A Comparative Study of Two Java High Performance Environments for Implementing Parallel Iterative Methods

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

Abstract. This paper aims at studying two Java high performance environments in order to implement parallel iterative methods on Grid infrastructures. We exhibit the important features offered by MPJ Express and Jace V2 to tackle the different issues linked to parallel iterative algorithms. Our study relies on the implementation of a typical iterative application: the multi-splitting method on a large scale grid platform.

1 Introduction

Currently there is a growing interest in developing Grid applications using the Java language. Even if its performance are not comparable to those of the C language for example, many reasons could explain this interest. Among them we can quote the two following ones. First, the ability of Java to handle the heterogeneity between different hardware and operating systems. With regard to the highly heterogeneous nature of the Grid this feature is essential. Second, its ability to support efficient communication. Becker et al shows in [1] how the Java NIO API [2] provides scalable non-blocking I/Os which perform very well.

It also appears that most Grid parallel applications use the message-passing paradigm. In this model, tasks co-operate by exchanging messages and the Message Passing Interface (MPI) is a standard for implementing message-passing applications. In this context, and considering the advantages of Java previously exposed, it is not surprising to see that many research projects aim at developing a message-passing system in Java. These projects can be classified into three classes. The projects of the first class [3] are built upon JNI [4] and use a native MPI implementation as communication layer. These projects provide efficient communication procedures but are not "pure" Java. Projects of the second class [5,6] are based on Java RMI. The RMI API is an elegant high-level "pure" Java solution for remote method invocations of distributed objects but offers little communication performances. In the third class, projects [1,7] use a low-level approach based on Java sockets. This ensures good communication performances and a truly "pure" Java portable environment.

M. Xu et al. (Eds.): APPT 2007, LNCS 4847, pp. 313–321, 2007.

Now, from an application point of view it appears that the well-known parallel iterative numerical algorithms have been the main class used in scientific applications so far. Unfortunately, they usually require several inter-processor communications and synchronizations (to update data and to start the next computation steps for example). The aim of this paper is to provide a comparative study of two "pure" Java message-passing environments for implementing parallel iterative numerical algorithms. We focus on MPJ Express which is developed at the University of Reading [1] and on Jace an environment we are currently developing at the Université de Franche-Comté. We present the features offered by these two environments to tackle the different issues linked to parallel iterative applications. Both of them use the Java NIO socket API and Jace also allows to implement a particular iterative model called *AIACs* (*Asynchronous Iteration-Asynchronous Communication*) algorithms [8,9]. Our study relies on the implementation, on the Grid'5000 testbed [10], of a typical numerical iterative method: the multisplitting method. We show that the communication layer of MPJ offers better performances than Jace. Nevertheless, the ability of Jace V2 to build asynchronous implementations allows it to outperform MPJ synchronous implementations.

This paper is organised as follows. In section 2, we present the motivations and scientific context of our work. In section 3, we describe the MPJE and Jace environments. We particularly describe the new architecture and the new features that we have implemented in the Jace environment (named Jace V2). Section 4 details the experiments we conducted on the Grid'5000 testbed. We present the multisplitting method (both its synchronous and asynchronous implementations) and we analyze the results of our test. We end in section 5 by some concluding remarks and future work.

2 Scientific Context and Motivations

As exposed in the introduction, parallel iterative methods are now widely used in many scientific domains. In the same way and due, in part, to its ability to tackle the heterogeneity problem of the Grid, the Java language is now a good candidate for developping high performance applications. But what are the main features that a Java programming environment must offer to develop efficient numerical iterative applications ?

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 [11]). In the *Synchronous Iterations - Synchronous Communications (SISC)* model data are exchanged at the end of each iteration. All the processors must begin the same iteration at the same time and important idle times on processors are generated. The *Synchronous Iterations - Asynchronous Communications (SIAC)* model can be compared to the previous one except that data required on another processor are sent asynchronously i.e. without stopping current computations. This technique allows to partially overlap communications by computations but unfortunately, the overlapping is only partial and important idle times remain. It is clear that, in a grid computing context, where computational nodes are large, heterogeneous and widely distributed, the idle times generated by synchronizations are very penalizing. One way to overcome this problem is to use the *Asynchronous Iterations - Asynchronous Communications (AIAC)* model. Here, local computations do not need to wait for required data. Processors can then perform

their iterations with the data present at that time. Figure 1 illustrates this model where the grey blocks represent the computation phases, the white spaces the idle times and the arrows the communications. With this algorithmic model, the number of iterations required before the convergence is generally greater than for the two former classes. But, and as detailed in [8], AIAC algorithms can significantly reduce overall execution times by suppressing idle times due to synchronizations especially in a grid computing context.

In this context, it appears that communications and synchronizations are crucial points that any message-passing environment must managed carefully. This communication management must be based on: an efficient point-to-point communication module, an efficient thread management module (for scalability reasons) and the ability to easily implement AIAC algorithms.

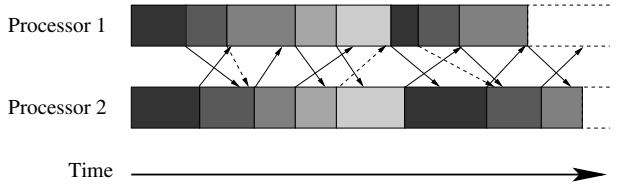

Fig. 1. The Asynchronous Iterations - Asynchronous Communications model

In the remainder, we present MPJ Express and Jace, two Java message-passing environments which aim at offering these features.

3 The MPJ Express and Jace V2 Environments

3.1 MPJ Express

MPJE is structured into a layered design (see figure 2) which allows to use different communication devices such as NIO or native MPI via JNI. In the following we only focus on the NIO device driver. In [1] Baker et al show how point-to-point NIO communications perform well and in [12] Pugh et al point out the good scalability of this package with respect to standard Java sockets.

The communication protocols. The NIO device driver (called `mjdev`) proposes three communication protocols.

- *The Eager-Send protocol*. This protocol is used for small messages (size smaller than 128 Kbytes). Assuming that the receiving part has got an unlimited memory for storing messages the number of control messages is minimized.
- *The Rendezvous protocol*. This protocol is used for large messages (size greater than 128 Kbytes). In this case, control messages are exchanged since their overhead is negligible.
- *The Shared Memory protocol*. This protocol is used when a process is sending a message to itself (inside the same JVM).

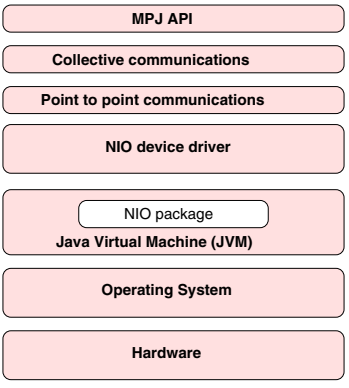

Fig. 2. The MPJ layered architecture

The buffering API. Java sockets are not able to directly access memory in order to read/write basic datatypes. Furthermore, the lack of pointers management could make difficult the use of complex operations such as gather/scatter. To overcome these difficulties, MPJ provides a buffering API in order to pack and unpack data to be sent [13]. Two kinds of buffers can be used: *static* and *dynamic* buffers. Static buffers can only contain primitive datatypes while dynamic buffers can deal with serialized Java objects. When a buffer is created, read and write operations are available to pack and unpack data on it.

The communications primitives. Blocking and non-blocking sending methods are available. These methods are called in the user thread and use the communication protocols previously described. In the same way, blocking and non-blocking receiving methods are available. These methods can be initiated by the user thread (eager-send protocol) or by the NIO selector thread (rendez-vous protocol).

Advantages and drawbacks. The MPJ API is complete and offers a MPI-like style of programming which makes the porting of existing applications easier. Based on a solid buffer management its communication layer is efficient and MPJ appears to be standard for implementing Java message-passing applications. Nevertheless, it appears that MPJ is not well suited for AIAC algorithms. Indeed, even if it is a thread-safe environment, its communication layer architecture is mono-threaded. It is shown in [8] that with this kind of process management it is difficult to implement efficient AIAC algorithms. Another drawback of MPJ is that the application deployment procedure can suffer from a lack of scalability since centralized communication schemes are used.

3.2 Jace V2

Jace [6] 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 management 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 four components: *the daemon, the worker, the computing task* and *the spawner*.

The daemon. The daemon is the core of the Jace system, it is launched on each node taking part in the computation. When a daemon is launched, a remote server is started on it and continuously waits 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 workers, monitoring and gathering the results . . . Daemons are structured as a binomial tree. This hierarchical view of the machines set achieves more efficient spawning and optimizes global communications.

The worker. The worker is the entity responsible for executing user applications. It is a Jace service created for each execution by the daemon. Figure 3 shows the internal architecture of the worker which is composed of two layers:

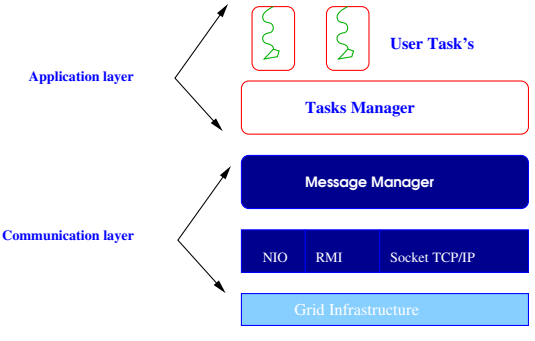

Fig. 3. Jace worker architecture

- **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) [2,12] and RMI (Remote Method Invocation).

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 be executed in the same worker and can share system resources.

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

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 some 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. For scalability reasons, that is achieved by using an efficient broadcast algorithm based on a binomial tree [14]. When a daemon receives the spawner message, it forwards this information to its neighbors and starts a worker to load and execute the user tasks.

Advantages and drawbacks. As presented above Jace is a multi-threaded environment very suitable for AIAC algorithms. For scalability reasons its application deployment procedure is designed in a highly distributed way. Unlike the MPJ one, the Jace communication layer is able to transfer any kind of data objects. This feature provides more flexibility but requires objects serialization which decreases overall performances.

4 Experiments

4.1 The Application: The Multisplitting Method

Consider the n dimensional linear system: $Ax = b$. As exposed in figure5, the A matrix is split into horizontal rectangle parts. Each of these parts is then affected to one processor. With this distribution, a processor is in charge of computing its $XSub$ part by iteratively solving the following subsystem:

$$ASub * XSub = BSub - DepLeft * XLeft - DepRight * XRight$$

This resolution can be processed by a direct solver such as SuperLu. Then the solution $XSub$ must be sent to each processor which depends on it. Now, if rectangle matrices are not disjoint, it appears that some computations will be redundant. This property

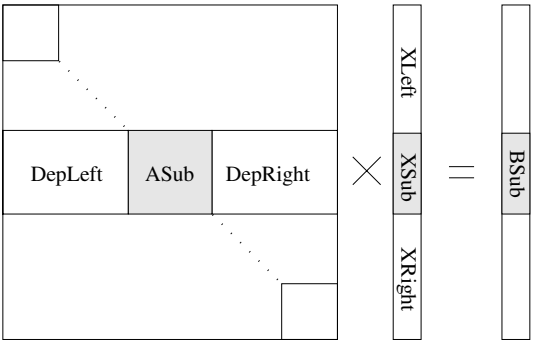

Fig. 4. Decomposition of the system

is called *overlapping* and should be taken into account. In this way, three policies can be applied. Either a processor ignores its components if its neighbors has computed them, or it ignores the neighbors components or it mixes the shared components (by computing their average for example). This parameter has an influence on the convergence speed of the algorithm. Interesting reader can find more details in [15].

For our purpose, this application is interesting for many reasons. First, the convergence of the method for both synchronous and asynchronous mode is shown in [15] (with some restrictions on the A matrix). Second, it appears that the computation/ communication ratio does not let performances be too dependant on the communication layer. In this way, we must be able to evaluate the whole of the target environments (memory management, threads management . . .). Finally, this application is not a "toy" application. It covers several scientific computation areas and its study in different contexts is relevant.

4.2 Experiments Results

The experiments have been conducted on the Grid'5000 platform. This testbed 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. The inter-sites links range from 2.5Gbps up to 10Gbps while most of the sites have a Gigabit Ethernet Network for local machines. For more details on the Grid'5000 architecture, interested readers can refer to: www.grid5000.fr. All the nodes run the Linux Debian distribution with the Sun Java 1.5 Java Virtual Machine.

Figure 5 shows the execution times of the multi-splitting application for different matrix sizes and with 150 nodes of the Grid'5000 testbed. The processors were distributed over 2 sites and the two Jace implementations (synchronous and asynchronous) rely on the Socket communication layer.

It appears that the two synchronous versions (MPJ and Jace) present some equivalent execution times. With respect to the architecture of the two communication layers, these results can be surprising since no object serialization is performed with MPJ while Jace requires this kind of process. We can explain these results by the fact that the target application is coarse grain and so is less sensitive to communication performances. These

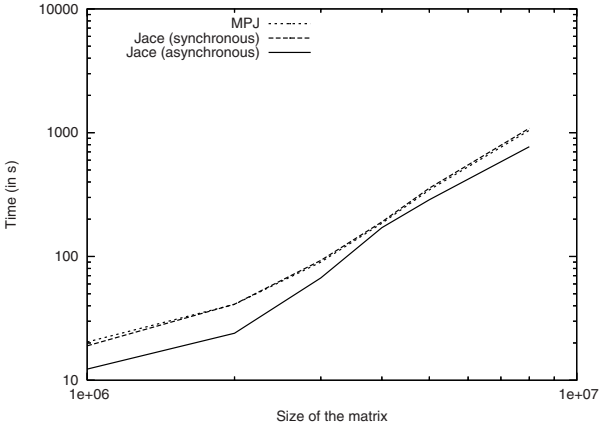

Fig. 5. Time to solve different generated matrices

tests also show the interest of AIAC algorithms since the Jace asynchronous version clearly outperforms the two synchronous ones. The fundamental properties of AIACs algorithms can explained these results. In particular, we can see here the efficiency of asynchronism which allows to obtain a good computations/communications overlapping. This underlines an important feature for Java high performance environments: the ability of easily implemented AIAC algorithms.

5 Concluding Remarks and Future Work

In this paper, we have presented a comparative study of two Java high performance environments (MPJ Express and Jace V2) for implementing parallel iterative methods. Through the implementation of a typical iterative application (the multi-splitting method) we have shown that the communication layer of MPJ Express is efficient. We have also shown that the ability of Jace to support the AIAC model is very relevant.

We are currently working on how AIAC algorithms behave on a peer-to-peer (P2P) architecture. We study the integration of Jace on P2P environments such as JXTA or ProActive [16] since these environments already propose standard P2P services such as failure detection, NAT traversing.

References

1. Baker, M., Carpenter, B., Shafi, A.: MPJ Express: Towards Thread Safe Java HPC. In: Cluster Computing, Barcelona, sept 2006, IEEE Computer Society Press, Los Alamitos (2006)
2. New I/O API, `http://java.sun.com/j2se/1.4.2/docs/guide/nio`
3. Ma, R., Wang, C.-L., Lau, F.: M-javampi: A java-mpi binding with process migration support. In: CCGRID 2002. Proc. of the 2nd IEEE/ACM Int. Symposium on Cluster Computing and the Grid, p. 255. IEEE Computer Society Press, Los Alamitos (2002)
4. JNI, `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`

5. Morin, S., Koren, I., Krishna, C.M.: Jmpi: Implementing the message passing standard in java. In: IPDPS 2002. Proc. of the 16th Int. Parallel and Distributed Processing Symposium, p. 191. IEEE Computer Society Press, Los Alamitos (2002)
6. Bahi, J., Domas, S., Mazouzi, K.: Jace: a java environment for distributed asynchronous iterative computations. In: 12th Euromicro Conference PDP 2004, pp. 350–357. IEEE Computer Society Press, Los Alamitos (2004)
7. MPP, `http://www.uib.no/People/nmabh/mtj/mpp/`
8. Bahi, J., Contassot-Vivier, S., Couturier, R.: Performance comparison of parallel programming environments for implementing AIAC algorithms. Journal of Supercomputing 35(3), 227–244 (2006)
9. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentice Hall, Englewood Cliffs NJ (1989)
10. Grid'5000, `http://www.grid5000.fr`
11. Bahi, J., Contassot-Vivier, S., Couturier, R.: Asynchronism for iterative algorithms in global computing environment. In: 16th Int. Symposium on High Performance Computing Systems and Applications, Moncton, Canada, pp. 90–97. IEEE Computer Society Press, Los Alamitos (2002)
12. Pugh, B., Spaccol, J.: 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 (October 2003)
13. Baker, M., Carpenter, B., Shafi, A.: An Approach to Buffer Management in Java HPC Messaging. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2006. LNCS, vol. 3991, Springer, Heidelberg (2006)
14. Gerbessiotis, A.V.: Architecture independent parallel binomial tree option price valuations. Parallel Computing 30(2), 301–316 (2004)
15. Bahi, J.M., Couturier, R.: Parallelization of direct algorithms using multisplitting methods in grid environments. In: IPDPS 2005, pp. 254b, 8 pages. IEEE Computer Society Press, Los Alamitos (2005)
16. Caromel, D., Di Constanzo, A., Mathieu, C.: Peer-to-peer for computational grids: Mixing clusters and desktop machines. Parallel Computing (2007)