best effort algorithm tasks 0 and 1 are mapped on processor 0 and so on. Since communications often take place between consecutive tasks the best effort policy encourages local communications and can be interesting when using multi-processor machines.

4 Experiments

In this section we describe the experiments we have performed in order to test the robustness of both Jace V2 and AIAC algorithms with large scale platforms. For this we have studied several problems that will allow us to measure the adequation of our platform and algorithms with scalability in a grid context with distant sites.

In order to make our experiments, we have chosen the Poisson problem discretized in two dimensions. This is a common problem in physics that models for instance heat problems. This linear elliptic partial differential equations system is defined as

$$-\Delta u = f. \qquad (1)$$

This equation is discretized using a finite difference scheme on a square domain using an uniform Cartesian grid. For more details concerning the finite difference methods, interested readers are invited to consult [12].

After discretization, we use an iterative solver to obtain the solution of the method. In parallel we use the multisplitting method to obtain a coarse grained algorithm suited to grid context [5]. In [5] the authors present the multisplitting method used with a direct solver to solve the linear system. For our experiments we have chosen an iterative one: the conjugate gradient algorithm. Roughly speaking the application consists in iteratively solving a sequential linear system on each processor and then exchanging some dependencies between neighbor processors before computing the next iteration. Considering the 2-dimensions case, each processor has two neighbors.

For our experiments we have used the Grid'5000 platform. Currently, this platform is composed of an average of 1,300 bi-processors that are located in 9 sites in France: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis and Toulouse (see figure 6). Most of those sites have a Gigabit Ethernet Network for local machines. Links between the different sites ranges from 2.5Gbps up to 10Gbps. For more details on the Grid'5000 architecture, interested readers are invited to visit the website: www.grid5000.fr.

In the following experiments, we have chosen different discretization grid sizes, different number of processors (in a local cluster or in different distant clusters), using one or two computing tasks on the same bi-processors and different precision convergence thresholds in order to have relatively short execution times.

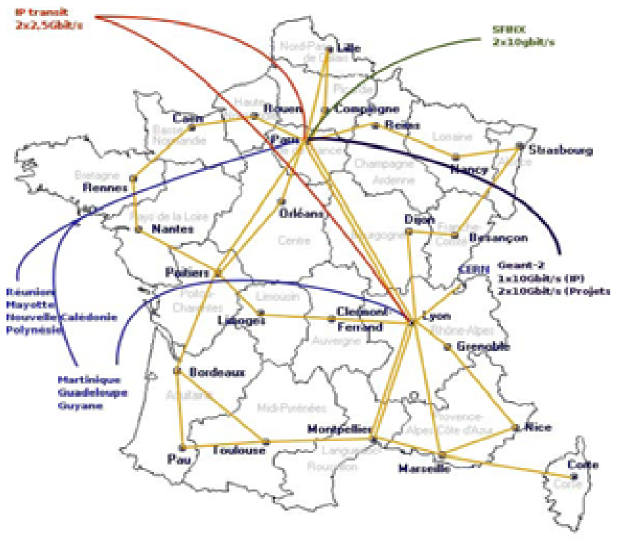

Figure 6. The Grid'5000 testbed

It should be noticed that the size of the matrix to solve is equal to the square size of the discretization grid (so a discretization grid size of 5,000 corresponds to a matrix of size 25,000,000). Moreover results presented here are an average of a series of 5 experiments, that is why with the Grid'5000 architecture with distant sites we cannot conduct longer experiments (reserving a large number of processors is unfortunately not easy). In the following we note $n \times 1$ if we use n bi-processors with one task per node and $n \times 2$ with two tasks per node.

In a first series of experiments we have compared the behavior of Jace V2 with the three different communication protocols (RMI, Socket and NIO) in order to measure their influence with a local cluster and with distant ones. Table 1 shows the execution times of a discretization grid of size 3,000 with 80×1 bi-processors in the Bordeaux cluster (AMD 248 2.2GHz), so 80 cpu. Table 2 shows the execution times of a discretization grid of size 7,000 with 200×1 bi-processors located in four clusters (80 in Bordeaux (AMD 248 2.2GHz), 50 in Lille (20 AMD 248 2.2GHz and 30 AMD 252 2.6GHz), 30 in Lyon (AMD 246 2GHz) and 40 in Rennes (Xeon 2.4GHz)), so 200 cpu.

Both of those tables shows that NIO is the fastest communication protocol and that the asynchronous version is faster than the synchronous one whatever the communication protocol used. As NIO is always the better communication protocol we always use it in the following.

Communication protocol	Execution times in s.	
	Synchronous	Asynchronous
RMI	344	250
Socket	343	230
NIO	340	220

Table 1. Influence of the communication protocol with a $3,000$ discretization grid size and with 80×1 bi-processors in Bordeaux

Communication protocol	Execution times in s.	
	Synchronous	Asynchronous
RMI	976	500
Socket	793	463
NIO	773	406

Table 2. Influence of the communication protocol with a $7,000$ discretization grid size and with 200×1 bi-processors scattered in Bordeaux, Lille, Lyon and Rennes

In a second series of experiments we have compared the behavior of Jace V2 using bi-processors with two tasks or using twice the number of bi-processors with only one task. We have used 40×2 bi-processors, so 80 cpu, in the cluster of Bordeaux (AMD 248 2.2GHz) using a round robin or best effort distribution mode of the tasks and with a discretization grid of size $3,000$. Results are presented in table 3. In the best effort mode, tasks 0 and 1 are on the same machine, tasks 2 and 3 are also on another one and so on, whereas with the round robin distribution, the first machine is in charge of tasks 0 and 40. So the best effort mode encourages local communication and is obviously preferred as shown in Table 3.

Distribution mode	Execution times in s.	
	Synchronous	Asynchronous
Round robin	413	297
Best effort	346	260

Table 3. Influence of the distribution mode with a 3000 discretization grid size and with 40×2 bi-processors in Bordeaux

Note that for the synchronous mode the best effort mode is close to the NIO results in table 1 (for 80×1) . For the asynchronous mode, using bi-processors is quite slower. This can be explained by the fact that on each processor the thread which manages communications is not always active because there are two computing tasks and threads are not so reactive in Java.

In a third series of experiments we compared the results of table 1 with executions on distant clusters with the same number of processors. In Table 4 we report the execution times with a discretization grid of size $3,000$ with 80×1 bi-processors located in four clusters (20 in Bordeaux (AMD 248 2.2GHz), 20 in Lille (10 AMD 248 2.2GHz and 10 AMD 252 2.6GHz), 20 in Lyon (AMD 246 2GHz) and 20 in Rennes (Xeon 2.4GHz)).

Execution times in s.	
Synchronous	Asynchronous
370	250

Table 4. Execution times of a $3,000$ discretization grid size with 80×1 bi-processors scattered in Bordeaux, Lille, Lyon and Rennes

Compared to the results of Table 1, we can see that results are almost similar although we have used distant sites. This confirm the good behavior of AIAC algorithms in a grid context. This can also be explained by the fact that machines of the first experiment (from Bordeaux) are slower than machines used with this experiment.

The last experiment has been conducted to analyze the behavior of Jace V2 with the largest number of processors that we could allocate during our experiment phase. We have used a discretization grid size of $9,000$ with 250×2 bi-processors, so 500 cpu located in four clusters (60 in Lille (40 AMD 248 2.2GHz and 20 AMD 252 2.6GHz), 40 Nancy (AMD 248 2.2GHz), 70 Rennes (40 Xeon 2.4GHz and 30 AMD 246 2GHz) and 80 Sophia (AMD 246 2GHz)).

execution times in s.	
synchronous	asynchronous
641	382

Table 5. Execution times of a $9,000$ discretization grid size with 250×2 bi-processors scattered in Lille, Nancy, Rennes and Sophia

To conclude these experiments we can report several important remarks. First, the asynchronous version of the solver is always faster than the synchronous one. The gain is about 60% on a large scale configuration. Second, NIO is always faster than the other communication protocols, that we have tested (RMI and Socket). Furthermore, and as described in section 3.1, its architecture leads to better scalability. Finally, using bi-processors with 2 tasks with a best effort distribution offers close performances to using twice the number of machines with only one task.

5 Conclusion

In this paper we have described a set of large scale experiments conducted on asynchronous iterative applications and with a Java-based execution environment. We show how this kind of algorithms are well suited for large scale grid contexts compared to synchronous ones. We also show how the new version of the Jace platform successfully faces the scalability issues.

Our current work focuses on developing our experiments and analysis. We are testing Jace V2 with other scientific applications and we are refining our quantitative analysis by measuring different parameters such as bootstrapping time, synchronizations time ...

In order to complete our large scale approach, we are currently working on how AIAC algorithms behave on a peer-to-peer (P2P) architecture. In this perspective, we carry on with the development of Jace. We study its integration on P2P environment such as JXTA and JGroups since these environments already propose standard P2P services such as failure detection, NAT traversing...

References

[1] *New I/O API*. http://java.sun.com/j2se/1.4.2/ docs/guide/nio.

[2] J. Bahi, S. Contassot-Vivier, and R. Couturier. Asynchronism for iterative algorithms in global computing environment. In *16th Int. Symposium on High Performance Computing Systems and Applications*, pages 90–97, Moncton, Canada, 2002. IEEE computer society press.

[3] J. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. *Journal of Supercomputing*, 35(3):227–244, 2006.

[4] J. Bahi, S. Domas, and K. Mazouzi. More on jace: New functionalities, new experiments. In *IPDPS 2006*, pages 231–239. IEEE Computer Society Press, April 2006.

[5] J. M. Bahi and R. Couturier. Parallelization of direct algorithms using multisplitting methods in grid environments. In *IPDPS 2005*, pages 254b, 8 pages. IEEE Computer Society Press, 2005.

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

[7] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[8] A. V. Gerbessiotis. Architecture independent parallel binomial tree option price valuations. *Parallel Computing*, 30(2):301–316, 2004.

[9] F. Huet, D. Caromel, and H. E. Bal. A High Performance Java Middleware with a Real Application. In *Proceedings of the Supercomputing Conference*, Pittsburgh, Pensylvania, USA, Nov. 2004.

[10] B. Pugh and J. Spaccol. MPJava: High Performance Message Passing in Java using Java.nio. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, College Station, Texas, USA, Oct. 2003.

[11] K. Seymour, A. Yarkhan, S. Agrawal, and J. Dongarra. Netsolve: Grid Enabling Scientific Computing Environments. In L. G. editor, editor, *Grid Computing and New Frontiers of High Performance Processing*, volume 14. Elsevier, 2005.

[12] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. Chapman and Hall, 1989.

[13] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation - Practice and Experience*, 17(2-4):323–356, 2005.

[14] R. V. van Nieuwpoort et al. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, 2005.