Heterogeneous Parallel Computing using Java and WMPI

Luis M. Silva Paulo Martins João Gabriel Silva

Departamento Engenharia Informática
Universidade de Coimbra - POLO II
Vila Franca - 3030 Coimbra
PORTUGAL
Email: luis@dei.uc.pt

Abstract

In this paper, we describe the implementation of a Java interface for WMPI, a Windows-based implementation of MPI that have been developed by our group. We show some details about the implementation and we present some experimental results that compare the performance of JWMPI, Java WPVM and the C programs counterparts.

In the second part of the paper we describe another tool that is oriented for Web-based computing and we present a solution to integrate WMPI with this tool, by making use of a Java bridge component and our Java bindings for WMPI. This solution allows the execution of meta-applications over a mixed configuration of platforms, execution models and programming languages. The overall system provides an integrated solution to solve the problem of heterogeneity and to unleash the potential of diverse computational resources and programming tools.

1. Introduction

PVM and MPI have become widely accepted in the high-performance community and there are several implementations of PVM and MPI for UNIX Workstations, supercomputers and parallel machines [PVM][MPI]. Both libraries provide a C, C++ and a Fortran interface. In the recent past we saw the release of some implementations of PVM and MPI for personal computers running the Windows operating system: two of them were implemented by our group at the University of Coimbra, namely WPVM [Alves95] and WMPI [Marinho98]. Some other implementations can be found in [PVMWIN32][MPICH/NT].

The libraries WPVM and WMPI already include interfaces for C and Fortran, but with the increasing number of Java programmers it seems quite promising that those communication libraries should also provide a Java interface.

According to [Blundon98] the current number of Java programmers varies between 250.000 and 2.5000.000. The same author predicts that in 2001 year there would be at least 5.000.000 Java programmers and Java will be the dominant language.

Java was developed as a language for the Internet but it is not restricted for the development of Web-pages with animated applets. In fact, the language is also being used for other class of applications, like client/server computing, office applications, embedded systems, programs that use databases and GUI-based interfaces and business applications [Hoff98]. The nice features provided by Java, like portability, robustness and flexibility can also be of much use for the development of scientific and parallel applications. Several projects and several people started also to use Java in parallel and scientific computing, and, as a consequence, a Java Grande Forum was even created in order to establish some consensus among the HPC community for the establishment of Java for parallel and scientific computing [Grande].

In this line it is meaningful to provide Java bindings for the existing standards, like MPI, and other libraries that have been widely used, like PVM. The idea is not to replace all the software written in traditional languages with new Java programs. By the contrary, the access to standard libraries is essential not only for performance reasons, but also for software engineering considerations: it would allow existing Fortran and C code to be reused at virtually no extra cost when writing new applications in Java.

With all these goals in mind we have ported the jPVM interface [jPVM] for the Windows version of PVM (WPVM) and we develop from scratch a similar Java interface for WMPI. JWPVM and JWMPI extend the capabilities of WPVM/WMPI to the new, exciting world of Java. These bindings allow Java applications and existing C/C++ applications to communicate with one another using the PVM/MPI API. In this paper we only describe the implementation of the JWMPI interface.

In this paper we also present an experimental study that uses our Java bindings to merge Web-based computing with Cluster-based computing. Some performance results and interesting conclusions about this study are presented in section 5.

The rest of the paper is organized as follows: the next section presents a brief overview of WPVM and WMPI libraries. Section 3 describes the features of our Java binding for WMPI, while section 4 describes how JWMPI has been integrated with another Java-based tool that is oriented to Web-based computing. Section 5 presents some performance results that were taken our Java bindings and with the integration of WMPI with the other tool. The related work is described in section 6, while section 7 concludes the paper.

2. WPVM and WMPI

WPVM and WMPI are full ports of the standard specifications of PVM and MPI, thereby ensuring that parallel applications developed on top of PVM and MPI can be executed in the MS Windows operating system, as long as they do not use any special feature of the underlying operating system. Both ports can run together in heterogeneous clusters of Windows 95/NT and Unix machines.

WPVM[1] (Windows Parallel Virtual Machine) is an implementation of the PVM message passing environment as defined in release 3.3 the original PVM package from the Oak Ridge National Laboratory. WPVM includes libraries for Borland C++ 4.51, Microsoft VisualC++ 2.0, Watcom 10.5 and Microsoft Fortran PowerStation.

On the other hand, WMPI[2] is an implementation of the Message Passing Interface standard for Microsoft Win32 platforms. It is based on MPICH 1.0.13 with the ch_p4 device from Argonne National Laboratory/Mississippi State University (ANL). WMPI includes libraries for Borland C++ 5.0, Microsoft Visual C++ 4.51 and Microsoft Fortran PowerStation.

3. JWMPI: The Java Binding for WMPI

To develop a Java binding we need a programming interface for the native methods. The JDK release from Sun provides a Java-to-native programming interface, called JNI [JNI]. It allows Java code that runs inside a Java Virtual Machine to interoperate with applications and libraries written in other programming languages, such as C and C++.

3.1 Overview

All JWMPI classes, constants, and methods are declared within the scope of a `wmpi` package. Thus, by importing the `wmpi` package or using the `wmpi.xxx` prefix, we can reference the WMPI Java wrapper. The classes of the `wmpi` package are those corresponding to the objects implicitly used by WMPI. An abbreviated definition of the `wmpi` package and its member classes is as follows:

```
package wmpi;

public class JWMPI;
public class MPI_Status;
public class MPI_Comm;
public class MPI_Group;
public class MPI_Datatype;
public class MPI_Op;
public class MPI_Request;
public class MPI_Errhandler;
```

Figure 1: The `wmpi` package.

[1] WPVM is available at: http://dsg.dei.uc.pt/wpvm/
[2] WMPI is available at: http://dsg.dei.uc.pt/w32mpi/

In the development of this package we tried to provide the user with a MPI-like API. The usual programmer of MPI will not find any difficulty with using JWMPI. To achieve this similarity, all the methods corresponding to WMPI functions are defined in class JWMPI and have exactly the same name and number of parameters. The user just needs to extend the JWMPI class.

In Figure 2 we can see a piece of a WMPI program written in C. In Figure 3 is presented the equivalent program ported to Java using our Java-to-WMPI interface. As we can see both programs look quite very similar.

```
#include <mpi.h>

void main(int argc, char ** argv){

   /* initialize MPI system */
   MPI_Init (&argc, &argv);

   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   ...
}
```

Figure 2: C code example using WMPI.

```
import wmpi.*;

public class MyClass extends JWMPI{

   public static void main(String args[]){

      /* initialize MPI system */
      MPI_Init(args);

      MPI_Comm_rank(MPI_COMM_WORLD,rank);
      MPI_Comm_size(MPI_COMM_WORLD,size);
      ...
   }
}
```

Figure 3: Java code example using the package `wmpi`.

3.2 Opaque objects used by WMPI

Opaque objects are system objects that are accessed through a handle. The user knows the handle to the object but does not know what is inside. Since the MPI does not specify the internal structure of these objects, there is no way to reconstruct them in Java. So, the best thing to do is to keep the handle to the object. To do this, we have implemented one Java class for each opaque object used by WMPI (see Figure 1).

These Java classes hide the handle to the real WMPI opaque objects. The programmer only has to create new instances of these objects and use them as arguments to JWMPI methods. In order to fit into some system that has 64 bits pointers, we use a Java `long` to store the WMPI object handle.

3.3 MPI_Status structure

Unlike the previous case, the `MPI_Status` structure fields are fully implemented by a Java object, as is represented in Figure 4.

```
Package wmpi;

Public class MPI_Status{

   Int count;
   public int MPI_SOURCE;
   public int MPI_TAG;
   public int MPI_ERROR;

}
```

Figure 4: Java representation of the MPI_Status structure.

The field `count` is not `public` because the MPI standard specifies that this field cannot be accessed directly by the user. There are specific methods to access this field.

3.4 Java Datatypes

The following table lists all the Java basic types and their corresponding C/C++ and MPI datatypes.

Java datatype	C/C++ Datatype	MPI datatype	JWMPI datatype
Byte	signed char	MPI_CHAR	MPI_BYTE
Char	unsigned short int	MPI_UNSIGNED_SHORT	MPI_CHAR
Short	signed short int	MPI_SHORT	MPI_SHORT
Boolean	unsigned char	MPI_UNSIGNED_CHAR	MPI_BOOLEAN
Int	signed long int	MPI_LONG	MPI_INT
Long	signed long long int	MPI_LONG_LONG_INT	MPI_LONG
Float	Float	MPI_FLOAT	MPI_FLOAT
Double	Double	MPI_DOUBLE	MPI_DOUBLE

Table 1: JWMPI datatypes.

Because Java is platform independent the size of simple types will be the same in all platforms. We have defined JWMPI datatypes that map directly to the Java datatypes and the user does not need to worry about the mapping between Java datatypes and MPI datatypes.

Beside these datatypes, JWMPI also provides the `MPI_PACKED` datatype that is used with packed messages, the `MPI_LB` pseudo-datatype that can be used to mark the lower bound of a datatype and `MPI_UB` that is used to mark the upper bound of a datatype.

4. Integrating Web-based Computing with JWMPI

In this section we will describe how JWMPI has been integrated with another tool for parallel processing that exploits the idea of Web-based computing. This tool is called JET and is described in [Silva97].

Originally the JET system was strictly oriented for Internet computing. However, in the recent version of JET it became possible to use some other existing high-performance computing resources, like cluster of workstations or parallel machines. The basic idea is to allow existing clusters of machines running PVM or MPI to inter-operate with a JET computation. The next sub-sections present an overview of the JET project and briefly describe the JET-Bridge, a software module that allows the integration of JET with WPVM/WMPI applications [Silva98].

4.1 A General Overview of the JET Project

JET is a Java software infrastructure that supports parallel processing of CPU-intensive problems that can be programmed in the Master/Worker paradigm. There is a Master process that is responsible for the decomposition of the problem into small and independent tasks. The tasks are distributed among the Worker processes that execute a quite simple cycle: receive a task, compute it and send the result back to the master. The Master is responsible for gathering the partial results and to merge them into the problem solution. Since every task is independent from each other, there is no need for communication between worker processes.

The Worker processes execute as Java applets inside a Web browser. The user that wants to volunteer his spare CPU cycles to a JET computation just need to access a Web page by using a Java-enabled browser. Then, she just has to click somewhere inside the page and one Worker Applet is downloaded to the client machine. This Applet will communicate with a JET Master that executes on the same remote machine where the Web page came from. Figure 5 presents the structure of the JET virtual machine.

The volunteer machines may join and leave the computation at any instant of time. Thereby, the execution environment is completely dynamic. The JET system provides some mechanisms to tolerate the frequent changes on the parallel virtual machine and include support for dynamic task distribution. These mechanisms are used for fault-tolerance and load-balancing purposes.

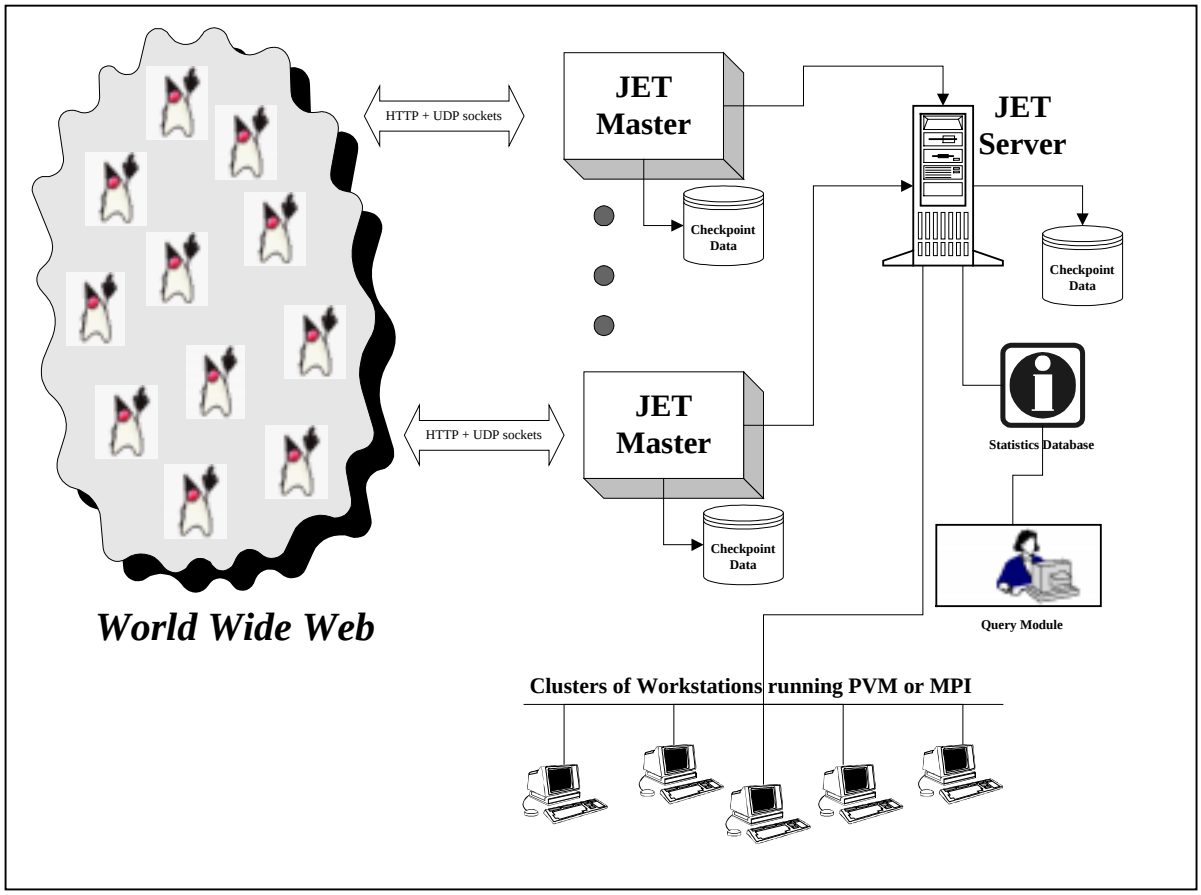

Figure 5: The Structure of the JET virtual machine.

4.2 The JET-Bridge

The functioning of the JET-Bridge assumes that the applications that will execute in the cluster elect one of the processes as the Master of the cluster. Usually this is the process with rank 0. The Master process is the only one that interacts with the JET-Bridge. Inside the cluster the application may follow any programming paradigm although we have only been used the JET-Bridge with Task-Farming applications.

The Master process of a WPVM/WMPI cluster needs to create an instance of an object (`JetBridge`) that implements a bridge between the cluster and the JET Master. This object is responsible by all the communication with the JET Master. The Master process from a WPVM/WMPI cluster gets some set of jobs from the JET Master, and maintains them in an internal buffer. These jobs are then distributed among the Workers of the cluster. Similarly, the results gathered by the WPVM/WMPI Master process are placed in a separate buffer and will be sent later to the JET Master. This scheme is represented in Figure 6.

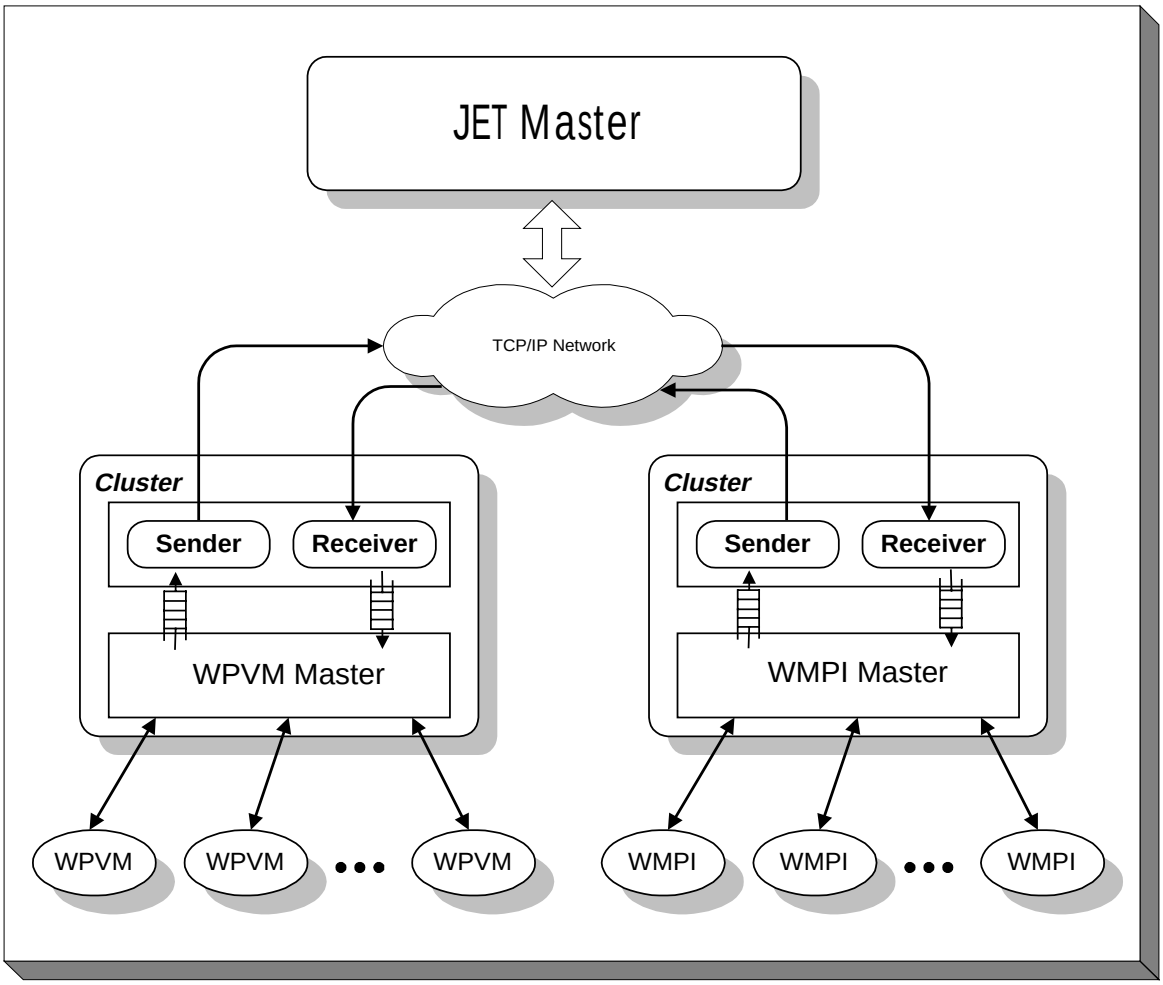

Figure 6: Interoperability of JET with PVM/MPI clusters.

The Master is the only process of the cluster that connects directly with the JET machine. This process is the only one that needs to be written in Java. The Worker processes can be implemented in any of the languages supported by WMPI/WPVM libraries (i.e. C, Fortran and Java) and all the heterogeneity is solved using the Java bindings. In the next section we present some performance results that show the effectiveness of this approach.

5. Performance Results

In this section we present some performance results of the two Java bindings that we have implemented: JWMPI and JWPVM. We also present some results of an experimental study that made use of JET-Bridge together with our Java bindings, to show the effectiveness of combining WPVM and WMPI with Web-based computations. All the measurements were taken with the `NQueens` benchmark with 14 queens in a cluster of Pentiums 200MHz running Windows NT 4.0, which are connected through a non-dedicated 10 Mbit/sec Ethernet.

The next Table presents the legend to some versions of the NQueens benchmark that we have implemented in our study. This legend will be used in some of the Figures that are presented in the rest of the section.

Versions of Nqueens Benchmark	
Legend	**Description**
CWMPI **CWPVM**	C version.
JWMPI **JWPVM**	Java version.
JWMPI (Native) **JWPVM (Native)**	Java version where the real computation is done by a call to a native method written in C.

Table 2: Legend to the different versions of NQueens Benchmark.

5.1 Java Bindings

In the first experiment that is presented in Figure 7 we compare the performance of the Java against the C version of the NQueens benchmark. Both versions were using WMPI and WPVM libraries to communicate. The Java processes are executing with a Just-in-Time compiler by using the Symantec JIT that is distributed with JDK1.1.4. It uses our Java bindings to access the WMPI/WPVM routines.

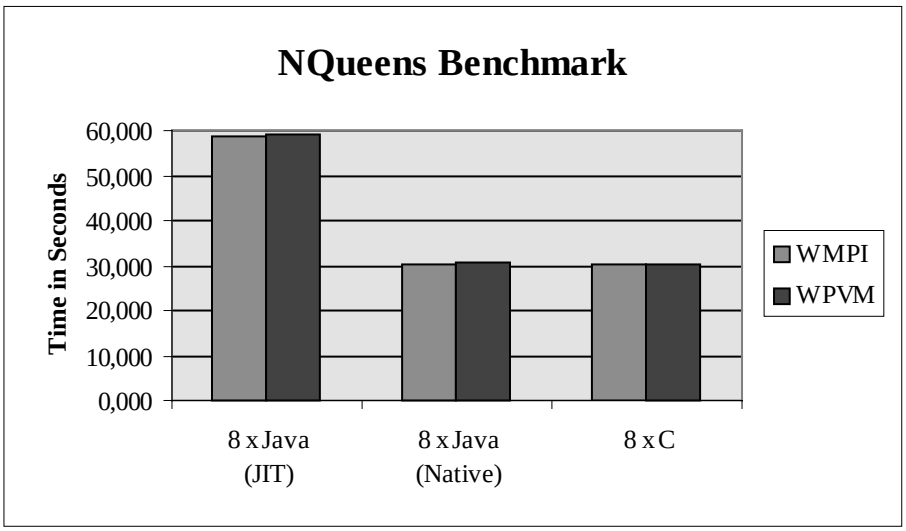

Figure 7: The performance of Java versus C.

As can be seen in Figure 7, Just-in-Time compilation cannot achieve the performance of C compiled code, but when compared with interpreted Java (see Figure 8) it presents a drastic increase in performance. We believe that with the evolution of JIT compilers and the appearance of new technologies that make use of dynamic compilation techniques, like the new HotSpot VM of Sun [Armstrong98] the Java performance gap of Java will be resolved. Figure 7 also presents a Java version of the NQueens benchmark that uses a native method written in C to compute the kernel of the algorithm. The results obtained with this Java (Native) version allow us to conclude that practically no overhead is introduced by our Java bindings.

In the next Figure we present the Java interpreted version results together with the previous ones. We can see that interpreted Java run approximately 10 times slower when compared with Just-in-Time compilation and near 20 times slower when compared with C compiled code. From now on all the presented performance results will use Just-in-Time compilation.

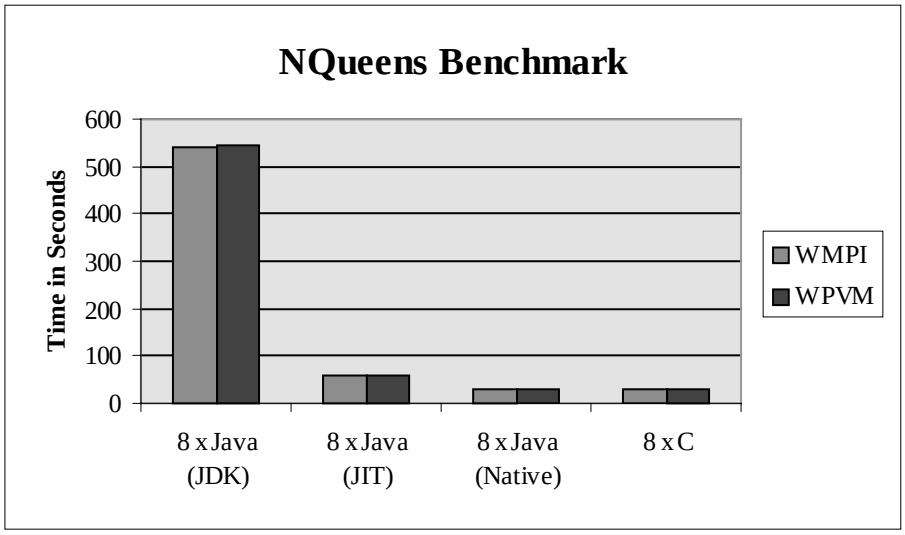

Figure 8: Interpreted Java versus JIT and C compiled code.

In Figure 9 we present several different combinations of using WMPI and an heterogeneous configuration of processes, where some of them were written in Java, others were written in C, and others were using Java and native code. These experiments are quite interesting since they show we can have real heterogeneous computations thanks to use of the Java bindings.

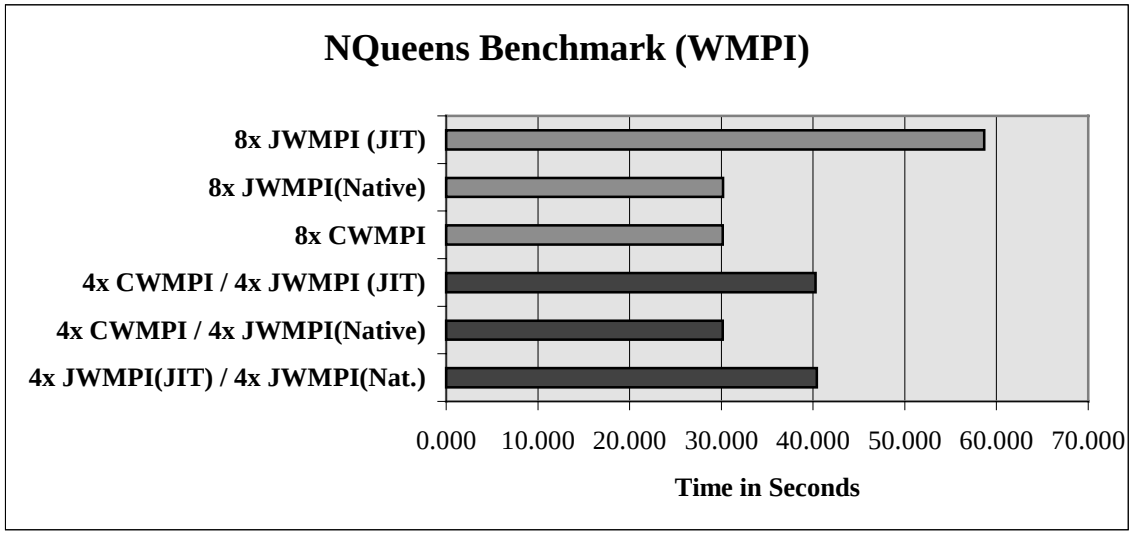

Figure 9: Heterogeneous clusters of processes using the WMPI library.

In our implementations of the NQueens Benchmark the jobs are delivered on demand, allowing the faster workers to compute more jobs than the slower ones. All the computations that include C processes or Java processes that use the native version of the kernel present the best performance.

The next sub-section presents some performance results of an experimental study that combines Web-based computations with PVM/MPI clusters.

5.2 Heterogeneous Parallel Computing

In Figure 10 we compare the performance results of four different computations. The first two columns represent the execution time of a JET computation in a cluster of 8 machines running the Java WMPI/WPVM versions of the NQueens benchmark. The third column presents a JET computation with 8 Java Applets running inside the Netscape Navigator 4.0. The last column presents a heterogeneous computation that combines a cluster of 3 Java WMPI processes, a cluster of 3 Java WPVM processes and 2 Java Applets.

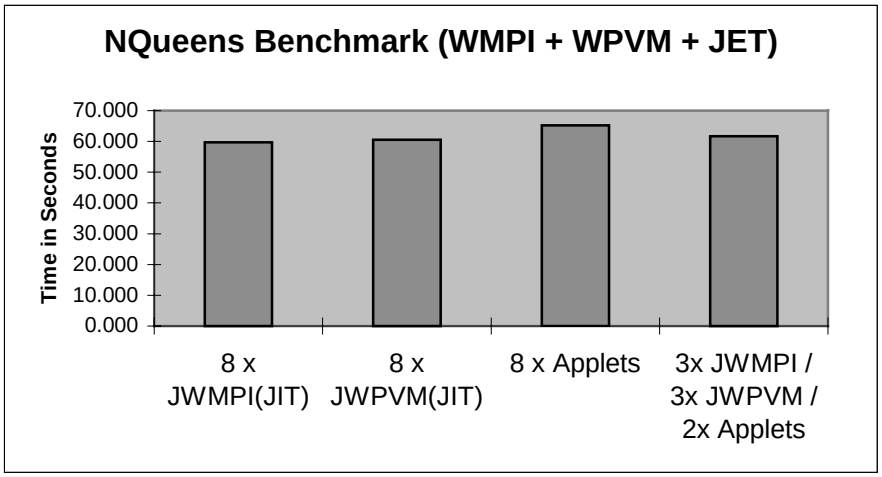

Figure 10: The performance of Cluster computing and Applet-based computing.

As we can see the results obtained with the WMPI and WPVM cluster running Java applications, are slightly better from the results obtained with Java Applets. Nevertheless, the execution time in the last configuration (JWMPI+JWPVM+Applets) seems very competitive with the results taken in pure clusters. In Table 3 we present the distribution of jobs among the different workers in this last heterogeneous configuration. These results are an average of 3 experiments. The Applet workers compute fewer jobs than the cluster workers (both in WMPI and WPVM).

NQueens Benchmark		Average
Machine	**Process**	**No of Jobs**
#1	Applet	19.33
#2	Applet	19.00
#3	JWPVM(JIT)	21.00
#4	JWPVM(JIT)	20.67
#5	JWPVM(JIT)	21.00
#6	JWMPI(JIT)	21.00
#7	JWMPI(JIT)	21.00
#8	JWMPI(JIT)	21.00
	Total Time (sec):	**61.723**

Table 3: Distribution of jobs in a heterogeneous JET computation.

In Figure 11 we present several different combinations of heterogeneous WMPI configurations. More important than the absolute results this experiment has shown the importance of the JET-Bridge and the Java bindings: these two modules allow the user to exploit the potential of a heterogeneous computation. Where some processes were executing as Java Applets, other as Java applications with WMPI, other processes were written in C and still others in a hybrid approach: Java and native C code. All these processes were executing inside the same application.

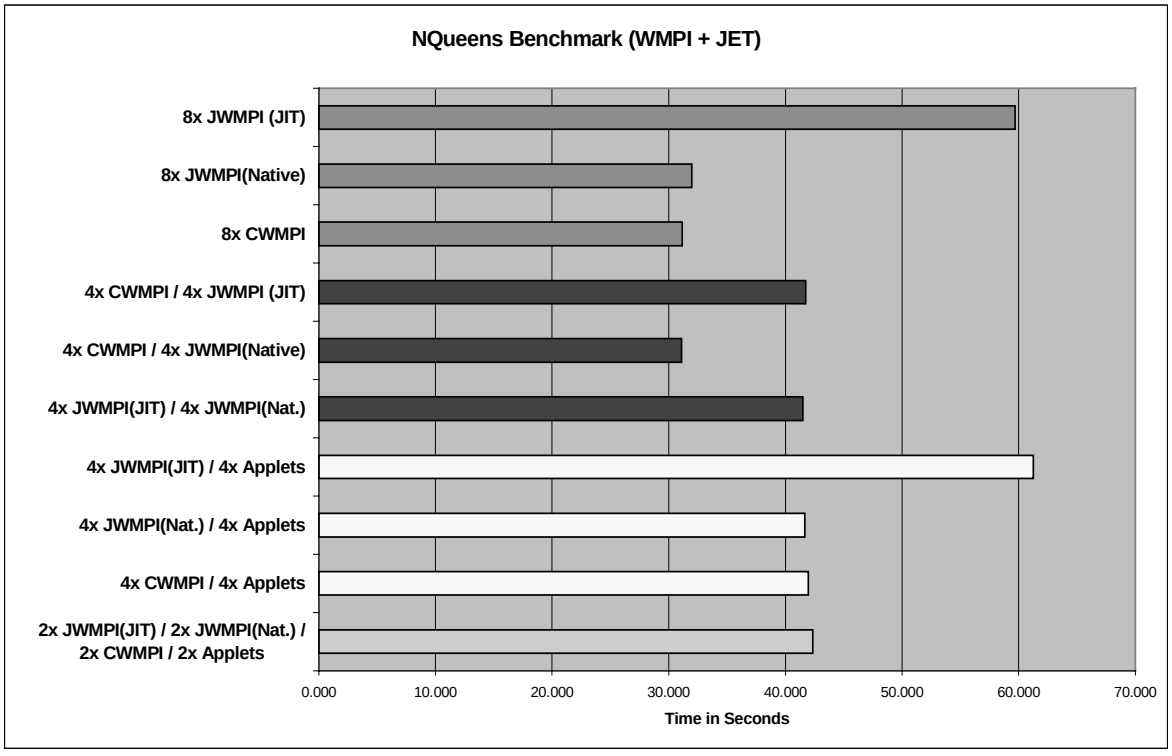

Figure 11: Performance results of heterogeneous configurations using WMPI.

This last heterogeneous configuration results in a sort of meta-application. Table 4 presents the distribution of jobs among the different type of workers in this computation. It presents the average results of 3 experiments.

NQueens Benchmark		Average
Machine	**Process**	**No of Jobs**
#1	JWMPI (JIT)	14.00
#2	JWMPI (JIT)	14.00
#3	Applet	13.00
#4	Applet	12.67
#5	JWMPI(Native)	26.67
#6	JWMPI(Native)	26.67
#7	CWMPI	27.00
#8	CWMPI	26.67
	Total Time (sec):	**42.332**

Table 4: Distribution of jobs among different WMPI processes in a JET computation.

As we can see from the Table, the majority of the jobs are performed by the C version and by the Java version that uses the native method. This sort of Master-Worker applications is automatically load-balanced, where the faster workers are able to compute more jobs than the slower ones.

6. Related Work

The idea of providing access to standard libraries written in other languages is very attractive and there are several similar on going projects.

JavaMPI is a Java binding for MPI that was developed in Syracuse University [JavaMPI]. In this binding only a few routines of MPI were implemented. JavaMPI does not use JNI; it uses instead the old native programming interface provided by JDK1.0.2 that is no longer supported in future releases of JDK.

The mpiJava [Baker98] is an object-oriented Java interface to the standard Message Passing Interface (MPI). The interface was developed as part of the HPJava project, but mpiJava itself does not assume any special extensions to the Java language - it should be portable to any platform that provides compatible Java-development and native MPI environments. The current release of mpiJava provides the full functionality of MPI 1.1 and, like JWMPI, is implemented as a set of JNI wrappers to native MPI packages. They intend to add new features such as object serialization.

jPVM is a Java to PVM interface [jPVM]. Like our binding jPVM also made use of JNI and we have ported this binding to our Windows version of PVM.

A very interesting idea was presented in [Mintchev97] and [Getov98]. They developed a tool, called JCI, that can be used for automatically binding existing native C libraries to Java. They used the JCI tool to bind MPI, PBLAS and ScaLAPACK to Java.

The JPVM library presented in [Ferrari98] provides a Java-based tool for PVM programming. Unlike jPVM, JPVM is a real implementation of PVM in Java and it presents a serious drawback: the lack of compatibility with PVM.

JavaNOW is another Java-based parallel programming tool presented in [JavaNOW]. This system implements the Linda programming model in Java and is not directly related with PVM and MPI. Finally, in [Foster96] was presented another Java binding, but this one is oriented to the Nexus system.

7. Conclusions

Providing access to standard libraries often used in high-performance and scientific programming seems imperative in order to allow the reuse of existing code that was developed with MPI and PVM.

In this paper, we have described the implementation of a Java interface for WMPI and we compared the performance of a parallel benchmark when using the Java interface in WMPI, WPVM and the corresponding C versions. The first results are quite promising and show the effectiveness of our Java binding for WMPI. Although the use of pure Java code seems quite promising there is still a small performance gap between Java and C code. Until the Java compiler technology reaches maturity, the use of native code in Java programs is certainly a way to improve performance.

The second set of results were taken in a mixed configuration where some of the processes were executing in Java and others in C. Those experiments show that it is possible to achieve really heterogeneous computations where we can have processes of the same parallel application running in different languages.

More than the heterogeneity at the language level we also presented a solution that masks the heterogeneity at the platforms level. With the use of the Java bindings for WMPI and the software module that was implemented in the JET system (JET-Bridge) it became possible to execute meta-applications using different tools: some tasks are executed in a Web-based computing tool while the other tasks can be executed in a cluster platform running WMPI.

It is our believe that Java will be the dominant language in the next coming years and that it can also be used for high-performance and scientific computing provided there are the right tools to achieve this goal. The Java components that were described in this paper can be a small but useful contribution to that goal.

References

[Alves95] A.Alves, L.M.Silva, J.Carreira, J.G.Silva, *"WPVM: Parallel Computing for the People"*, Proc. of HPCN'95, High Performance Computing and Networking Europe, May 1995, Milano, Italy, Lecture Notes in Computer Science 918, pp. 582-587

[Armstrong98] Eric Armstrong, "*HotSpot: A new breed of virtual machine*", JavaWorld, March 1998, http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.html

[Baker98] M. Baker, B. Carpenter, Sung H. Ko, and X. Li. *"mpiJava: A Java interface to MPI"*. Presented at First UK Workshop on Java for High Performance Network Computing, Europar 1998.

[Blundon98] W.Blundon, "*Predictions for the Millenium*", Java World Magazine, February 1998,

[Ferrari98] A.J. Ferrari, *"JPVM: Network Parallel Computing in Java"*, Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing, February 1998, Palo Alto, California

[Foster96] I.Foster, G.K.Thiruvathukal, S.Tuecke. *"Technologies for Ubiquitous Supercomputing: A Java Interface to the Nexus Communication System*", Syracuse NY, August 1996.

[Getov98] V. Getov, S. Flynn-Hummel, S. Mintchev, *"High-Performance Parallel Programming in Java: Exploiting Native Libraries"*, Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing, February 1998, Palo Alto, California

[Grande] Java Grande Forum home page, `http://www.javagrande.org/`

[Hoff98] A. van Hoff, "*Java: Getting Down to Business*", Dr Dobbs Journal, pp. 20-24, January 1998

[JavaMPI] MPI Java Wrapper Inplementation, by Yuh-Jye Chang, B. Carpenter, G. Fox, http://www.npac.syr.edu/users/yjchang/mpi/mpi.html

[JavaNOW] JavaNOW Project, http://www.mcs.anl.gov/george/projects.htm

[JNI] Java Native Interface Homepage, http://www.javasoft.com/docs/books/tutorial/native1.1/

[jPVM] jPVM Homepage, http://homer.isye.gatech.edu/chmsr/jPVM/

[Marinho98] J. G. Silva, *"WMPI: Message Passing Interface for Win32 Clusters"*, Proceedings of EuroPVM/MPI98, 5th European PVM/MPI User's Group Meeting, September 1998, Liverpool, UK, Lecture Notes in Computer Science 1497, pp. 113-120

[Mintchev97] S. Mintchev, V. Getov, "*Automatic Binding of Native Scientific Libraries to Java*", Proceedings of ISCOPE'97, Springer LNCS, September 5, 1997

[MPI] Message Passing Interface, http://www.mcs.anl.gov/mpi/

[MPICH/NT] MPI on Windows NT, http://www.erc.msstate.edu/mpi/mpiNT.html/

[PVM] Parallel Virtual Machine, http://www.epm.ornl.gov/pvm/

[PVMWIN32] PVMWIN32 ftp site, http://www.netlib.org/pvm3/win32/

[Silva97] L.M.Silva, H.Pedroso, J.G.Silva. "*The Design of JET: A Java Library for Embarrassingly Parallel Applications*", WOTUG'20 - Parallel Programming and Java Conference, Twente, Netherlands, 1997

[Silva98] L.M.Silva, P. Martins, J. G. Silva, "Merging Web-Based with Cluster-Based Computing", Proceedings of ISCOPE'98, Second International Symposium in Object-Oriented Parallel Environments, Santa Fe, NM, USA, December 1998, Springer LNCS 1505, pp. 119-126

http://www.javaworld.com/javaworld/jw-02-1998/jw-02-blundon.html/