JOPI: a Java object-passing interface

Jameela Al-Jaroodi¹, Nader Mohamed¹,*,†, Hong Jiang² and David Swanson²

¹Department of Electrical and Computer Engineering, Stevens Institute of Technology, Burchard 216, Castle Point on Hudson, Hoboken, NJ 07030, U.S.A.
²Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE 68588-0115, U.S.A.

SUMMARY
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 a Message Passing Interface (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. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: Java; heterogeneous systems; cluster; parallel programming; object-passing; object-oriented systems

1. INTRODUCTION
Increasing demand for high performance and powerful computing resources has led to great advances in the area of distributed and parallel computing. In addition, clusters and heterogeneous networked systems have become increasingly more popular. This direction has 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 increasing steadily and many
research groups have started exploring the possibilities of using Java for multi-processor systems. Due to Java’s machine independence, a program can execute 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, along with an introduction to Java object-passing interface (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, thus providing a clear set of application program interfaces (APIs) to be used for parallel programming in Java. This model utilizes the object serialization primitives available in Java to facilitate the process. Compared with the message-passing interface (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. The use of objects for exchanging complex structures and large data sets among parallel and distributed processes results in significantly reduced program code size. Furthermore, using object passing enables the programmer to separate the problem solution from the parallelization process, which simplifies program 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 applications. Moreover, the run-time environment allows JOPI to be used on clusters and heterogeneous systems. Another feature of JOPI is that it is a pure Java implementation and uses socket programming for communication. JOPI provides Java developers with a powerful tool to write compute-intensive applications and execute them on all available computer 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 the usage of JOPI for parallel programming with that of MPI, socket programming, and remote method invocation (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 form 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, RMI [1]; object serialization and reflection [2,3] that allow developers to write their own distributed applications. Some of these techniques, such as socket programming, are efficient but complex, while others, such as RMI, are relatively simple to use for development, but less efficient than sockets. However, some effort is being invested to implement more efficient RMI [4]. 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.

On the other hand, parallel programming models such as message passing and distributed shared memory have been available for other languages for some time. The MPI [5] 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. Nevertheless, 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, with 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 (e.g. C++ bindings). In addition, 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 [6,7], Charm++ [8] and ABC++ [9]. Many other extensions to languages such as Eiffel, Smalltalk and FORTRAN 90 were developed to provide parallelizing mechanisms to these languages [10].

To introduce some of the above features into Java, recently some effort has been invested to provide MPI bindings for Java based on the MPI for Java (MPJ) draft specifications [11]. Most approaches rely on some of the features provided by Java while others use Java native interface (JNI) or Java-to-C interface (JCI) to link Java with MPI. In addition, other approaches and programming models have been used by many research groups to provide parallel Java capability. The study in [12] provides more information about many of these projects, including the different models and approaches employed. Our approach in JOPI is unique in that it uses objects as the means of communicating data and logic among processes, while preserving the similarity to MPI in program constructs.

3. OBJECT-ORIENTED AND PARALLEL PROGRAMMING IN JAVA

The object-oriented programming paradigm is well defined and widely used in application development. The use of objects provides clean interfaces, thus facilitating object reuse and ‘plug and play’ modules. In addition, object-oriented approaches are becoming useful for parallel programming.

Java is a successful object-oriented language that truly supports object-oriented features and benefits. Java as described by its developers [13] 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 RMI. These two approaches can easily support distributed applications, but require considerable effort to build a parallel application. The details of the parallelization process, communication, synchronization, etc., pose a great hurdle in using these methods to create a parallel application. However, these details are general and common in all parallel applications. 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 [12].

In addition, the portability and machine independence of Java provides a unique opportunity to utilize multiple heterogeneous platforms simultaneously to execute parallel applications since an
application written in Java need only be compiled once then deployed on any system with a Java Virtual Machine (JVM). 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 \[14,15\] 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 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. The separation between the two components (the programming model interface, JOPI and its run-time support environment) is also a unique feature. This approach allows for flexible development, efficient utilization of resources and easier expansions. In addition, the run-time support is capable of supporting other programming models, such as the distributed shared memory, that require a similar set of services provided by this environment.

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 on heterogeneous systems. It provides an object-passing interface based on the same characteristics of the message-passing model 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 \[16\] 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

With 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. The 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 a distributed memory system.

4.2. Comparing JOPI, MPI, RMI and sockets

In this section, we describe the differences between JOPI and other parallel and distributed application development models and tools available, such as MPI, RMI and sockets. Some of the advantageous features provided by the object-passing model (JOPI) over the message-passing model (MPI) are:

1. it is easy to separate the problem solution from the parallelization process, thus allowing users to change the solution or the parallelization method without affecting the other;
2. the separation reduces the complexity of the development process and makes debugging and testing easier and faster;
3. the object-oriented approach allows for the reuse of 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 Java object-passing model has some disadvantages compared to the message-passing model:

1. objects are not suitable for simple data types or small size data exchanges;
2. Java does not treat arrays like C does, which makes array decomposition and reconstruction more difficult in JOPI;
3. current implementations of Java are slower than C and Java compilers do not optimize code as C does.
Table I. Summary comparison of JOPI, MPI, sockets and RMI.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Object-passing Message-passing Sockets and Parallel Java with RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JOPI</td>
</tr>
<tr>
<td>Ease of coding</td>
<td>Easy</td>
</tr>
<tr>
<td>Ease of deployment</td>
<td>Easy</td>
</tr>
<tr>
<td>Size of code for sending complex structures</td>
<td>Small</td>
</tr>
<tr>
<td>Process naming</td>
<td>By process ID</td>
</tr>
<tr>
<td>Communication performance efficiency</td>
<td>Efficient for medium and large objects. Reliable</td>
</tr>
<tr>
<td>Communication reliability</td>
<td>Reliable</td>
</tr>
<tr>
<td>Communication type</td>
<td>Point-to-point and group communication</td>
</tr>
<tr>
<td>Asynchronous communication</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for parallel control services</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamically scheduling parallel jobs</td>
<td>Provided by JOPI run-time system</td>
</tr>
<tr>
<td>Heterogeneity support</td>
<td>Yes</td>
</tr>
<tr>
<td>Binary code portability</td>
<td>Yes</td>
</tr>
<tr>
<td>Code reusability</td>
<td>Easy</td>
</tr>
<tr>
<td>Code maintainability</td>
<td>Easy, everything as objects</td>
</tr>
</tbody>
</table>
The 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, although very intricate, to build parallel applications using them. On the other hand, JOPI also differs from MPI in some ways such as the use of objects for communications, thus the ability to exchange data and logic. Table I contrasts the main features of JOPI, MPI, sockets and RMI in the context of parallel programming support.

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: JOPI class and APIs, the run-time environment and client services. In this section, we discuss these components and show the system’s main features.

5.1. JOPI's APIs

With JOPI, users can initiate point-to-point communications, group communications and synchronization, in addition to other functions and attributes. The parallelization process is easily coded using the available JOPI APIs, which include the following.

1. *myPID* and *nprocs* are used to get process ID and number of processes used, respectively. These values are assigned by the system such that each process will have a unique identifier within the application. As in MPI, the main (master) process is assigned the number 0 and the rest of the *n* processes are given the number 1 through *n* − 1.

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. These attributes provide flexibility to the processes to exchange objects of different classes or tags, or receive objects from an unidentified source.

3. Point-to-point communication provides methods to exchange objects between two processes. *send* and *receive* are used for blocking (synchronous) communication. *Isend* and *Ireceive* are for non-blocking (asynchronous) communication. The *send/Isend* methods require specifying the receiving process ID, the object to be sent and a tag value. In addition, the *Isend* will return a request identifier. The *receive/Ireceive* methods require specifying the sending process ID, the class of the object to be received and the tag value and return the received object or the request identifier.

4. Methods to check for the completion of non-blocking communication are *testIsend* and *testIreceive*, which check if *Isend* or *Ireceive* has completed and return a Boolean flag. *waitIsend* and *waitIreceive* will check for *Isend or Ireceive* completion and return only when the operation is complete. These methods use the request identifier provided by the *Isend or Ireceive* methods. In future implementations, the test and wait operations can be merged to be similar to MPI.
5. Broadcast is done using the \texttt{Bcast} 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. This group identifier is specified by the user and can be used to differentiate multiple broadcast messages and to selectively broadcast messages to specific subgroups.

6. A group synchronization method, \texttt{barrier}, is provided to synchronize a number of processes at any given point in time. It requires specifying a barrier ID and number of processes to synchronize. The barrier ID is required to allow processes to distinguish different barriers issued simultaneously.

7. Other methods of JOPI include \texttt{print}, which follows the formatting conventions used in Java and directs all output to the master node, and \texttt{close}, which must be used at the end of the program to close the JOPI interface.

5.2. The JOPI class library

JOPI is a class that provides the user with the interface for realizing object passing. Java users can use the JOPI APIs defined in this class to write parallel Java programs and compile them using the standard Java compiler. The generated class can be executed on the JOPI run-time environment in distributed or cluster systems. The JOPI class is implemented using multithreads, including 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, deserialize them and store their references in a queue for the received objects references until they are needed by the process. This queue adds reliability to the communication process by ensuring all arriving objects are stored for later \texttt{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 to ensuring better utilization of the internal resources in the machine.

Figures 1 and 2 show sample codes using JOPI for a primitive version of the parallel matrix multiplication. The class \texttt{Matrix} (Figure 1) is a regular class for matrix operations that is used to get 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 directly to generate and multiply matrices sequentially. In order to use it for parallel matrix multiplication, we only need to make it serializable. The \texttt{matrixjn} class (Figure 2) contains the parallelization mechanism. In this class we instantiate an object \texttt{mp} from the JOPI class to be used to parallelize the program. Using the \texttt{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 \texttt{Matrix} class can be used in a sequential or parallel program. However, to facilitate the parallelization process the methods \texttt{subMatrix} and \texttt{update} were added. These two classes provide sub-matrices of a defined size to be used by the different processes. Thus, they only utilize small parts of the memory for the sub-matrix required. 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
// Parallel matrix multiplication, multiplies two dense matrices
// 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 1. Matrix class, used for matrix manipulation.

complex applications. The main step is to have a well-engineered problem solution that can be easily adapted for a parallel solution by either 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.

5.3. The JOPI run-time environment

Software agents have been used for the JOPI run-time environment [15], which provides flexibility, scalability and ease of management of different resources. Agents continuously reside on participating machines to deploy the parallel Java code, support its execution and facilitate object passing (see Figure 3). In addition, agents are multithreaded so they could handle multiple users and multiple user jobs simultaneously. The agent startup and operation mechanisms are discussed in detail in [14]. The agents contain many effective function components such as the Request Manager, which manages
// Parallel matrix multiplication program
// Main class – Parallelism Class
public class Matrix
{
    public static void main(String[] args)
    {
        JOPI mp = new JOPI(args); // Instantiating JOPI object
        Matrix 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++)
            {
                mp.send(i,A,1);
                mp.send(i,B.subMatrix(0,length*i,B.row-1,length*(i+1)-1 ),2);
            }

            // 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++)
            {
                mp.send(i,A,1);
                mp.send(i,B.subMatrix(0,length*i,B.row-1,length*(i+1)-1 ),2);
            }
        }

        else // Other processes section
        {
            // 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
        }
    }

    ...
    }
}

Figure 2. Matrix class, used for parallelization steps.
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.

The scheduler is an independent component that provides process and resource scheduling for the submitted parallel applications. The current implementation deploys a basic scheduler that uses processor response time to an execution probe (a test program executed simultaneously on all machines) as its base to build a schedule, consequently allocating user processes to the fastest responding processors. This scheme is based on the assumption that the processors responding faster are the ones that are less loaded and can accommodate more processes.

The run-time support also provides monitoring and control mechanisms such that the agents can remotely deploy processes, control the execution environment and enforce the security execution modes. In addition, the system provides the user with a direct monitoring and control interface such that they can monitor their jobs and control their execution.

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 the 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 \texttt{pjava} that is used to execute the parallel program, \texttt{pingAgent} that checks the status of the agents and \texttt{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...
FIGURE 4. The matrix2.pj file, used to automatically schedule and start the parallel Java program on six processors.

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

FIGURE 5. The matrix2.pj file, provides a user-defined schedule for allocating and executing the processes in the parallel Java program on six processors.

```
Mainclass matrix
Classes matrix.class Matrix.class
NoLocal
Agent01 2
Agent03 1
Agent12 3
```

needs to be written, which includes the names of all classes in the program and the execution schedule. Alternatively, the user may also use the AutoScheduler 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 AutoScheduler 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. Figure 5 shows an example of how processors are selected manually by the user. The run-time environment automatically assigns names for all the agents residing on the machines such as Agent01, Agent02, Agent03 etc. The user can use these names to specify the machines and number of processors needed. For example, in Figure 5 the user selects to run two processes on the machine running Agent01, one process on the machine running Agent03 and three processes on the machine running Agent12. More information about the run-time support environment can be found in [15].

6. PERFORMANCE EVALUATION

Benchmark programs were written to evaluate the performance of JOPI and to compare it with that of MPI. All experiments, except for those reported in Section 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 JVM version sdk 1.3.1. The experiments were designed to measure and compare the
communication performance and the parallelization efficiency of JOPI versus 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, and another in Java with JOPI. These programs were executed on Sandhills, starting with messages of size 1 KB up to 4 MB. The round trip time, $RTT$, was registered and the effective bandwidth was calculated. Figure 6 shows a graphical view of the results gathered, which indicates that both MPI and JOPI perform best with large messages. JOPI incurs a high overhead at sizes less than 64 KB, 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 pronounced 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.

6.2. Matrix multiplication

Dense matrix multiplication algorithms from [19] were used to compare the performance of JOPI with MPI. The matrix size was $720 \times 720$. A matrix class was first developed to provide the methods needed to create and multiply sub-matrices, create a sub-matrix object and combine sub-matrices into one matrix. A matrix object was instantiated for the first matrix and was sent to all processes. The second matrix was divided into groups of columns (sub-matrices), each of which was an instantiation of the same matrix object and was 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 (Figures 7 and 8), we observe the following.

1. JOPI achieved high speedup, calculated as the execution time of the application on a single processor over the execution time of the application on \( n \) processors. 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 enabled JOPI to get closer to MPI in speed as the number of processors increased. 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 in our experiments resulted in a large difference in performance between JOPI and MPI in terms of execution time, giving the advantage to the C program. However, this approach may not be fair for Java because Java does not perform code optimization as C does. The execution times for the un-optimized version were much worse (longer) for C and thus very close to those of Java.

4. The parallelization overhead is high in both MPI and JOPI; however, the impact of this overhead on JOPI is shown to diminish with a larger number of processors.

In a second set of experiments for PMM, asynchronous communication was used to overlap the computation with the communication in the PMM operations. The results obtained, in terms of execution time and speedup (see Figures 9 and 10), are generally better than the first experiments for both MPI and JOPI mainly because of the overlap between the computations and communication. Moreover, the sequential version of the C program was also altered to the new algorithm (using the asynchronous communications), which resulted in an improved execution time using a single processor. In addition, the observed trends of speedup increases remained very similar to the results of the first set of experiments.

6.3. The Traveling Salesman Problem (TSP)

The solution algorithm chosen for the TSP is based on branch-and-bound search [17]. This problem required the use of many of the JOPI primitives to implement an efficient load-balanced solution. A class was created to handle the search mechanism, given the proper range in the problem and a bounding value (minimum current tour value). Another class for the result tour was also created to be used by the different processes to report their results to the master process. The minimum tour value and the search range were also represented as objects. Broadcast was used to distribute the original problem object to the processes. Each process performed the search within its range, reported the local best tour to the master process and requested a new range for the next search. In addition, the process
broadcast the local minimum tour value found. This allowed other processes to update the minimum value to speed up their search. In addition, asynchronous communication was 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 (see Figure 11) show good speedup results as compared to the ideal case (speedup = sequential time/number of processors), and with growing number of processors and fixed problem size (22 cities). The efficiency can be calculated by dividing the speedup obtained by the total number of processors used and we were able to achieve over 90% efficiency for up to 12 processors. The results also indicate that many problems that require infrequent communication can be implemented efficiently using this system on a cluster.
Table II. List of machines used in the experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Platform description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSNT</td>
<td>3 CPU, Intel x86 700 MHz, 1.5 GB RAM. OS: Windows 2000 adv. server</td>
</tr>
<tr>
<td>RCF</td>
<td>Origin 2000, 32 CPU, 250 MHz, 4 MB cache, 8 GB RAM. OS: IRIX 6.5.13</td>
</tr>
<tr>
<td>SH</td>
<td>Cluster, 24 nodes, dual 1.2 MHz AthlonMP, 256 KB cache, 1 GB RAM per node. OS: Linux</td>
</tr>
</tbody>
</table>

6.4. The heterogeneous system experiments

This set of experiments shows the capabilities of JOPI and its run-time environment to execute parallel applications on heterogeneous platforms with minimum user involvement. The performance measures are generally good; however, experiments involving more than one machine also relied on the campus network for inter-machine communications. This resulted in an added communication overhead since the network is shared by many users and usually is loaded. Nevertheless, the gains are still considerably good. In addition, we include the performance evaluation model developed and discussed in [18] to evaluate these experiments. The speedup and efficiency in this model depend on the power of a machine $M_i$, defined by the amount of work it can complete in unit time, and the power weight $W_i(A)$ of machine $M_i$ with application $A$, which defines the amount of work $M_i$ can complete relative to the fastest available machine in the system. Thus,

$$W_i(A) = \frac{\min_{j=1}^{m} T(A, M_j)}{T(A, M_i)}$$  \hspace{1cm} (1)

where $T(A, M_j)$ is the elapsed execution time for application $A$ on machine $M_j$ and $m$ is the total number of machines in the heterogeneous system $HS$. Based on the general definition of speedup, the expression for speedup $SP$ of the heterogeneous system is derived as

$$SP = \frac{\min_{j=1}^{m} T(A, M_j)}{T(A, HS)}$$  \hspace{1cm} (2)

However, when machines execute one or more tasks simultaneously the speedup does not reflect the true performance measure. Therefore, we calculate the efficiency as

$$E = \frac{SP}{\sum_{j=1}^{m} W_j n_j}$$  \hspace{1cm} (3)

where $n_j$ is the number of processors (or tasks) used in machine $M_j$. The product $n_j W_j$ gives an estimate of the power weight when $n_j$ processors are used to execute $n_j$ tasks of the application on the machine $M_j$. The experiments used the standard JVM sdk 1.3.1 and were executed on homogeneous and heterogeneous clusters (see Table II).

6.4.1. TSP

As in Section 6.3, the solution for TSP is based on a branch-and-bound search approach [17]. TSP was executed on a heterogeneous cluster consisting of nodes from SH, RCF and CSNT (see Table III).
In this case, the initial calculations of the power weight ($PW$), based on Equation (1), show that SH is the fastest machine and that CSNT has around 0.94 power relative to SH. Although the speedup increases as more processors are added, the true effect of the different processor speeds is not apparent by the speedup measure alone. On the other hand, the efficiency values, calculated using Equation (3), show this effect clearly. The results show that the efficiency at six processors (two, two and two, for SH, CSNT and RCF, respectively) is around 80.7% compared with 62% using the normal calculations of speedup divided by the number of processors, while at six processors (four and two for SH and CSNT, respectively) it was around 77.8% compared with 76% in the normal calculations. This difference reflects the nature of the machines used and how much each machine contributes in the processing. When the two values for efficiency are close, then it reflects that all processors in the heterogeneous collection have similar power and contribute relatively equally in the processing. However, the large difference indicates that there are processors that have considerably low or high processing power compared with the others, thus signifying the level of heterogeneity of the system executing the application. More details about the performance model and analysis can be found in [18].

6.4.2. Matrix multiplication (MM)

In this set of experiments, a dense MM algorithm [19] was employed with load balancing mechanism and synchronous point-to-point communication. A matrix of size $1800 \times 1800$ floating numbers was used, with a stripe size of 300 rows or columns. This experiment also showed the benefit of using the object-oriented approach in parallelizing the problem. Although we needed to change the parallelization policy to achieve load balance, we were still able to use the same matrix class created in Section 6.2 without any changes. In addition, the separation of the parallelization details from the problem allowed us to easily change the distribution mechanism and the processes’ loads.

The results are shown in Table IV, where the power weight $PW$ of the machines show that SH has the best response time and thus is taken as the reference for $PW$ calculations using Equation (1) for RCF and CSNT. In addition, the speedup was calculated using Equation (2); however, the speedup...
Table IV. Performance measurements for MM on a heterogeneous system.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Elapsed time (s)</th>
<th>Relative speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>280.83</td>
<td>0.944</td>
<td>100.00</td>
</tr>
<tr>
<td>1</td>
<td>807.65</td>
<td>0.328</td>
<td>100.00</td>
</tr>
<tr>
<td>1</td>
<td>265.19</td>
<td>1.00</td>
<td>100.00</td>
</tr>
<tr>
<td>3</td>
<td>100.93</td>
<td>2.627</td>
<td>92.7448</td>
</tr>
<tr>
<td>3</td>
<td>416.49</td>
<td>0.656</td>
<td>64.6401</td>
</tr>
<tr>
<td>3</td>
<td>100.24</td>
<td>2.645</td>
<td>88.1888</td>
</tr>
<tr>
<td>6</td>
<td>55.708</td>
<td>4.760</td>
<td>81.612</td>
</tr>
<tr>
<td>12</td>
<td>42.992</td>
<td>6.168</td>
<td>63.9152</td>
</tr>
<tr>
<td>16</td>
<td>34.591</td>
<td>7.666</td>
<td>50.1866</td>
</tr>
</tbody>
</table>

The efficiency, calculated using Equation (3), on the other hand, takes into account the differences in machine performances and hence better represents the performance. For example, with 12 processors in SH we achieved 74% efficiency as opposed to the 63% efficiency achieved with a collection of six processors from SH, three from CSNT and three from RCF, where RCF is much slower than the other two machines. In addition, comparing the efficiency calculated here with the efficiency calculated by dividing speedup by the total number of processors (regardless of their contribution), which in this case amounts to 51%, we find that the latter underestimates the efficiency, especially when there are more slow machines in the system.

6.4.3. Discussion

The distributed machines used here are connected by a general-purpose local area network that is shared among a large number of parallel machines (e.g. clusters and/or SMPs), servers and personal computers on campus. This has resulted in high load on the network and high transfer delays. It is therefore possible to achieve much better performance if the machines are connected using a high-speed network or dedicated connections. In general, due to the communication overhead imposed by the distributed setting of the machines, the system is most suitable for applications that have high computation to communication ratio, or coarse-grain parallelism, as expected. However, the system provides a unique opportunity to simultaneously utilize any collection of existing machines instead of being confined to using each machine separately. This is mainly due to the use of the agents’ technology to facilitate application deployment and the portability and machine independence of Java. Generally, with JOPI it is possible to utilize all available resources without having to deal with the differences between platforms and to optimize the performance of the parallel applications. Hence, tasks can be distributed among machines based on their requirements and suitability to the platforms 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 the passing of 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 experimental results show that JOPI performs very well, particularly with larger object sizes. JOPI in its current form is most suitable for coarse-grain parallelisms 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.

ACKNOWLEDGEMENTS

This project was partially supported by a National Science Foundation grant (EPS-0091900), a Nebraska University Foundation grant and a UNL Academic Priority Grant, for which we are grateful. We would also like to thank other members of the secure distributed information (SDI) group [20] and the research computing facility [21] at the University of Nebraska–Lincoln for their continuous assistance and support. In addition, we would like to extend our gratitude to the referees of this special issue, who provided invaluable comments and suggestions that helped improve the quality and presentation of this paper. A preliminary version of this paper appeared in the Proceedings of the Joint Java Grande–ISCOPE Conference held on 3–5 November, 2002, Seattle, WA, U.S.A.

REFERENCES