Heterogeneous Parallel Computing using Java and WMPI

Luís Moura 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 present briefly the implementation of a Java interface for WMPI, a Windows-based implementation of MPI. Then, we describe a system 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 the Java bindings for WMPI. This solution allows the execution of meta-applications over a mixed configuration of platforms, execution models and programming languages. The resulting system provides a way to solve the problem of heterogeneity and to unleash the potential of diverse computational resources and programming tools.

1. Introduction

The Java language has achieved an incredible success among the IT community and it has been largely used in Internet applications, client/server computing, office tools, embedded systems and business applications [Hoff98]. The nice features of the language, like portability, robustness and flexibility can also be very interesting for the development of scientific and parallel applications. Several projects started to use Java in high-performance and numeric computing and, as a consequence, a JavaGrande Forum [JavaGrande] was created in order to establish some consensus among the HPC community that is developing Java-based tools and systems.

One of the goals of the JavaGrande is to define a specification for a Java MPI API and a position document is available at [Carpenter]. There are some implementations of Java bindings for MPI, like `mpiJava` [Baker98], `JavaMPI` [Mintchev97], `MPIJ` [Dogma] and `JMPI` [Crawford].

Since 1994 we have been involved in the development of MPI and PVM libraries for Windows NT [Alves95][Marinho98] and both implementations (`WPVM` and `WMPI`) have been highly used.
We also provide a Java-based interface for Windows PVM and MPI. We have ported the `jPVM` interface [jPVM] for `WPVM` and a Java binding for `WMPI` has been developed from scratch. In this paper we briefly describe the implementation of the `JWMPI` interface.

These Java bindings allow the communication with C/C++ applications by using MPI or PVM. The idea is not to replace all the software written in traditional languages with new Java programs. On 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 a reduced cost when writing new applications in Java.

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. The related work is described in Section 6, while Section 7 concludes the paper.

2. Implementations of PVM and MPI for Windows NT

`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. These ports can run in heterogeneous clusters of Windows 95/NT and Unix machines.

`WPVM` (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 (v1.3). The library is available at [WPVM].

On the other hand, `WMPI` 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. The library is available at [WMPI].

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.

In the development of this package we tried to provide an MPI-like API. To achieve this similarity, all the methods corresponding to WMPI functions are defined in class JWMPI and have 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.

```
#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[]){
      Integer rank = new Integer();
      Integer size = new Integer();
      /* 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 JWMPI.

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.

3.5 Problems due to strong typing and the absence of pointers

All MPI functions with choice arguments associate actual arguments of different datatypes with the same dummy argument. Java does not allow this since it has no pointers and the associated casting. When we have methods with different arguments we need to use method overloading as shown in Figure 5.

```
public static native MPI_Send (int[], ...);
public static native MPI_Send (long[], ...);
```

Figure 5: Example of methods with different argument datatypes.

However, there are many MPI communication functions, e.g. `MPI_Send`, `MPI_Bsend`, `MPI_Ssend`, `MPI_Rsend`, etc. If we used this approach, we would have several native methods for each function and datatype. To overcome this problem we introduce a set of native methods, called **`SetData`**, that performs the polymorphism between different Java datatypes. These native methods return the starting memory address of the array as a Java long variable that can be used in the MPI routines.

For example, in Figure 6 we can see the original `WMPI` call to `MPI_Send` in C/C++, followed by the equivalent Java call using our `JWMPI` interface.

```
MPI_Send(array, 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD);
MPI_Send(SetData(array), 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD);
```

Figure 6: Equivalent C/C++ and Java calls to `MPI_Send`.

Another variant of these methods was also implemented. This variant has one additional argument that specifies an index into the array. Figure 7 shows an example of this use.

```
MPI_Send(&array[5], 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD);
MPI_Send(SetData(array,5), 10, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD);
```

Figure 7: Equivalent C/C++ and Java calls to `MPI_Send` using an index.

3.6 Mapping between `WMPI` and `JWMPI` arguments

Table 2 presents the mapping between `WMPI` and `JWMPI` arguments. There are some exceptions to the mapping scheme presented in the Table. The methods `MPI_Pack`, `MPI_Unpack`, `MPI_Attach_buffer` and `MPI_Dettach_buffer` use a byte array instead of the value returned by the method `SetData`. This is due to the particular behaviour of these routines. When we are packing and unpacking data or declaring a memory zone to be attached to `WMPI` is much more efficient to use an array of bytes.

WMPI argument	JWMPI argument
Int	int
int*	java.lang.Integer
int[]	int[]
void[]	SetData(type[])
MPI_Aint	long
MPI_Aint*	java.lang.Long
MPI_Aint[]	long[]
MPI_Status (or *)	MPI_Status
MPI_Datatype (or *)	MPI_Datatype
MPI_Comm (or *)	MPI_Comm
MPI_Group (or *)	MPI_Group
MPI_Request (or *)	MPI_Request
MPI_Op (or *)	MPI_Op
MPI_Errhandler (or *)	MPI_Errhandler

Table 2: Mapping between WMPI and JWMPI arguments.

4. Integrating Web-based Computing with JWMPI

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

Originally the JET system was strictly oriented for volunteer computing over the Web. 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 just needs 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 8 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 mechanisms to tolerate failures in the applications and include support for dynamic task distribution. These mechanisms are used for fault-tolerance and load-balancing purposes.

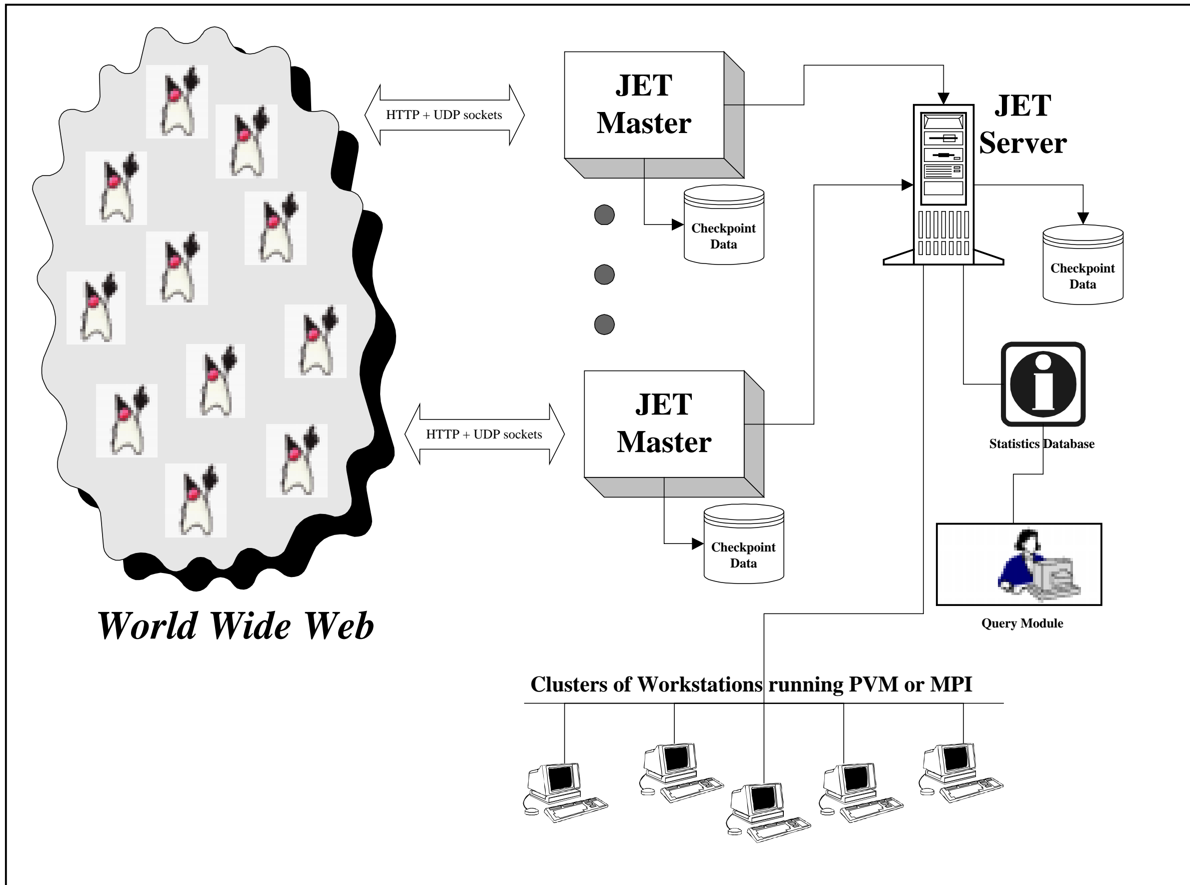

Figure 8: The Structure of the JET virtual machine.

4.2 The `JET-Bridge`

The `JET-Bridge` component assumes that the application that will execute in the cluster selects one of the processes as the Master. 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 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 for 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 9.

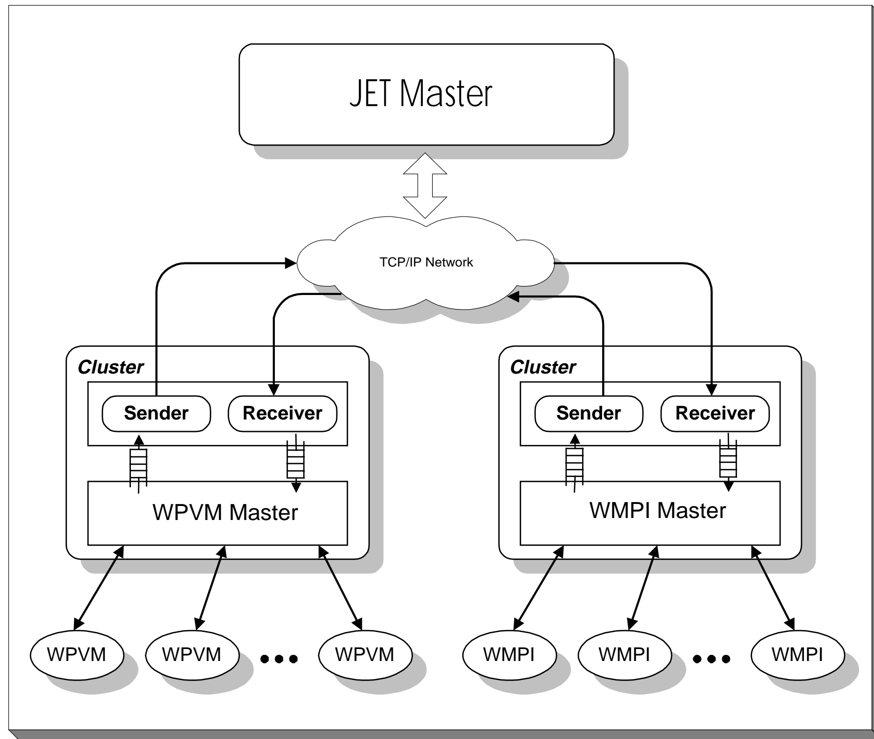

Figure 9: Interoperability of JET with WPVM/WMPI clusters.

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 native method written in C.

Table 3: Legend to the different versions of NQueens benchmark.

5.1 Java Bindings

In the first experiment, that is presented in Figure 10. 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.

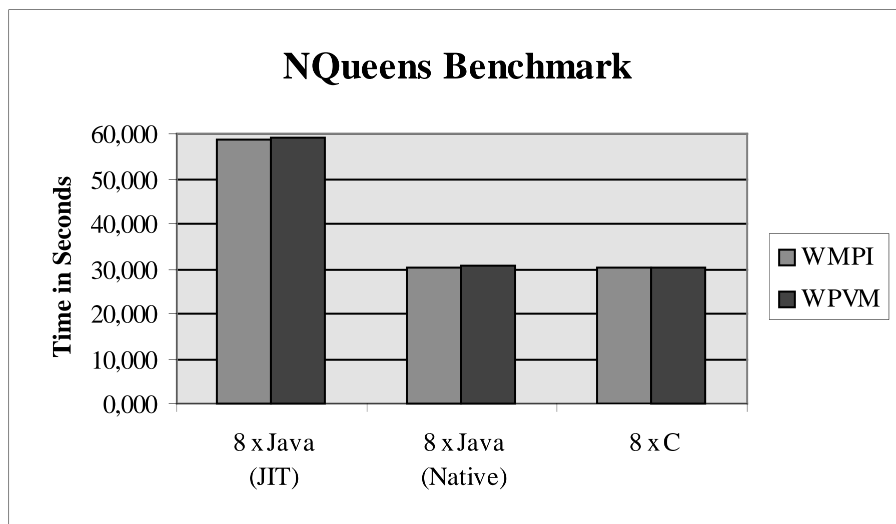

Figure 10: The performance of Java versus C.

As can be seen in Figure 10, just-in-time compilation cannot achieve the performance of C compiled code. It is about two times slower, which actually is not that bad. The Figure 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 Figure 11 we present several different combinations of using WMPI and an heterogeneous configuration of processes, where some of them were written in Java, other processes were written in C and the remaining were using Java and native code.

These experiments are quite interesting since they show we can have real heterogeneous computations by using our Java bindings.

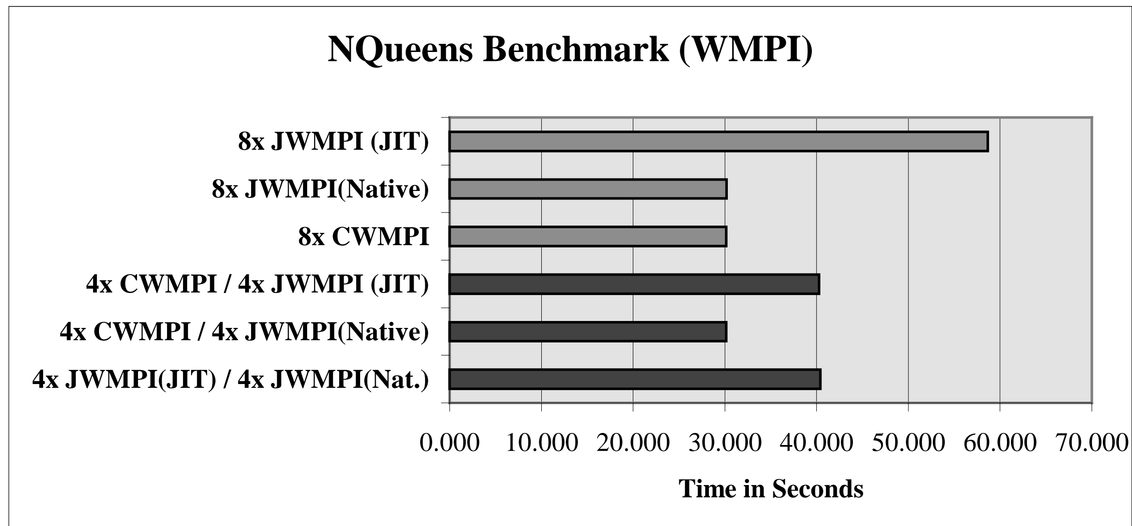

Figure 11: Heterogeneous clusters of processes using the `WMPI` library.

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 12 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 `JWMPI` processes, a cluster of 3 `JWPVM` processes and 2 Java applets.

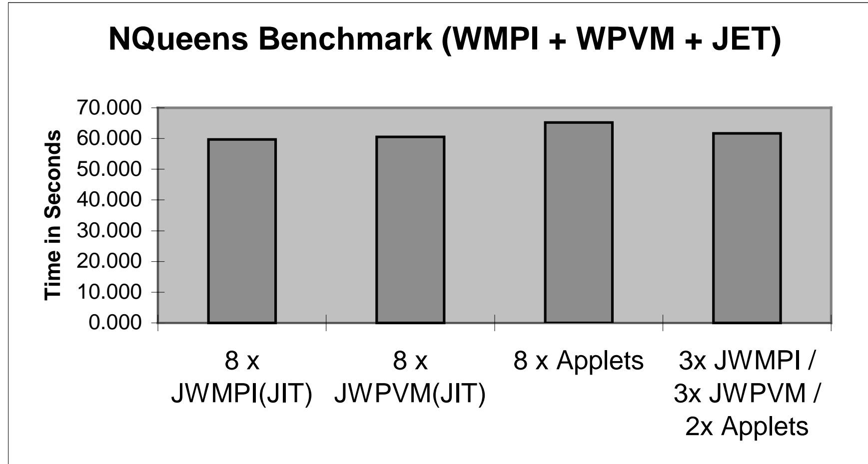

Figure 12: The performance of Cluster computing and applet-based computing.

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 4 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 4: Distribution of jobs in a heterogeneous `JET` computation.

In Figure 13 we present several different combinations of heterogeneous `WMPI` configurations. This experiment shows the importance of the `JET-Bridge` and the Java bindings: combining these two modules allow the user to exploit the potential of a heterogeneous computation. In the same application we could have processes executing as Java applets, processes running `JWMPI` and C processes running `WMPI`.

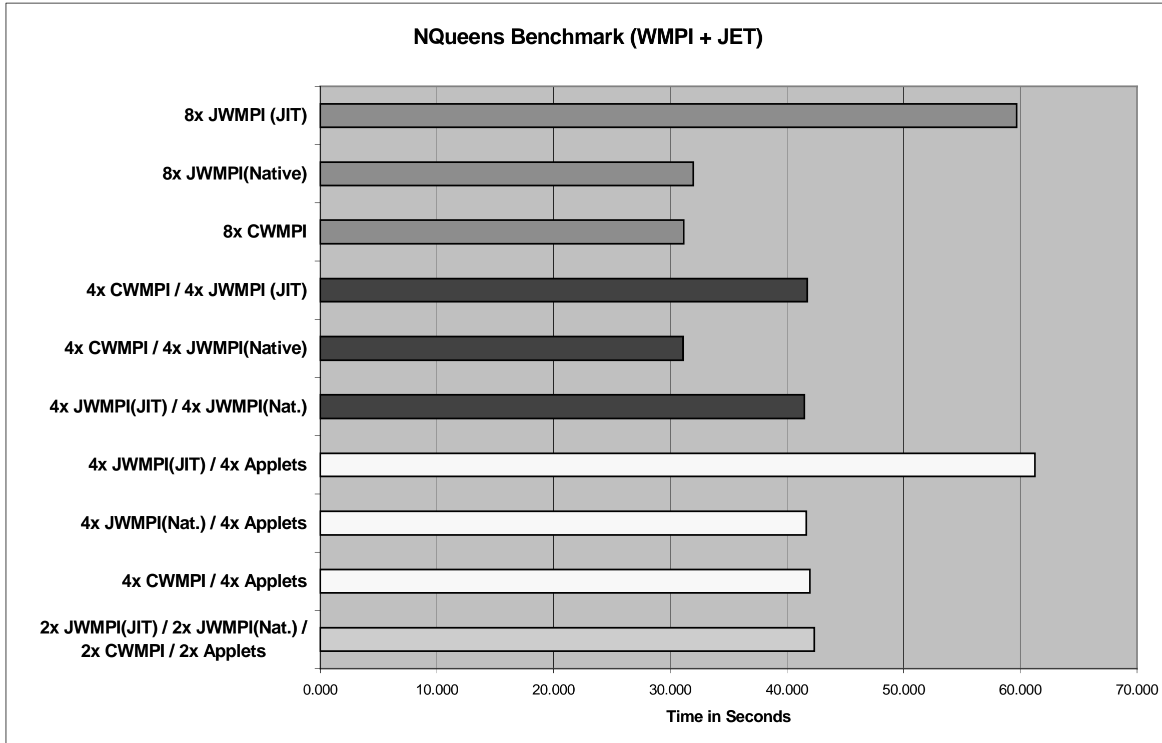

Figure 13: Performance results of heterogeneous configurations using `WMPI`.

This last heterogeneous configuration results in a sort of meta-application. Table 5 presents the distribution of jobs among the different types 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 5: 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

There are some other examples of Java bindings for MPI. In [Baker98] is presented `mpiJava` an object-oriented implementation of a Java interface for MPI. This Java binding are [Baker98] is very similar to our binding and it also runs on top of WMPI. A different approach is presented in [Mintchev97] where the Java wrappers are automatically generated from the C MPI header by using a Java-to-C interface generator tool (`JCI`). This same tool was also used to bind PBLAS and ScaLAPACK to Java. A full implementation of MPI written in Java has been made in the DOGMA Project [Dogma]. This strategy departs from the previous one since the MPI core was re-implemented in Java, leading to a potentially poor performance. Finally, MPI SoftTech Technologies has announced a Java binding for their commercial implementation of MPI [Crawford].

In the area of Web-based computing there are some other projects that present some similar goals to the `JET` Project. Examples include the Javelin project [Capello97], Charlotte [Baratloo96], Albatross [Bal98], Ninflet [Takagi98], ParaWeb [Brecht96] and Popcorn [Camiel96]. However, none of these projects has shown the same level of effectiveness in the merging of Web-based computing with cluster-based computing and in the interoperability of Java-based WMPI and WPVM together with C code written in those libraries.

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 briefly described the implementation of a Java interface for `WMPI` and we compared the performance of a parallel benchmark when using the `JWMPI`, `JWPVM` and the corresponding C versions. The first results are quite promising and show the effectiveness of our Java bindings. 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 of the language we have presented a solution that masks the heterogeneity at the platform level. With the use of the Java bindings for `WMPI` and the software `JET-Bridge` component it became possible to execute meta-applications using different tools: some tasks are executed in a Web-based computing tool (as Java applets) while the other tasks can be executed in a cluster platform running `WMPI`.

It is our belief that Java will be the dominant language in the 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.

Acknowledgments

This work was partially supported by the Portuguese *Ministério da Ciência e Tecnologia*, the European Union through the R&D Unit 326/94 (CISUC) and the project PRAXIS XXI 2/2.1/TIT/1625/95 (PARQUANTUM).

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

[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.

[Bal98] H.Bal. "*Albatross- Wide Area Cluster Computing*", http://www.cs.vu.nl/albatross

[Baratloo96] A.Baratloo, M.Karaul, Z.Kedem, P.Wyckoff. *"Charlotte: Metacomputing on the Web"*, Proc. ISCA International Conference on Parallel and Distributed Computing, PDCS'96, Dijon, France, pp.181-188, Sept. 1996

[Brecht96] T.Brecht, H.Sandhu, M.Shan, J.Talbot. *"ParaWeb: Towards World-Wide Supercomputing"*, Proc. 7th SIGOPS European Workshop on System Support for Worldwide Applications, 1996

[Camiel96] N.Camiel, S.London, N.Nisan, O.Regev. "*The Popcorn Project: Distributed Computation over the Internet in Java*", Proc. 6th International World Wide Web Conference, April 1997

[Cappelo97] P.Cappelo, B.Christiansen, M.F.Ionescu, M.Neary, K.Schauser, D.Wu. *"Javelin: Internet-based Parallel Computing using Java"*, ACM 1997 Workshop on Java for Science and Engineering Computation, Las Vegas, USA, June 1997

[Carpenter] B.Carpenter, V.Getov, G.Judd, T.Skjellum, G.Fox, "MPI for Java: Position Document and Draft API Specification", JGF-TR-03, JavaGrande Forum, Nov. 1998

[Crawford] G.Crawford, Y.Dandass, A.Skjellum. "The JMPI Commercial Message Passing Environment and Specification". http://www.mpi-softtech.com/publications

[Dogma] DOGMA Architecture, http://ccc.cs.byu.edu/DOGMA

[JavaGrande] 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

[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. Marinho, 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. "Towards Portable Message Passing in Java: Binding MPI", Proc. EuroPVM-MPI'97, Lecture Notes in Computer Science, Vol. 1332, Springer-Verlag, pp. 135-142, 1997

[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

[Takagi98] H. Takagi, S. Matsuoka, H. Nakada, S.Sekiguchi, M.Satoh, U.Nagashima. "*Ninflet: A Migratable Parallel Objects Framework using Java*", ACM 1998 Workshop on Java for High-Performance Network Computing, Palo Alto, CA, February 1998

[WPVM] WPVM Home Page. http://dsg.dei.uc.pt/wpvm/

[WMPI] WMPI Home Page: http://dsg.dei.uc.pt/w32mpi/