A Java Interface for WMPI

Paulo Martins, Luís M. Silva, João Gabriel Silva

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

Abstract. Traditionally, parallel programs have been developed in Fortran and C. However, the extraordinary success of the Java language has raised an impressive hype among the community of software programmers, in such a way that it seems quite promising that parallel programmers could also use Java in their applications.
In this paper, we describe the implementation of Java interface for WMPI, a Windows-based implementations of MPI that have been developed and supported by our group. We show some details about the implementation and we present some experimental results that compare the performance of Java WMPI, Java WPVM and the C programs counterparts.

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 [1][2]. Both libraries provide a C, C++ and a Fortran interface.

We have ported the PVM and MPI libraries to Microsoft Win32 platforms. The PVM port is called WPVM [3] (Windows Parallel Virtual Machine) and the MPI port is called WMPI [4]. 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 [5] 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 the recent past we have seen that 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 [6]. 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. In fact there is already a Java Grande Forum that was created to develop some consensus among the HPC community for the establishment of Java for parallel and scientific computing [7].

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 JavaPVM interface [8] for the Windows version of PVM (WPVM) and we develop from scratch a similar Java interface for WMPI. JavaWPVM and JavaWMPI 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 JavaWMPI interface. 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 to WMPI interface, while section 4 presents some performance results. The related work is described in section 5, while section 6 concludes the paper.

2. WPVM and WMPI

Both 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. Both ports can run together in a heterogeneous cluster 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 of 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.

---

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

3. A Java Binding for WMPI

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

3.1 Overview

All JavaWMPI 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.

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

3.2 Opaque objects used by WMPI

Opaque objects are system objects that are accessed through some handler. 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.

These Java classes hide inside them 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 JavaWMPI 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 Java Datatypes

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

Table 1. JavaWMPI datatypes.

Java datatype	C/C++ Datatype	MPI datatype	JavaWMPI 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

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

Beside these datatypes, JavaWMPI 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.

3.4 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 would have to use method overloading as shown in Figure 1.

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

Fig. 1. 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 tons of 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 2 we can see the original WMPI call to `MPI_Send` in C/C++, followed by the equivalent Java call using our Java to WMPI interface.

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

Fig. 2. Equivalent C/C++ and Java calls to MPI_Send.

Another variant of these methods was implemented. This variant has one additional argument that specifies an index into the array. Figure 3 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);
```

Fig. 3. Equivalent C/C++ and Java calls to MPI_Send using an index.

3.5 Mapping between WMPI and JavaWMPI arguments

The next Table presents the mapping between WMPI and JavaWMPI arguments.

Table 2. Mapping between WMPI and JavaWMPI arguments.

WMPI argument	JavaWMPI 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

When we use `int*` we refer to a pointer to a `int` variable and when we use `int[]` we refer to an array of `int` variables.

There are some exceptions to the mapping scheme presented in Table 2. 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 more efficient to use an array of bytes.

4. Performance Results

In this section we present some performance results of the two Java bindings that we have implemented. 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` application that we have implemented in our study. This legend will be used in the Figures that are presented in the rest of the section.

Table 3. Different versions of the `NQueens` benchmark.

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

In the first experiment, presented in Figure 4, we compare the performance of the Java version of the `NQueens` benchmark with the C version, both 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.

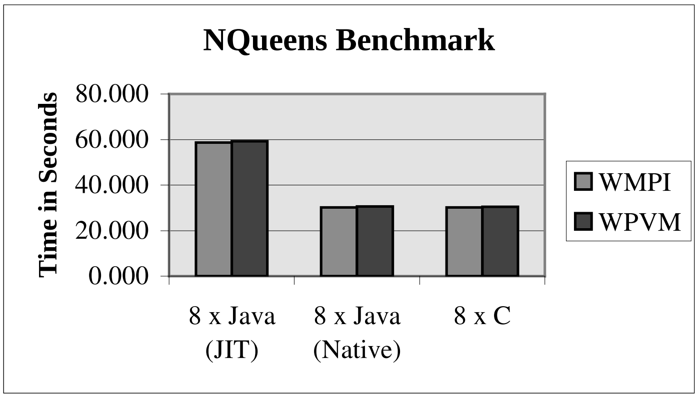

Fig. 4. The performance of Java versus C.

As can be seen in Figure 4, Just-in-Time compilation cannot achieve the performance of C compiled code, but when compared with interpreted Java it presents a drastic increase in performance. 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. 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 [9], the Java language will run as fast even or faster than compiled code.

Figure 4 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 5 we present several different combinations of heterogeneous clusters of processes using WMPI library. As can be seen we combined in an effective way programs that use Java and C processes in the same computation. These experiments are quite interesting since they prove that with the help provided by our Java bindings, we can achieve real heterogeneous computations.

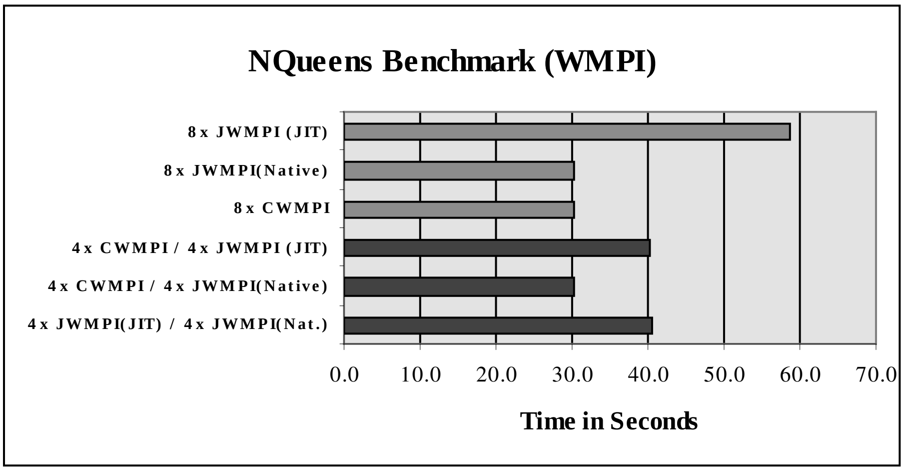

Fig. 5. Heterogeneous clusters of processes using the WMPI library.

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.

Until the Java compiler technology reaches maturity, the use of native code in Java programs is certainly a way to improve performance. 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.

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 [10]. 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.

JavaPVM is a Java to PVM interface [8]. Like our binding JavaPVM also made use of JNI and we have ported this binding to our Windows version of PVM.

A very interesting idea was presented in [11] and [12]. 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 [13] provides a Java-based tool for PVM programming. Unlike JavaPVM, 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 [14]. This system implemented the Linda programming model in Java and is not directly related with PVM and MPI. Finally, in [15] was presented another Java binding, but this one is oriented to the Nexus system.

7. Conclusions

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 C versions counterparts. The first results are quite promising and show that it is possible to program an application in Java that makes use of WPVM and WMPI. Some other 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 with some of the parts running in different languages.

References

1. Parallel Virtual Machine, http://www.epm.ornl.gov/pvm/
2. Message Passing Interface, http://www.mcs.anl.gov/mpi/
3. 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
4. Windows Message Passing Interface, http://dsg.dei.uc.pt/wmpi/
5. W.Blundon, "*Predictions for the Millennium*", Java World Magazine, February 1998, http://www.javaworld.com/javaworld/jw-02-1998/jw-02-blundon.html/
6. A. van Hoff, "*Java: Getting Down to Business*", Dr Dobbs Journal, pp. 20-24, January 1998
7. First Meeting of the Java Grande Forum, http://jhpc.org/grande/meeting.htm/
8. JavaPVM Homepage, http://homer.isye.gatech.edu/chmsr/JavaPVM/
9. Eric Armstrong, "*HotSpot: A new breed of virtual machine*", JavaWorld, March 1998, http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.html
10. MPI Java Wrapper Implementation, by Yuh-Jye Chang, B. Carpenter, G. Fox, http://www.npac.syr.edu/users/yjchang/mpi/mpi.html
11. S. Mintchev, V. Getov, "*Automatic Binding of Native Scientific Libraries to Java*", Proceedings of ISCOPE'97, Springer LNCS, September 5, 1997
12. 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
13. 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
14. JavaNOW Project, http://www.mcs.anl.gov/george/projects.htm
15. I.Foster, G.K.Thiruvathukal, S.Tuecke. "*Technologies for Ubiquitous Supercomputing: A Java Interface to the Nexus Communication System*", Syracuse NY, August 1996.