On Implementations Issues of Parallel Computing Applications in Java

Freddy Pérez
Advisor: Dr. Wilson Rivera
Electrical and Computer Engineering Department
University of Puerto Rico, Mayaguez Campus
Mayaguez, Puerto Rico 00681-5829
{f_perez, wrivera}@ece.uprm.edu

Abstract
This paper analyzes the current state of Java in numerical computing and the different approaches to parallel computing in Java. Matrix-matrix multiplication is an important kernel in High Performance Computing since many algorithms use it. The problem is to optimize data transfer and computation to perform the multiplication.
We review different approaches and issues related to the implementation of matrix-matrix multiplication. We implemented an algorithm in mpiJava that improve significantly others implementations of matrix-matrix multiplication in others approaches to parallel computing in Java.

1. Introduction
Over the last years Java has captured the attention of programmers and the different disciplines especially those that need high performance computing and numerical simulation. Numerical simulation is characterized by the following procedure. First, observe of real world. Next, derive mathematical equations that describes the phenomena. Then, discretize the equations resulting by suitable methods. Finally, solve the system equations resulting of the discretization. This procedure has a natural object oriented behavior. Thus, Computational Fluid Dynamics (CFD) has saw in Java an attractive option for implementation of its diverse methods. These methods use often matrix multiplication. This issue motivated to write this paper.
This paper is organized as follows. Section 2 discusses Java performance in numerical computing. Section 3 describes different approaches to parallel computing in Java. Section 4 presents an implementation of the matrix-matrix multiplication using mpiJava followed by an analysis of the results. This algorithm is based on use of the techniques suggested for improved the performance of Java. Finally, conclusions and future works are listed in Section 5.

2. Java and Numerical Computing
Java has achieved rapid success because of several features such as portability, safety and pervasiveness. Java is portable at both the source (the text in a *.java) and object (the byte code *.class) format levels, which means that programs can run on any machine that has an implementation of the Java Virtual Machine (JVM) and an appropriate Java compiler. Java code is safe because it executes programs in an environment that users are not doing nothing that they are not authorized. The most important feature Java offers is its pervasiveness, in all aspects. For example, Java is opening the way of new scale of Web-based global computing [Blount98]. In addition, Java is an object-oriented language, robust with no pointers and provides garbage collection.
These features make Java attractive for scientific computing. However, there are factors that limit Java for numerical computing.
One of these shortcomings is that Java treat multidimensional arrays only as arrays of one-dimensional arrays. Each access to a multidimensional array element requires multiple bound checks at runtime.
Java Grande Forum has proposed to implement a set of Java class of immutable, rectangular multidimensional array with the same
capabilities, and same performance, as Fortran arrays [JavaGrande]. Boisvert and Moreira [Boisvert01] showed that Java codes perform competitively with optimized C and Fortran. They compare three Java environments, IBM, Sun and Microsoft with two optimizing compilers: Borland and Microsoft. Java Sun 1.2 outperforms the other compilers of Java (IBM 1.8, MS 1.14) and C (Borland 5.5, MS VC++ 5.0) on 500-Mhz Intel Pentium on Windows 98 platforms, for SciMark benchmark. They pointed out that they are not comparing the two languages per se, but rather different implementations of compilers and execution environments.

3. Java and Parallel Computing

Parallel applications are implemented using message-passing programming using Parallel Virtual Machine (PVM), that contains the virtual machine capabilities required for meta computing, or the Message Passing Interface (MPI), a set of optimized libraries. These libraries were designed using the notion of objects. A number of different approaches have been proposed in order to use Java for parallel computing:

- mpiJava, Java/DSM, JavaPVM, use Java as wrapper for existing frameworks and libraries.
- MPI, jmpi, DOGMA, JVPM, JavaNOW, use pure Java libraries.
- HPJava, Manta, JavaParty, Titanium, extend Java language with new keywords. Use preprocessor or own compiler to create Java (byte code).
- WebFlow, IceT Javelin, web oriented and use Java applets to execute parallel tasks.

3.1 mpiJava

mpiJava was developed at the Syracuse University by Bryan Carpenter, Mark Baker, Geoffrey Fox and Guansong Zhang. This approach implements a Java API for MPI. The mpiJava API is modeled as closely as the C++ binding defined in the MPI 2.0 standard. Its communication functions are members of Comm or its subclasses [Baker98]. Thus the standard send and received operations of MPI are members of Comm with interfaces

\[
\text{public void Send ( Object buf, int offset, int count, Datatype data type, int dest, int tag)}
\]

where \( \text{buf} \) is an array, \( \text{offset} \) is the element where message starts, \( \text{Datatype} \) class describes the type of elements, \( \text{count} \) is the amount of elements to sent or received, and \( \text{tag} \) is used to identify the message. The basic Datatypes of mpiJava are shown in the Table 1. mpiJava is implemented as a Java interface to an underlying MPI implementation. The interface between mpiJava and MPI via the Java Native Interface (JNI) is showed.

<table>
<thead>
<tr>
<th>Table 1: Basic Datatypes of mpiJava</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI Datatype</strong></td>
</tr>
<tr>
<td>MPI.BYTE</td>
</tr>
<tr>
<td>MPI.CHAR</td>
</tr>
<tr>
<td>MPI.SHORT</td>
</tr>
<tr>
<td>MPI.BOOLEAN</td>
</tr>
<tr>
<td>MPI.INT</td>
</tr>
<tr>
<td>MPI.LONG</td>
</tr>
<tr>
<td>MPI.FLOAT</td>
</tr>
<tr>
<td>MPI.DOUBLE</td>
</tr>
<tr>
<td>MPI.OBJECT</td>
</tr>
</tbody>
</table>

This approach presents some limitations. It cannot mix primitive types or fields from different objects, displacements only operate within one-dimensional arrays, and cannot use MPI_TYPE_VECTOR to describe sections of multidimensional arrays [Carpenter]. mpiJava depends on the availability of a platform-dependent native MPI implementation. Consequently, this approach conflicts with the essence of Java, “Write once run anywhere”.

3.2 MPJ

MPJ was developed by a group of enthusiasts, chaired by Vladimir Getov. First, we discuss about the class structure shown in the Figure 1. Where the class MPI has only static members and global constants such as COMM_WORLD. The class Comm is the communicator class. This class contains all communicator functions such as send and receive. The Datatype class describes the type of elements in the message buffers of communication [Carpenter00]. The basic datatypes are the same of the mpiJava.
The syntax of a communication operation in MPJ is very similar to mpiJava and C or Fortran MPI communication. Next we show the syntax for send and receive:

```java
void Comm.Send ( Object buf, int offset,
int count, Datatype datatype, int dest,
int tag)

Status Comm.Recv( Object buf, int offset,
int count, Datatype data type, int
source, int tag)
```

This approach has an important benefit: it is highly portable because assumes only a Java development environment. However, the performance is moderate because it may need JNI for some applications (e.g. marshalling arrays) and also because of network speed limits.

### 3.3 jmpi

This approach proposes a pure Java implementation. jmpi is a class library of Java. The pure Java implemetation helps jmpi achieve portability and security. But the problem of execution slow is inherited. 

jmpi is built upon the Java Parallel Virtual Machine (JPVM) system that provides better communication in a networked environment. JPVM is based in PVM, it offers features such as thread safety, multiple communication and default-case direct message routing. JPVM is more like MPI implementations that others PVM architectures [Dincer98].

### 3.4 IceT

IceT is a metacomputing system. The basis for IceT computing is process-oriented distributed memory multicomputing. IceT is an experimental project in distributed systems [IceT].

### 4. Matrix - Matrix Multiplication

Consider the problem of computing $C=AXB$, where $A$, $B$ are dense matrices of size $MxN$ y $NxP$. We know that matrix-matrix multiplication involves $O(N^3)$ operations, because each $C_{ij}$ of $C$ is equal to:

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} B_{k,j}$$

We are looking for an efficient parallel algorithm for matrix-matrix multiplication using mpiJava. Consequently, we examine several algorithms developed for this purpose. First, consider an one-dimensional, column wise decomposition such that each processor is responsible for all computation associated with the $C_{ij}$'s at the assigned columns. Each task requires all of matrix $A$ to compute the $C_{ij}$ corresponding with its column. The algorithm perform very well when $N$ is much larger than number of processors $P$ [Foster95].

In a two-dimensional decomposition each task requires an entire row $A_{i,*}$ and column $B_{*,j}$ of $A$ and $B$ respectively. The one-dimensional decomposition requires $N^2/P$ data and the two-dimensional decomposition requires $N^2 P^{1/2}$.

Because of features of Java for manage arrays, we consider a one-dimensional row wise decomposition, as showed in Figure 2.

![Figure 2: One-dimensional row wise decomposition](image-url)
temp = temp +
RBuffer[k+i*n]*bb[k+j*p];
}
subtot[j + i*p] = temp;
temp = 0.0;
}

5. Results and Analysis

Using the later approach we tested the implementation and obtain the results shown in Table 2.

Table 2: Numerical results for matrix-matrix multiplication

<table>
<thead>
<tr>
<th>np/Size</th>
<th>2x2</th>
<th>128x128</th>
<th>256x256</th>
<th>400x400</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.8E-40</td>
<td>0.8832</td>
<td>7.5412</td>
<td>28.7133</td>
</tr>
<tr>
<td>2</td>
<td>8.2E-40</td>
<td>0.4445</td>
<td>3.8617</td>
<td>14.6902</td>
</tr>
<tr>
<td>4</td>
<td>0.0010</td>
<td>0.7305</td>
<td>7.0851</td>
<td>26.9468</td>
</tr>
<tr>
<td>8</td>
<td>0.0013</td>
<td>0.3050</td>
<td>3.5316</td>
<td>14.6650</td>
</tr>
<tr>
<td>16</td>
<td>0.0028</td>
<td>0.2015</td>
<td>1.8891</td>
<td>8.2826</td>
</tr>
</tbody>
</table>

comparing these results with the obtained by jmpi [Dincer98] and IceT [Gray97] we see that improve their results.

Moreover, the table show that the algorithm improve when the number of processors increment, except when we use 4 processors that produces an overhead at moment of execution. This problem is produced by the machine architecture. Also we see that efficiency is relatively maintained constant. This function change between 1 and 0.5, thus we could say that the algorithm is scalable.

6. Conclusions and Future Work

We believe Java will be an important tool for scientific and high performance applications. mpiJava need improve anything its features of installation and implement new functions that let to use bidimensional arrays.

The results showed before indicate that mpiJava is a tool more efficient for parallel processing that the approaches above.

We will work at design an implementation that optimize the Java resource and difficult.

References


[IceT] "The IceT Project", http://www.mathcs.emory.edu/icet/
