JOPI: A Java Object-Passing Interface
Nader Mohamed, Jameela Al-Jaroodi, Hong Jiang, and David Swanson
Department of Computer Science and Engineering
University of Nebraska–Lincoln, Lincoln, NE 68588-0115
Tel: (402) 472 1648
nmohamed, jaljaroo, jiang, dswanson@cse.unl.edu

ABSTRACT
Recently there has been an increasing interest in developing parallel programming capabilities in Java to harness the vast resources available in clusters, grids and heterogeneous networked systems. In this paper, we introduce a Java object-passing interface (JOPI) library. JOPI provides Java programmers with the necessary functionality to write object-passing parallel programs in distributed heterogeneous systems. JOPI provides an MPI-like interface that can be used to exchange objects among processes. In addition to the well-known benefits of the object-oriented development model, using objects to exchange information in JOPI is advantageous because it facilitates passing complex structures and enables the programmer to isolate the problem space from the parallelization problem. The run-time environment for JOPI is portable, efficient and provides the necessary functionality to deploy and execute parallel Java programs. Experiments were conducted on a cluster system and a collection of heterogeneous platforms to measure JOPI’s performance and compare it with MPI. The results show good performance gains using JOPI.

Categories and Subject Descriptors

General Terms
Experimentation, Languages.

Keywords
Java, heterogeneous systems, cluster, parallel programming, object-passing, object-oriented Systems.

1. INTRODUCTION
Increasing demand for high performance and powerful computing resources led to great advances in the area of distributed and parallel computing. In addition, clusters and heterogeneous networked systems have become very popular. This direction also led to the need for system and application software that can provide transparent and efficient utilization of the multiple machines in a distributed system. Moreover, Java’s popularity among developers is also increasing steadily and many research groups have started exploring the possibilities of using Java for multi-processor systems. Due to Java’s machine independence, it is possible to execute a program on any platform with a Java virtual machine (JVM).

In this paper, we discuss the advantages of the object-oriented paradigm and its appropriateness for parallel programming. Then we introduce JOPI, its support for parallel programming in Java and its unique features and advantages. JOPI extends Java to utilize Java objects for communication among the parallel application processes. Compared to MPI (data-passing), it is much simpler and more powerful and advantageous to use object-passing for communication. First, the syntax of the communication methods is much simpler and easier to remember. Second, complex data structures can be exchanged easily in a single step. Using objects for exchanging complex structures and huge data sets among parallel and distributed processes results in significantly reduced program code size. Furthermore, using object passing allows the programmer to separate the problem solution from the parallelization process, which simplifies maintenance and allows for reuse.

The JOPI run-time environment facilitates object-passing processes by providing services that include scheduling, controlling, remote class loading and deployment of user programs. Moreover, the run-time environment allows JOPI to be used on clusters and heterogeneous systems. Another feature in JOPI is that it is a pure Java implementation and it uses socket programming for communication. JOPI provides Java developers with a powerful tool to write compute-intensive applications and execute them on all available compute-resources regardless of the architectures or operating systems used.

In the rest of this paper, we first introduce in Section 2 some background and related work. In Section 3, we discuss object-oriented programming and the use of Java in parallel programming. Section 4 discusses JOPI, its characteristics, features, advantages, and disadvantages. We also compare usage of JOPI for parallel programming with that of MPI, socket programming, and RMI. Section 5 describes the architecture of JOPI and its run-time environment. Section 6 reports on the experiments conducted to evaluate the performance of JOPI. Finally, Section 7 concludes the paper.
2. BACKGROUND AND RELATED WORK

Java in its current state provides features and classes that facilitate distributed application development. However, the development process is usually very complex and time consuming. Java provides methods for socket programming, remote method invocation (RMI), object serialization and reflection [19] [4] that allow developers to write their own distributed applications. Some of these techniques, such as socket programming, are efficient but complex, while others, like RMI, are relatively simple to use for development, but less efficient than sockets. However, some effort is being invested to implement more efficient RMI such as in [13]. Using the available methods in Java, a daring programmer may write a parallel program, but the complexity of the task deters almost all from tackling this intricate task.

Message Passing Interface [12] is a library of routines provided for users who wish to write parallel and distributed programs. MPI-1 was developed for use mainly with FORTRAN and C and provides a number of library functions to exchange messages among processes. Using MPI for parallel programming is not trivial and requires full awareness of the parallelization issues and details of message exchange. On the other hand, this provides the programmer with high flexibility. MPI can be used to exchange messages containing one or more elements of the same data type. In addition, packed data elements of different lengths or types and structures can be exchanged, but this requires additional coding. Later, MPI-2 was developed as an extension of MPI-1 with additional functionality such as process creation and management, one-sided communications, I/O, and additional language bindings such as C++ bindings.

Object-Oriented MPI was introduced to provide C++ programmers with more abstract message-passing methods. A number of extensions were developed to provide object orientation for C++ and FORTRAN 90 such as OOMPI [20] [14], Charm++ [11] and ABC++ [3]. Many other extensions to languages such as Eiffel, Smalltalk and FORTRAN 90 were developed to provide parallelizing mechanisms to these languages [21].

Recently some effort has also been invested to provide MPI bindings for Java based on the MPI for Java (MPJ) draft specifications [5]. Most approaches rely on some of the features provided by Java while others used Java native interface (JNI) or Java-to-C interface (JCI) to link Java with MPI. In addition, many research groups have worked on providing parallel Java using other approaches and programming models. The paper [1] provides more information about many of these projects, their models and approaches.

3. OBJECT-ORIENTED AND PARALLEL PROGRAMMING IN JAVA

The object-oriented programming paradigm is well defined and widely used in applications development. Using objects allows for providing clean interfaces, thus facilitating object reuse and “plug and play” modules. In addition, object-oriented approaches are becoming useful for parallel programming.

Java is currently a greatly successful object-oriented language that truly supports object-oriented features and benefits. Java as described by its developers [6] is “A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language”. This description indicates the suitability of Java for many different types of applications such as client/server-distributed applications, web-based applications and server applications. With all the features available in Java, it was logical to build some extensions for Java to support parallel programming. The basic form can be considered at the level of socket programming and then in remote method invocation (RMI). These two approaches can easily support distributed applications, but require a considerable effort to build a parallel application. The details of the parallelization process, communication, synchronization, etc. pose a great hurdle in using these methods, as is, to create a parallel application. However, these details are general and are similar for any parallel application. Thus using the object-oriented approach in Java allows for creating some support environments and APIs to simplify the parallelization process for the application developer. Some effort was made to provide parallel capabilities in Java using available technologies such as RMI, JNI and others [1].

In addition, the machine independence of Java provides a unique opportunity to utilize multiple heterogeneous platforms to execute heterogeneous parallel applications. This can be made possible by providing the necessary mechanisms to schedule, deploy and control such applications in the heterogeneous system. Currently, the agent-based run-time environment [2] provided for JOPI can support such heterogeneous platforms.

One of the most important features that differentiates JOPI from the other systems studied is the use of objects as a the means of communication. Another difference is in the run-time environment that provides dynamic job deployment, secure remote access and parallel execution across heterogeneous platforms.

4. AN OVERVIEW OF JOPI

JOPI provides a pure Java implementation for parallel programming using an object-passing model on a distributed memory system. JOPI relies on the run-time environment to execute user parallel programs on multiple machines, on clusters, or heterogeneous systems. It provides an object-passing interface based on the same characteristics of message-passing and it is very similar to MPI, which provides users with a familiar interface and easier to learn methods. Message-passing (or object-passing) is a model of explicit parallel programming. The message-passing model has a number of characteristics [8] that also apply to the concept of object passing:

1. Multithreading: a message-passing program has multiple processes that have their own control and may execute different code.
3. Separate address space: each process has its own address space and exchanges information using special message-passing functions.
4. Explicit interaction: user must resolve all interaction issues such as communication, synchronization, and aggregation.
5. Explicit allocation: workload and data must be explicitly allocated to processes by the user.

In addition to the above characteristics, object-passing allows for logic (not just data) to be exchanged among processes. This feature provides a flexible, scalable, and easy way to use the parallel environment.

4.1 JOPI’s Features
Using JOPI, data is exchanged among different machines by means of objects. These objects may represent sub-problem objects, sub-solution objects, or update objects. This provides a number of advantages over MPI. First, the syntax of the communication methods is much simpler and easier to remember. Another advantage is that complex data structures can be exchanged easily in a single step. Furthermore, using objects allows the programmer to separate the problem solution from the parallelization process. For example, the programmer can write the classes that encapsulate all problem attributes and contain methods to solve a given problem in addition to methods to divide the problem into sub-problems and combine the partial solutions to form the final solution. Then in a different class the parallelization sequence can be given, deciding on the size of the sub-problems, how many processes to use and how they will cooperate. The parallelizing object can divide the problem object to get sub-problem objects. These objects will encapsulate all data related to the sub-problem and the logic to solve the sub-problem. Each sub-problem object is sent to a different machine for processing. The sub-result objects from different processes can be combined into a final result object.

Another advantage of object-passing is that a well-designed object-oriented parallel program can pass objects among its processes. There is no need for extra coding for converting an object presentation to a message to be transmitted, and then converting the message back to an object. Although standard Java provides object serialization for object transport, it does not provide optimizations and primitives to support parallel Java programming. JOPI extends the object serialization to be used for parallel environments. Objects can be sent point-to-point synchronously or asynchronously, or they can be broadcast to all parallel processes located in different machines.

In addition, the benefit of encapsulating data and methods in one object is that it makes reuse and maintenance easier. Object-passing model allows users to extend the features and advantages of the object-oriented concept to the communication process in parallel applications. Utilizing object-oriented concepts in the communication mechanism among parallel processes allows the advantages of reusability and ease of maintenance of the object-oriented system to be extended for parallel applications running in distributed memory system.

4.2 JOPI’s APIs
Using JOPI, users can initiate point-to-point communications, group communications and synchronization, in addition to a few other functions and attributes. The parallelization process is easily coded using the available JOPI APIs, which include:

1. myPID and nprocs are used to get process ID and number of processes used, respectively.
2. The static attributes ANY_SOURCE, ANY_TAG and ANY_CLASS are used by the receive method to receive objects from any source, having any tag value and/or of any class type, respectively.
3. Point-to-point communication provides methods to exchange objects between two processes. send and receive are used for blocking (synchronous) communication. Isend and Irecv are for non-blocking (asynchronous) communication.
4. Methods to check for the completion of non-blocking communication are testSend and testReceive, which check if Isend or Irecv has completed and return a Boolean flag. waitSend and waitReceive will check for Isend or Irecv completion and return true only when the operation is complete.
5. Broadcast is done using Broadcast method. The initiating process must specify the object to be sent and a group identifier. The receiving process needs to specify the group identifier only.
6. A group synchronization method is available to synchronize a number of processes at any given point in time. This is barrier and requires specifying a barrier ID and number of processes to synchronize.
7. Other methods are available such as print, which will direct all output to the root node, and close, which must be used at the end of the program to close the JOPI interface.

4.3 Comparing JOPI, MPI, RMI and Sockets
Some advantageous features provided by the object passing approach over message passing are:

1. It is easy to separate the problem solution from the parallelization process allowing users to change the solution or the parallelization method without affecting the other.
2. The separation reduces the complexity the development process and makes debugging and testing easier and faster.
3. The object-oriented approach allows for reusing classes created for a problem to solve other problems.
4. Compared to MPI, exchanging complex structures is easier with JOPI since structures can be enclosed within an object.

However, the object-passing approach has some disadvantages compared to the message passing approach:

1. Objects are not suitable for simple data type or small size data exchanges.
2. Java does not treat arrays like C, which makes array decomposition and reconstruction more difficult in Java.
3. Current implementations of Java are slower than C and Java compilers do not optimize code as C does.
Other available communication mechanisms provided by standard Java, such as socket programming and RMI provide facilities for communication in distributed systems at different levels of abstraction and with varying levels of user friendliness and efficiency. Compared to JOPI and MPI, RMI and socket programming do not explicitly support the parallel-programming model. However, it is possible, though very intricate, to build parallel applications using them. Table 1 contrasts the main features of JOPI, MPI, sockets and RMI in the context of parallel programming support.

Table 1 Summary comparison of JOPI, MPI, Sockets and RMI.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Object-Passing JOPI</th>
<th>Message-Passing MPI</th>
<th>Sockets &amp; I/O Streams</th>
<th>Parallel Java with RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of coding</td>
<td>Easy</td>
<td>Difficult for complex structures</td>
<td>Difficult</td>
<td>Difficult for parallel process</td>
</tr>
<tr>
<td>Ease of deployment</td>
<td>Easy</td>
<td>Easy</td>
<td>Medium. Need to manually deploy and start remote processes</td>
<td>Difficult. Need to create stub skeleton &amp; register remote objects</td>
</tr>
<tr>
<td>Size of code for sending complex structures</td>
<td>Small</td>
<td>Large – need a lot of codes</td>
<td>Medium. Need to define multiple I/O streams</td>
<td>Small</td>
</tr>
<tr>
<td>Process Naming</td>
<td>By Process ID</td>
<td>By Process ID</td>
<td>By IP address &amp; port no.</td>
<td>By service provided</td>
</tr>
<tr>
<td>Communication performance efficiency</td>
<td>Efficient for medium and large objects.</td>
<td>Efficient for small and medium messages.</td>
<td>Efficient for large objects</td>
<td>Less efficient than sockets</td>
</tr>
<tr>
<td>Communication Reliability</td>
<td>Reliable</td>
<td>Reliable</td>
<td>Unreliable, needs buffering mechanism</td>
<td>Reliable</td>
</tr>
<tr>
<td>Primitive type</td>
<td>Point-to-Point and group communication</td>
<td>Point-to-Point and group communication</td>
<td>Point-to-point and unreliable group communication</td>
<td>Point-to-point only</td>
</tr>
<tr>
<td>Asynchronous Comm.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes, in Java 1.4.0</td>
<td>No</td>
</tr>
<tr>
<td>Providing Scatter &amp; Gather</td>
<td>No, To be provided</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Support for parallel control services</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Dynamically Scheduling parallel Jobs</td>
<td>Provided by JOPI run-time system</td>
<td>Needs a third party tool.</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Heterogeneity support</td>
<td>Yes</td>
<td>Yes, but difficult</td>
<td>Yes, manual deployments</td>
<td>Yes</td>
</tr>
<tr>
<td>Code portability</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Code reusability</td>
<td>Easy</td>
<td>Difficult</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
<tr>
<td>Code maintainability</td>
<td>Easy, everything as objects</td>
<td>Difficult</td>
<td>Difficult</td>
<td>Medium</td>
</tr>
</tbody>
</table>

5. THE JOPI ARCHITECTURE
JOPI is based on a distributed memory model, using object-passing. The system has a number of components that collectively provide a parallel programming environment to the user. These components are: the Java object-passing interface (JOPI) class, the run-time environment and client services. In this section, we discuss these components and show the system’s main features.

5.1 The JOPI Class Library
JOPI is a class that provides the user with the interface for realizing object-passing. Java Users can use JOPI class to write parallel Java programs and compile them using the standard Java compiler. The generated class can be run on the JOPI run-time environment in distributed or cluster systems. The JOPI class is a multithreaded interface, which contains a receiving thread and some threads to support non-blocking operations. A receiving thread is created for each process to continuously wait for incoming objects and store them in a message queue that holds the received objects until they are needed by the process. This queue adds reliability to the communication process by ensuring all arriving objects are stored for later receive requests. The broadcast operation is implemented in a dynamic tree structure. When a JOPI thread receives a broadcast object, it should pass it to two other nodes in the system plus all the other JOPI threads on the same node. This allows for an efficient distribution of the broadcast object among all threads. In addition, this allows for better utilization of the internal resources in the machine.

Figures 1 and 2 show sample codes using JOPI for a primitive parallel matrix multiplication. The class Matrix (figure 1) is a regular class for matrix operations that is used to generate a matrix, multiply two matrices, partition a matrix into sub-matrices and update a matrix using the given sub-matrices. This class can be used immediately to generate and multiply matrices sequentially. However, to be able to use it for parallel matrix multiplication, we only need to make it serializable. The matrix_m class (Figure 2) contains the parallelization
mechanism. In this class we instantiate an object mp from the JOPI class to be used for parallelizing the program. Using the mp object, we invoke the necessary methods. The master process (process 0) is responsible for initializing the matrices, dividing one matrix into sub-matrices and distributing the first matrix and the appropriate sub-matrices to each process (including itself). When multiplication is complete, the master process will receive the result sub-matrices and update the original result matrix. All other processes will receive the matrices, perform the multiplication and send back the result sub-matrices. As shown, the Matrix class can be used in a sequential or parallel program. However, to facilitate the parallelization process the methods subMatrix and update were added. Using this same class, different parallelization techniques can be used. In addition, it is also possible to modify the Matrix class to provide additional functions or change the algorithms used without having to modify the parallelization class. Similarly, this approach can be used to develop larger and more complex applications. The main step is to have a well-engineered problem solution that can be easily adapted for a parallel solution either by modifying some of its methods or adding a few more methods to handle partitions and updates. This approach simplifies the development process and allows for easier maintenance and object reuse.

```java
// Parallel matrix multiplication program
// Main class – Parallelism Class
class matrix_m {
    public static void main(String[] args) {
        JOPI mp = new JOPI(args); // Instantiating a JOPI object
        A,B,C; // Declaring three matrix objects

        if( mp.myPID == 0 ) // Master thread section
            // Get Matrices A and B
            A = new Matrix("matrixA.dat");
            B = new Matrix( "matrixB.dat" );

        // Find Size of matrix partition based on no. of processes
        int length = B.col / mp.nprocs;

        // Send A and sub-matrix of B to each process
        for(i=1;i<mp.nprocs;i++)
            { // Define result matrix
                C = new Matrix( A.row , B.col );

                // Multiply A by first submatrix of B
                C.update(A.multiply(B.subMatrix(0,0,B.row-,length-1)));

                // Receive result sub-matrices from other
                // processes and update result matrix
                for(i=1;i<mp.nprocs;i++)
                    C.update((Matrix) mp.receive(mp.ANY_SOURCE,
                            mp.ANY_CLASS, mp.ANY_TAG));

            } // Other processes section
        else
            // Receive matrix A & sub-matrix B, calculate A x B
            A = (Matrix) mp.receive(0,mp.ANY_CLASS,1);
            B = (Matrix) mp.receive(0,mp.ANY_CLASS,2);
            C = A.multiply(B);
            mp.send(0,C,0); // Send sub-result C back to thread 0
            // ...}
```

5.2 The JOPI Run-time Environment

Software agents have been utilized for the JOPI run-time environment [2], which provides flexibility, scalability and ease of managing different resources. Agents reside on participating machines to deploy the parallel Java code, support its execution, and facilitate object-passing (see Figure 3). The agents contain many components to function effectively such as the Request Manager, which manages user job requests, the Resource Manager, which manages and maintains the system resources, and the Security Manager, which provides security measures for the system in addition to other components such as the Remote Class Loader and the Scheduler.

---

**Figure 1** Matrix class, used for matrix manipulation

```java
// Matrix class, must be persistent – Problem Class
class Matrix implements Serializable {
    public int col; // Number of columns
    public int row; // Number of rows
    private int orgCol; // Starting column (for sub-matrix)
    private int orgRow; // Starting row (for sub-matrix)
    public float[][] v; // Matrix Values

    // Initialize the Matrix from Data File
    public Matrix(String fileName) {
        // ...}
    }

    // Multiply method – multiply this matrix by matrix B
    // New result matrix is returned
    public Matrix multiply(Matrix B) {
        // ...}

    // Method to get sub-matrix
    public Matrix subMatrix(int fromRow, int fromCol, int toRow,
                            int toCol) {
        // ...}

    // Method to update this matrix using sub-matrix B
    public void update(Matrix B) {
        // ...}
}
```

**Figure 2** matrix_m class, used for parallelization steps
The system does not require manual class deployment or the existence of a shared file system. As part of the deployment, the agents move user classes to the target machines, convert them into threads and run the threads directly from local memory (on the remote machines). For high throughput, agents are designed to be multithreaded, where each thread will serve a client’s request. The communication mechanism between agents/clients is built directly on TCP/IP to make them more efficient.

Client services provide commands for the users to run the parallel programs and check some of the systems status information. These commands are supported by the agents residing on the machines participating in the system. Examples include pjava that is used to execute the parallel program, pingAgent that checks the status of the agents, and killJob that terminates a user job. When a parallel Java program is written, it is compiled using the regular Java compiler. Then a parallel execution file needs to be written, which includes the names of all classes in the program and the execution schedule. Alternatively, the user may also use Auto scheduler provided by the run-time environment. Finally, this parallel execution file is invoked using the pjava command. Figure 4 shows a sample parallel file for the example shown in figures 1 and 2. This file executes the matrix multiplication program on six remote processors selected automatically by the agent’s scheduler. The NoLocal option is used to force execution of the master process on a remote node. However, if the master process requires access to the local system resources such as the hard disk, then this option should not be used. The AutoSchedule can be used to allow the agents to select the fastest responding machines for executing the program. However, the user can manually select the machines by listing the agents names (on the machines) and the number of processes needed on each machine.

```
Mainclass matrix_m
Classes matrix_m.class Matrix.class
NoLocal
AutoSchedule 6
```

Figure 4 matrix.pj file, used to schedule and start the parallel Java program.

6. PERFORMANCE EVALUATION

Benchmark programs were written to evaluate the performance of JOPI and to compare it with MPI. All experiments, except those reported in 6.4, were conducted on Sandhills, a cluster of 24 Dual AMD 1.2 GHz processor nodes with 256 KB cache and 1 GB RAM per node. The cluster is connected via a 100 Mbps Ethernet network. All experiments for JOPI were conducted using standard Java virtual machine version sdk 1.3.1. The experiments were designed to measure the communication performance and the parallelization efficiency and compare it to MPI. In addition, an experiment was designed to show that JOPI, with its run-time environment, could support a collection of heterogeneous platforms.

6.1 Communication performance

Two Ping-Pong programs were written, one in C and MPI, another in Java with JOPI. These programs were executed on Sandhills starting with messages of size one byte up to four Megabytes. The round trip time, RTT, was registered and the effective bandwidth was calculated. Figure 5 shows a graphical view of the results gathered, which show that both MPI and JOPI perform best with large messages. JOPI shows a high overhead at sizes less than 64KB, but after that, it starts to match and eventually surpass the MPI performance. The major cause of low performance of JOPI at small object sizes is the serialization/deserialization overhead, which is more evident at small messages. However, this can be overcome by providing direct memory access primitives to exchange small messages of a specific data type instead of objects. The obtained results indicate that JOPI is most suitable for coarse grain communication where messages are large and less frequent.

![Figure 5 Communication performance of JOPI and MPI](image)

6.2 Matrix Multiplication

Dense matrix multiplication algorithm [7] was used to compare the performance of JOPI with MPI. The matrix size was 720x720. A matrix class was first developed to provide the methods needed to create, multiply, create a sub-matrix object, and combine sub-matrices into one matrix. A matrix object is instantiated for the first matrix and is sent to all processes. The
second matrix was divided into groups of columns (sub-matrices) each of which is an instantiation of the same matrix object and is then distributed to processes, including the root process. All programs used blocking point-to-point communication methods to ensure uniformity between the Java and C programs. Matrix elements are randomly generated floating point numbers. C programs were compiled with gcc and maximum optimization (-O3). From the measurements obtained (Figure 6), we observe the following:

1. JOPI achieved high speedup results. For example, at two processors, speedup was 1.89 and at four processors, it was 3.05.
2. Although JOPI is generally slower than MPI, it was possible to achieve much higher speedups using JOPI. This enables JOPI to get closer to MPI in speed as the number of processors increases. For example, for sequential versions, C is 2.52 times faster than Java, but at eight processors, C is only 1.74 times faster.
3. Using the optimized version of the C programs resulted in a large difference in performance between JOPI and MPI in terms of execution time. However, this approach is not fair for Java because Java does not do code optimization as C.
4. The parallelization overhead is high in both MPI and JOPI; however, JOPI shows less effect of this overhead with a larger number of processors.

![Figure 6 Speedup graph MPI and JOPI](image)

In addition, the process broadcasts the local minimum tour value found. This allows other processes to update their minimum value to speedup their search. In addition, asynchronous communication is used by the processes to report their results to the master process, while continuing to process other parts of the search tree. The results obtained (Table 2) show good speedup and efficiency results with growing number of processors and fixed problem size (22 cities). The results also indicate that many problems that require infrequent communication can be implemented efficiently using this system on a cluster.

![Table 2 Speedup & efficiency results for TSP](image)

<table>
<thead>
<tr>
<th># proc.</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec.)</td>
<td>33986.68</td>
<td>18255.84</td>
<td>9130.18</td>
<td>4634.58</td>
<td>3136.71</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.000</td>
<td>1.862</td>
<td>3.722</td>
<td>7.333</td>
<td>10.835</td>
</tr>
<tr>
<td>Efficiency</td>
<td>100.00%</td>
<td>93.08%</td>
<td>93.06%</td>
<td>91.67%</td>
<td>90.29%</td>
</tr>
</tbody>
</table>

6.4 Heterogeneous system Experiment

This experiment shows the capabilities of JOPI and its run-time environment to execute parallel applications on heterogeneous platforms with minimum user involvement. The matrix multiplication algorithm [7] used here divides the first matrix into blocks of rows and the second into blocks of columns. This is done using the same matrix class used in 6.2, but here the distribution policy differs. Each process gets a block of rows and a block of columns. When work is done, the process sends its results to the master process and takes new blocks to process. Here a matrix of 1800x1800 floating numbers was used, with a stripe size of 300 rows or columns. The method used also shows the benefit of using the object-oriented approach in parallelizing the problem. Although we needed to change the parallelization policy to have a load balancing algorithm, we were still able to use the same matrix class created in 6.2 without any changes. However, this can only be beneficial if the original class was well engineered to facilitate reuse. In addition, the separation of the parallelization details from the problem allowed us to easily change the distribution mechanism and the processes loads.

The machines used were the Linux nodes (Sandhills), RCF [16], and a Windows2000 server (Table 3). To fairly compare performance, the sequential running time for the program was measured on each platform. As a first order approximation, speedup is calculated individually for each participating platform by dividing the sequential time on that platform by the parallel execution time on the system. However, the speedup calculated using the fastest machine in the system gives a rough estimate of the parallel program performance. Due to the communication overhead imposed by the distributed setting of the cluster, the system is most suitable for applications that have high computation to communication ratio, or coarse grain parallelism, as expected.

The results obtained from this experiment show that it is feasible to use different platforms collectively to execute a parallel application. This is mainly due to the use of the agents technology to facilitate application deployment and
the portability and machine independence of Java. Finally, it is apparent that Java is improving and this means that it is becoming more feasible to use Java for high performance computing. In addition, the total execution time is slightly higher than the ideal execution time for 12 processors on the fastest machine (Sandhills), which is 398/12 = 33.17 seconds. This increase in time is mainly due to having a much slower machine (RCF) in the collective. Generally, using JOPI it is possible to utilize all available resources without having to deal with the differences between platforms. The ability of utilizing a heterogeneous environment with varying architectures and specifications also provides an opportunity to further optimize the performance of parallel applications. Hence, tasks can be distributed among machines based on their requirements and suitability to the platform used. This means that if some tasks require heavy communication, they can be assigned to a multi-processor machine, while tasks that are relatively independent and require less communication can be assigned to a cluster.

7. CONCLUSION
JOPI is an object-passing interface for Java that is based on the general characteristics of the message-passing model. JOPI supports passing objects between processes, which simplifies programming and provides more control. JOPI’s object-passing interface fully accentuates the benefits of the object-oriented approach in application development, in the context of parallel and distributed computing. The ability to separate the problem from the parallelization process, encapsulate data and methods for exchange, and easily incorporate complex structures for communication are the principal advantages of this approach. The JOPI run-time environment is implemented in pure Java for portability and supported by the use of software agents that provide flexibility and control. The experiment results show that JOPI performs very well, particularly with larger object sizes. JOPI in its current form is most suitable for coarse grain parallelism and performs best with applications that have high computation to communication ratios. In addition, the experiments revealed that JOPI is truly portable and can be used across multiple platforms without any additional effort.

8. ACKNOWLEDGMENTS
This project was partially supported by a National Science Foundation grant (EPS-0091900) and a Nebraska University Foundation grant, for which we are grateful. We would also like to thank other members of the secure distributed information (SDI) group [18] and the research computing facility [16] at the University of Nebraska-Lincoln for their continuous help and support.

9. REFERENCES
http://java.sun.com/docs/books/tutorial/index.html

Table 3 The results of the heterogeneous experiment.

<table>
<thead>
<tr>
<th>Cluster Specifications</th>
<th>Operating system</th>
<th>Procs used</th>
<th>Time seq.</th>
<th>Speedup for each platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 nodes, dual AMD AthlonMP, 1.2 MHz, 256KB cache, 1GB RAM per node, Fast Ethernet</td>
<td>Linux</td>
<td>6, including root</td>
<td>398</td>
<td>9.258</td>
</tr>
<tr>
<td>RCF: Origin 2000, 32 processors, 250 MHz, 4MB cache, 8GB RAM</td>
<td>IRIX</td>
<td>3</td>
<td>1549</td>
<td>36.030</td>
</tr>
<tr>
<td>CSNT-TS: 3 CPUs, Intel x86 700MHz, Total 1.5GB RAM</td>
<td>Windows2000 adv. server</td>
<td>3</td>
<td>419</td>
<td>9.746</td>
</tr>
<tr>
<td>Total Processors</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel time</td>
<td>42.992</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


[14] OOMPI web page: www.mpi.nd.edu/research/oompi


[18] Secure Distributed Information Group at UNL. http://rcf.unl.edu/~sdi

