University of Nebraska-Lincoln
Computer Science and Engineering

JOPI: Java Object-Passing Interface

User's Guide

Date	**Version**	**Prepared by**
February 20, 2004	1.0.0 (Beta)	Jameela Al-Jaroodi & Nader Mohamed

Table of Contents

1. Introduction

Clusters require system and software services that can support the distributed architectures and provide transparent and efficient utilization of the multiple machines available. In addition, Java is becoming a good option for developing high performance and distributed applications on clusters. We introduce an infrastructure for a parallel programming environment in Java. The system provides Java programmers with the necessary functionality to write object-passing parallel programs in clusters. The main features of JOPI are: (1) It is suitable for clusters and distributed heterogeneous systems, (2) It utilizes the object-oriented programming paradigm for parallel programming thus simplifying the development process, and (3) It is most suitable for large scale applications and applications with low communication-to-computation ratio.

The Java object-passing interface (JOPI) provides an MPI-like interface that can be used to exchange objects among processes. Using objects to exchange information is advantageous because it facilitates passing complex structures and enables the programmer to isolate the problem space from the parallelization problem. Software agents were used to provide the necessary functionality on the participating processors. Users need not deal with the mechanisms of deploying and loading user classes on the heterogeneous system nor to deal with scheduling, controlling, monitoring, and running user jobs. In addition, users need not deal with managing the system resources. Since this system is written completely in Java, it is portable and can allow run programs in parallel across multiple heterogeneous platforms. A number of experiments were conducted using this system to measure its performance. The results gathered show that it is possible to gain good performance using this system and that future potential for enhancements and optimizations are feasible.

This system provides an infrastructure for high performance computing in Java (parallel Java) for clusters or heterogeneous systems. Software agents were used to provide the necessary functions that support the JOPI on clusters and distributed environments. Some of the benefits of using agents are: portability, expandability, flexibility, security, and resources management. Using the Java Object Passing Interface (JOPI) provided, the programmer will be able to write parallel programs that can run on multiple nodes of a cluster or network where the agents reside. In addition, the system is written completely in standard Java and can be used on any machine that has a Java virtual machine (JVM).

This document is a user guide for the system. It will provide information about how to use the environment and how to write parallel Java programs using the JOPI class. Since the interface provided is similar in some ways to standard MPI, we decided to include some information about MPI first. The third section introduces the different methods and attributes of the JOPI class and how to use them when writing the parallel programs. Then the next section talks about the environment and the different commands available to run the parallel Java programs.

2. MPI Vs. JOPI

Object passing like message passing is a model of explicit parallel programming with the following characteristics:

1) Multithreading: a message-passing program has multiple processes that have their own control and may execute different code.
2) Asynchronous parallelism: processes execute asynchronously and require explicit synchronization.
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.
6) logic Exchange: Object-passing allows processes to exchange codes (not just data), which provides a flexible, scalable, and easy way to use the parallel environment.

The Message Passing Interface (MPI) 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 language. It provides a number of library functions to exchange messages between processes. It provides functions for point-to-point communication, group communications, synchronization and others. Using MPI for parallel programming is not trivial. It requires the programmer to be aware of almost all of the parallelization issues and details of the messages to be exchanged. On the other hand this provides the programmer with high flexibility. MPI-1 can be used to exchange messages containing one data type such as integers, float or char. In addition it allows packing and unpacking of data of different lengths or types. It also can allow passing arrays and structures, but this requires a lot of coding.

Later MPI-1.2 and MPI-2 was developed as extensions of MPI-1 additional functionality such as process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, and additional language bindings such as C++ bindings. Object-Oriented MPI (OOMPI) was introduced a few years ago with the main concern to provide C++ programmers with more abstract message passing methods.

JOPI provides similar functions using objects as a means of exchanging data and logic. This provides users with the ability to encapsulate not just the data, but also the logic that uses that data into a single object that can be then sent to other processes to be used there.

The object-oriented paradigm preserved in JOPI allows users to simplify the process of writing parallel programs. The programmer here has the flexibility of using the available JOPI methods to exploit the different properties to optimize the programs. Compared to the standard MPI, JOPI allows the programmer to exchange objects instead of predefined data elements. This means that the programmer need not worry about the type or size of the message to be sent. All that is required is to have all the data encapsulated within an object.

When using JOPI, the programmer needs to identify the problem and write the necessary code to solve it sequentially. Then methods can be added to handle problem partitioning for a parallel solution and to reconstruct the solution from the partial solutions computed by the different processes. The parallelization process can then be easily done using the available methods and the JOPI methods.

To support JOPI's library functions, we included the following features in the agent-based run-time infrastructure:

1) Providing mechanisms to load user programs onto the remote JVMs of the cluster nodes and the machines used
2) Managing cluster resources and scheduling user jobs
3) Providing security measures to protect the user programs and the machines used
4) Providing job and thread naming to facilitate simultaneous multi-user and multi-job executions and proper process identification
5) Providing user commands and control operations that facilitate job submission, monitoring and control

3. JOPI API

This part contains all the functions that provide the user with the Java parallel programming object-passing interface. Using the functions in this class the user can send, receive and broadcast objects. Please keep in mind that Java is case-sensitive thus JOPI is also case-sensitive. Therefore, make sure to follow the proper syntax and letter-case for the methods and attributes of JOPI.

The objects that are used in send and receive must be defined to implement Serializable (i.e. they must be serializable for the system to be able to move them to remote processors). Example:

```
class List_Class implements Serializable
```

In addition, other attributes to identify the process ID and number of processes are available. Before using the methods available in JOPI class, like with any other Java classes, the user must define an instance of the JOPI class for the program.

```
JOPI  mp = new JOPI(String[] args)
```

Where *args* is the array of arguments passed to main and *mp* is the name of the instance created from JOPI.

The following gives details of the methods and attributes in the JOPI class, which the user can now use to write a parallel Java program. We will use the instantiated JOPI object *mp* (defined above) in all our following examples for the sake of clarity. However, users are free to select the names of their objects and the user-defined classes and attributes.

Attributes

A number of attributes are defined in the JOPI class that support parallel programming using the object-passing model. The explicit communication requires the programmer to identify individual processes and other supporting values. First the user needs to know the process identification number of each processes used and the total number of processes in the program. In addition, some general attributes are needed to issue general operations that do not require specific values.

a. *mp.myPID*. This field is an integer and contains the current process identification number, which can be used in other methods such as *send*, *receive*, *Isend*, *Ireceive*, etc.

b. *mp.nprocs*. This field is an integer and contains the number of processes in the user job (program). This is necessary for workload distribution among participating processes and for group communication and synchronization operations.

c. *JOPI.ANY_TAG*. This is a static attribute that can be used as a wild card with the receive operations such that it will accept an object with any tag value.

d. *JOPI.ANY_CLASS*. This is a static attribute that can be used as a wild card with the receive operations such that it will accept an object of any class type.

e. *JOPI.ANY_SOURCE*. This is a static attribute that can be used as a wild card with the receive operations such that it will accept an object from any source.

The static attributes are useful for receive operations where the tag, class type, and/or source are not statically determined before run-time. For example if a process needs to accept the first response that arrives from any one of multiple processes then it cannot specify the source. As a result it uses the *JOPI.ANY_SOURCE* to accept any incoming object.

Methods

The JOPI class provides a number of methods for different operations in parallel and distributed programming. Many of these support explicit exchange of objects among the processes in synchronous and asynchronous modes.

1. Synchronous (blocking) point-to-point operations

a. *send*. This method sends an object from one process (thread) to another. When it is used, it blocks the sending process until the message has reached its destination. The user needs to specify the target process ID (*pid*), the object to be sent, and the *tag*. As in standard MPI, the tag is used to pair send and receive operations to make sure the receiving process receives the correct object.

```
mp. send(int pid, Object obj, int tag)
```

Where *pid* and *tag* are integers. Example: Assume we have an object called greetings defined from a class Greetings_Class, then the following line will send the object *greetings* to process number *2* with tag value *0*.

```
mp.send(2, greetings, 0)
```

b. *receive*. This is a method used by a process to receive an object sent by another process. The user can specify where the object is coming from (*pid*), The class name (a string argument) that the object belongs to (*className*), and the tag (*tag*). The user also has the choice to use *JOPI.ANY_TAG*, *JOPI.ANY_CLASS*, or *JOPI.ANY_SOURCE* to receive an object with any tag value, of any class type, or from any other process, respectively. This method blocks the receiving process until the receive operation is complete and the object is in the receiver's buffer. It always returns an object (the received object). A cast for the class type of the object received must be given to get the correct object format. Usually if *className* is given it should be the same as *class_type*.

```
Obj_ref = (class_type) mp.receive(int pid, String className, int tag)
```

This will receive an object from process *pid* with class type *className* and tag value of *tag*. The user can also use *JOPI.ANY_SOURCE, JOPI.ANY_CLASS*, and/or *JOPI.ANY_TAG* in place of *pid, className*, and/or *tag* respectively.

Examples:

```
Obj_ref = (class_type)  mp.receive(JOPI.ANY_SOURCE,
                                   JOPI.ANY_CLASS,  JOPI.ANY_TAG)
```

This allows the receive method to accept any incoming message from any process, of any class type and with any tag value.

```
Obj_ref = (class_type) mp.receive(4, JOPI.ANY_CLASS, 0)
```

This receives an object of any class type from process number *4* and with tag value *0*.

```
greetings = (Greetings_Class)  mp.receive(0,"Greetings_Class", 0)
```

This will receive the object *greetings* sent by process *0*.   (see the send example)

2. **Synchronous (blocking) group operations**

   a. *Bcast*. This is a group communication method that allows one process to send an object to all other processes. When broadcast is used, all processes will issue the Bcast command. The sender will specify the object to be sent and all other processes will act as receivers to get that object. Broadcast is blocking so the sending object will block until the object is successfully sent, while each receiving process will wait until the object is received in its buffer before continuing. There are two variations of the broadcast method:

   ```
   mp.Bcast(Object obj, int group_no)
   ```

   Used by the sending process to broadcast the object *obj* to the group identified by *group_no*. *obj* is a string and *group_no* is an integer.

   ```
   Obj_ref = (class_type) mp.Bcast(int group_no)
   ```

   Used by all other processes to receive the broadcast message the group number, *group_no*, here must match the one specified by the sending process. This method will return the received object. To receive the correct object it must be cast into the proper class type using *class_type*.

   b. *barrier*. This is a synchronization method used to synchronize a group of processes. When the barrier method is invoked with a given number of processes, any process reaching this method will suspend its execution and wait until the specified number of processes reaches the barrier then they all can resume execution at the same time.

   ```
   mp.barrier(int id, int no_procs)
   ```

   Here the *id* (integer) is used to identify the barrier since many barriers may be issued in a program. *no_procs* (integer) indicates the number of processes that must reach the barrier before all of them can go on.

3. Asynchronous (non-blocking) point-to-point operations

a. *Isend*. This method (like *send*) is used by a process to send an object to another process. Here it allows the process to issue a send request and go back to its computations without waiting for the send operation to complete. Later the process can use the *testIsend* or *waitIsend* methods to check if the send operation was completed. The method *Isend* will return a request identification number *requestID* to be used for verifying the send completion. When a process issues an *Isend* operation, it should not modify the object sent before making sure the operation has completed using the *testIsend* or *waitIsend* methods. Otherwise, such changes may affect the results of the execution since the data sent may not be the one intended.

RequestID = mp.Isend(int pid, Object obj, int tag)

b. *Ireceive*. This method (like *receive*) is used to receive an object from another process. Here it allows the process to issue a request to receive an object and then continue with its processing. Later the process can issue a *testIreceive* or *waitIreceive* function to find if the receive request was fulfilled. This *Ireceive* will return a request identification number *requestID* that can be used to check the request.

RequestID = (class_type) mp.Ireceive(int pid,
String className, int tag)

This will issue a request to receive an object identified by *className* from a sender identified by *pid* with a given *tag* value. As in blocking receive a cast must be given for the object received. In addition, the static attributes *JOPI.ANY_SOURCE*, *JOPI.ANY_CLASS*, and *JOPI.ANT_TAG* can be used just like with the *send* method. When *Ireceive* is used, the object will not be available for use by the process until the receive operation is completed. Therefore, it is necessary to check for its completion before using the object.

c. *testIsend*. This is an asynchronous (non-blocking) method to check if an asynchronous send operation (*Isend*) has completed. It uses the *requestID* to identify the send operation to be checked. When *testIsend* is issued the system checks if the request for send identified by the *requestID* has completed. It will return true if completed and false if not.

SendFlag = mp.testIsend(int requestID)

This will check the send operation and return true if the requested send operation has completed and false otherwise. *SendFlag* must be defined as Boolean.

d. *testIreceive*. This is an asynchronous (non-blocking) method to check if an asynchronous receive operation (*Ireceive*) has completed. It uses the *requestID* to identify the receive operation to be checked. When *testIreceive* is issued the system checks if the request for receive identified by the *requestID* has completed. It will return the received object if completed and *Null* if not.

Obj_ref = (class_type) mp.testIreceive(int requestID

This will check the receive operation and will return the received object in the *Obj_ref* if the receive operation has completed and will return *Null* otherwise. Here also a cast needs to be given for the object received. The programmer must check the object to know if the receive operation has completed (test for *Null*) before trying to use that object.

e. *waitIsend*. This is a synchronous (blocking) method to check for asynchronous (non-blocking) send operation completion. *waitIsend* will not return until the send operation is completed. As in *testIsend*, it uses *requestID* to identify the operation to be checked.

   mp.waitIsend(int requestID)

   This will check the send operation and only return if the requested send operation has completed. An *Isend* followed immediately by *waitIsend* will be equivalent to using the *send* method.

f. *waitIreceive*. This is a synchronous (blocking) method to check for the completion of asynchronous (non-blocking) receive operation. *waitIreceive* will not return until the receive operation is completed and the object is in the receive buffer. On return this method will return the received object. As in *testIreceive*, it uses *requestID* to identify the operation to be checked.

   Obj_ref = (class_type) mp.waitIreceive(int requested)

   This will block the process until the object requested is received and will return that object. Here also a combination of *Ireceive* followed immediately by *waitIreceive* is equivalent to a synchronous *receive* method. A cast for the object received must be given.

4. Support operations

a. *receiveSource*. This method is used after a successfully completed receive operation to identify the source of the object received. This is mainly useful if the receive operation used the *JOPI.ANY_SOURCE* attribute. This operation can follow the *receive*, *Ireceive*, *Bcast* (at the receiving end), successful *testIreceive* (if object returned is not Null), and *waitIreceive* methods. This method will return an integer value representing the sending processor number. The result returned will correspond to the last successfully completed receive operation only. The method cannot provide information about other past receive operations.

   Source_no = mp.receiveSource()

b. *receiveClass*. This method is used after a successfully completed receive operation to identify the class type of the object received. This is mainly useful if the receive operation used the *JOPI.ANY_CLASS* attribute. This operation can follow the *receive*, *Ireceive*, *Bcast* (at the receiving end), successful *testIreceive* (if object returned is not Null), and *waitIreceive* methods. This method will return a string which provides the class name the received object belongs to. The result returned will correspond to the last successfully completed receive operation only. The method cannot provide information about other past receive operations.

Class_name = mp.receiveClass()

c. *receiveTag*. This method is used after a successfully completed receive operation to identify the tag value associated with the received object. This is mainly useful if the receive operation used the *JOPI.ANY_TAG* attribute. This operation can follow the *receive*, *Ireceive*, *Bcast* (at the receiving end), successful *testIreceive* (if object returned is not Null), and *waitIreceive* method. This method will return an integer value representing the tag value associated with the object. The result returned will correspond to the last successfully completed receive operation only. The method cannot provide information about other past receive operations.

Tag_no = mp.receiveTag()

d. *print*. This method is similar to the regular print methods in Java except that the output is directed to the client station where the user job was initiated. This allows for all screen output to be available on the user console. The format used in this method is exactly the same as in the regular print method in Java.

mp.print(String msg)

e. *close*. This method is used at the end of the program to end the JOPI interface. Using this method will ensure that the program will correctly exit and all processes related to it will be terminated.

mp.close()

Example

In the figure below we illustrate JOPI's API usage (Please note that the line numbers are only for illustration purposes and that they are not part of the program). This program simply sends a message "Hello World" to all processes that will receive and print that message along with the process identification number. In line 10 an instance of the JOPI class called *mp* is instantiated. In line 11 *mp.myPID* is used to get the current process ID. Line 15 sends the string object to other processes (String in Java is a serializable object; therefore it can be directly sent using JOPI send method). In line 20 the other processes issue a *receive* to receive the object. In this case we used *JOPI.ANY_TAG and JOPI.ANY_CLASS* to accept any object with any tag value. In many other cases the user may need to explicitly specify the class type and/or the tag.

Notice that it is possible to use broadcast method in this example to distribute the greeting message to the processes. To do that all you need to do is replace lines 14 and 15 with the following:

mp.Bcast(greeting, 1);

which will broadcast the string object to all other processes and line 20 by:

Greeting = (String) mp.Bcast(1);

mp.barrier(int id, int No_procs)

which will first allow each process to receive the broadcast message and then wait until all processes reach this point (making sure all received the message) before printing the message and exiting.

```
import java.io.*;
import java.util.*;
import java.net.*;

public class Hello_1                        // Main class
{
    public static void main(String[] args)
    {
        int j;
        String greeting;
        JOPI mp = new JOPI(args);           // Instantiating a message passing object
        if( mp.myPID == 0 )                 // process 0
        {
          greeting = "Hello World";
          for(j = 1; j < mp.nprocs; j++)
              mp.send(j, greeting, 0);
          mp.print(greeting+" I am process number "+mp.myPID);
        }
        else                                //  other processes
        {
            greeting = (String) mp.receive(0,JOPI.ANY_CLASS,JOPI.ANY_TAG);
            mp.print(greeting+ " I am process number "+mp.myPID);
        }
        mp.close();                         // Close message passing object
    }
}
```

Example. A parallel JOPI program to print a greeting message from all processes.

4. Compiling and Running Programs

When the program is complete, it should be compiled using *javac* like any other regular Java program. Example, if the program name is *prog1.java* then compile as follows:

```
javac Hello_1.java
```

To run the program after successful compilation, the user needs to define a parallel job file. This file is given a .pj extension and should contain the following information (see the example below; please note that the line numbers are only for illustration purposes and that they are not part of the file):

1. The name of the main class in the program preceded by the keyword *MainClass* (line 1)

2. The keyword *Classes* followed by the names of all classes used in the program including the main class (line 2).

3. The number of processors needed and how they are scheduled. The schedule for the processes can be left to the system (using *AutoSchedule* option), in this case the user need only specify the number of processes needed to run the program (line 3). Example:

```
AutoSchedule  6
```

This will make the system automatically create a schedule for six processes.

On the other hand, the user can specify his/her own schedule by listing the agents and number of threads to run on each agent. Example:

```
Agent agent1   2
Agent agent2   1
Agent agent3   2
```

Agent is a keyword and *agent1*, *agent2* and *agent3* are the names of the agents on the machines used. These names are defined in the system (in the agents list file) and the user can get a list of these names by using the command *pingAgent*. The integer number following the agent name specifies how many threads should run on that machine. This sequence (may replace line 3) means that two threads will run on the machine that has *agent1* and the machine that has *agent3* on it and one thread will run on the machine that has *agent2* on it. The total number of threads (processes) here is 5.

The system allows multiple users to run multiple jobs at the same time. To properly manage these jobs, each job has multiple levels of identification, starting with a unique *job ID* assigned by the system. The *user ID* and the *program name* further distinguish different jobs. Within each job, *thread IDs* are used to identify the remote threads of the job. To protect user applications and the systems executing them, two modes of execution are utilized: (1) Agent mode, which allows a process to access all available resources on local and remote machines, and (2) user mode, which allows access to the CPU and memory for execution only on remote machines. With the security modes in place, the user processes have full access to resources on their local machine (where the user job was initiated). In addition, user processes have very limited access to all remote machines' resources (since they are running in user mode). To provide users with access to necessary resources for their application, the root (master) process is forced to execute on the user's local machine. However, the user has the option to override this setting and execute the root process on a remote machine, thus limiting its access to system resources.

The user can specify if root process should or should not run on the local machine (line 4 in the example). *NoLocal* means that root process should not run on the local machine. If it is not included, then the root process will run on the local machine.

When JOPI is first installed, the administrator may bypass the security restrictions thus allowing user processes to have full access to all recourses on all the machines used. If this is true, then the *NoLocal* option will not have much effect on resource access.

```
1       MainClass Hello_1
2       Classes Hello_1.class
3       AutoSchedule 4
4       NoLocal
```

Example: hello.pj file for the Hello_1.java program shown earlier. In this case the main class is Hello_1 and it is the only class in the program. Automatic scheduling was selected and all processes will run on remote machines.

When the .pj file is ready, the user can run the program using the following command:

```
pjava filename.pj
```

Example:

```
pjava hello.pj
```

The client services class is the link between the user and the agents. It accepts requests from the user, encapsulate the job and the necessary information into a request object, and passes it to the agent. It also handless all the commands issued by the user such as *pingAgent* and *listThreads*. A number of commands are available for the users through the client services:

a. *pjava filename.pj* ➔ This command is used to initiate a user job (start execution of the parallel program). This will deploy the user processes on the requested number of processors and start running. When the user runs this command the client services takes the user job along with all the necessary information to start that job and puts it in an object. This object is then passed to the agents for deployment and execution. As shown earlier, the schedule used for running the user job can be specified by the user or left to the scheduler to generate. For this command to work a user file **.pj* must be created to identify the user program classes and the schedule for running the program.

b. *pingAgent [all] | [agentname]* ➔ This is used to list the available agent(s) and their status (*Active* or *Inactive*) on the system. If an agent name is given the status of that agent is checked and displayed. The *all* option will make the command check all the agents and display the full list of agents. Not specifying any option will default to *all*. The status of an agent is either *Active* indicating that it executing and ready to accept user jobs or *Inactive* indicating that it is down and not available for executing user jobs.

c. *listThreads [all] | [agentname]* ➔ This is used to list all active threads on the agent(s). The user can specify a specific agent or all agents to be checked. The default option for this command is *all*. This command provides the following information about each thread:
    i. *Agent*: the agent on which the thread is running on
    ii. *JobID*: the identification number of the job this thread belongs to
    iii. *User*: the user this job belongs to
    iv. *Program*: The name of the main class (program) that created this thread.
    v. *Pid*: the identification number of the thread
    vi. *R. Time*: the amount of time passed since the thread started execution.

d. *testAgents* ➔ Used to report some performance features about the machines where the agents are running. This command activates a small program on all agents and measures the amount of time it takes each agent to complete that program. This gives a relative measurement of how much load (or how fast) the agent can execute programs. This is currently used by the scheduler to identify the fastest available agents to schedule the jobs accordingly. It is also available for the users for testing purposes.

e. *killJob jobID* ➔ This is a command to terminate a given user job (like the kill command in UNIX). The user needs to specify the *jobID* (can be found using *listThreads* command) with the command. When a job is killed, all the threads that belong to it will be terminated on all the machines originally executing this job.

5. Error Messages

When compiling a program that uses JOPI, the syntax errors and standard Java errors will all be identified by the Java compiler. However, there are other parts that are checked by the JOPI run-time environment and may generate a different set of errors. We used unique identifiers for these errors starting with two characters indicating the type of the error followed by a three digit number for the error. Short error messages follow the identifier for the user's information. The following table lists the error messages and the possible actions that can be done to resolve them. The errors are of three types:

1. PJxxx are errors concerning the .pj file. Generally these errors can be resolved by checking the .pj file and making some corrections.

2. UPxxx are errors related to the user program. In most cases the exception messages would help in identifying the source of the error.

3. SYxxx are system related errors. Many of these errors may require the help of the JOPI administrator to resolve.

Error Codes	Display Messages	Action
PJ001	Agent <agent id> is not defined in the agents list file or inactive	1. Check the agents and their status (active/inactive) using pingAgent 2. Check agent names in your .pj file and make sure they are spelled correctly and only active agents are used
PJ002	Parallel Java file <file id> can not be read	1. Check .pj file name 2. Make sure the file is accessible and in the right location (verify the path settings)
PJ003	Unknown command <command line> in .pj file	1. Check spelling and syntax of the command indicated
PJ004	Class <class name> could not be loaded	1. Make sure the class name is included correctly in the .pj file 2. Verify the class name spelling and case in the .pj file
PJ005	Class file <file name> does not exist	1. Make sure the class names listed in the .pj file are the same as the ones used in your program 2. Make sure the class names are correct (names in .pj file match the names in the source code) 3. Make sure the class is accessible and in the correct location. 4. Check the path settings to your files

Error Codes	**Display Messages**	**Action**
SY001	Agents list file <file name> can not be read	1. Make sure the agents file is available in its designated location (physical file is calle d agents) 2. Make sure the path settings to the agents list file is correct 3. Contact the system administrator if the file is not available
SY004	Directory file <directory file> does not exist	1. Contact admin to verify the existence of this directory and make sure it is available and in the correct location
SY005	Timeout in send operation	1. Thread unable to send a message to destination before timeout. 2. Check network connectivity between nodes 3. If persistent, report problem to admin. to check.
SY006	Receiving Thread Error	1. Report problem to admin. to check. However, admin may not be able to solve this immediately.
UP001	Error in running user threads (Java run-time error)	1. Check your source code (in most cases there may be some exception messages displayed that can help pinpoint the errors)

6. Example JOPI Programs

Here we provide a full example of parallel Java programs written using JOPI. The code can be easily extracted and compiled as a Java program. The .pj examples given here are specific to the system we are using and may require changes (mainly the agent names and number of processors used).

Summation of numbers in a list

Although not a very useful application, the following is a complete parallel program that calculates the sum of the elements of a list. The class *listClass* defines the methods and attributes needed to create, divide and manipulate the list. The main class is responsible for the parallelization process. Process 0 is the master process and it uses the methods in *listClass* to divide the list into sub-lists, sends the sub-lists to other processes, calculates its own sub-list and then receives and adds the sub sum values received from the other processes and reports the final result. Each process including the master process will calculate the sub sum of part of the list, print the partial result and send it back to process *0*. The program also calculates the time of execution using the system *time* method. JOPI can only send and receive objects; therefore when an integer value was to be sent it was defined using the Java *Integer* class. This way the *subSumResult* variable becomes an object of the class *Integer*, which is serializable, and can be used by JOPI. Following is the full program code followed by two versions of the job execution file and the steps of compiling and running the program.

```
/**********************************************************************/
// The source code for the example program list1.java
import java.io.*;
import java.util.*;
import java.net.*;

public class list1            // Main class
{
     public static void main(String[] args)
     {
          int MAX_SIZE = 6000;          //Maximum size of the list
          int RANGE_FROM = 1;           //Minimum value of an element
          int RANGE_TO = 10;            //Maximum value of an element
          int i, j, load, sumValue;
          long startTime=0,endTime=0,diff=0;
          JOPI mp = new JOPI(args);       //Instantiating a JOPI object
          Integer subSumResult;          //Defining an Integer object
          listClass fullList, partList;  //defining a list objects
          if( mp.myPID == 0 )             // process 0
               {
                 fullList = new listClass(MAX_SIZE, RANGE_FROM, RANGE_TO);
                 load = MAX_SIZE / mp.nprocs;  //Size of sublists
                 mp.print("Process load = "+load);
                 startTime = System.currentTimeMillis();
```

```
                  for(j=1; j<mp.nprocs; j++)   //Repeat for all processes
                  {
                    //Make a sublist of the original list
                    partList = fullList.subList(j*load, (j*load)+load-1);
                    //Send the sublist to process j
                    mp.send(j,partList,0);
                  }
                  //Process 0 also works on a sublist
                  partList = fullList.subList(0, load-1);
                  sumValue = partList.sum();
                  mp.print("Process "+mp.myPID+" sub sum = "+sumValue);
                  for (j=1; j < mp.nprocs; j++)  //Repeat for all processes
                  {
                    //Receive sublist sum and add to overall sum sumValue
                    subSumResult =(Integer) mp.receive(j,JOPI.ANY_CLASS,1);
                    sumValue += subSumResult.intValue();
                  }
                  // Register end time and calculate and print total time
                  endTime = System.currentTimeMillis();
                  diff = endTime - startTime;
                  mp.print("List sum = "+sumValue);
                  mp.print("Time taken = "+diff);
               }
          else     // Other processes
               {
                   //Receive sublist from process 0
                   partList = (listClass) mp.receive(0, JOPI.ANY_CLASS, 0);
                   //Calculate and send the sum to process 0
                   int subSum = partList.sum();
                   mp.print("Process "+mp.myPID+" sub sum = "+subSum);
                   subSumResult = new Integer( subSum );
                   mp.send(0, subSumResult, 1);
               }
          mp.close();          // Close message passing object
     }
}

class listClass implements Serializable   // Problem class
{
     public  int size;
     public  int[] list;
     public listClass(int tsize)
     {
         size = tsize;
         list = new int[tsize];
     }

     public listClass(int tsize,int fromNo,int toNo)
     {
         int i;
         size = tsize;
         list = new int[tsize];
         Random r = new Random();
         for(i=0;i<size;i++)
            list[i] = r.nextInt(toNo-fromNo+1) + fromNo;
     }
     public listClass subList(int fromPos,int toPos)  // Sub-Problem method
     {
         listClass t = new listClass(toPos-fromPos+1);
         System.arraycopy(list,fromPos,t.list,0,toPos-fromPos+1);
         return (t);
     }
     public int sum()     // Problem Solution method
     {
         int i;
         int result = 0;
         for(i=0;i<size;i++)
             result += list[i];
         return( result );
     }
}
/*************************************************************************/
```

To compile the program:

$ javac list1.java

The contents of lista.pj:

```
MainClass list1
Classes list1.class listClass.class
AutoSchedule 5
NoLocal
```

To run the program:

$ pjava lista.pj

Output:

```
=============================================
Reading the parallel java job file
Loading user classes
Implementing job schedule
Starting Job ID 20351 with 5 threads
Running 2 threads at agent agent5
Running 2 threads at agent agent2
Running 1 threads at agent agent1
=============================================
[0]: Process load = 1200
[1]: Process 1 sub sum = 6426
[2]: Process 2 sub sum = 6664
```

```
[3]: Process 3 sub sum = 6672
[0]: Process 0 sub sum = 6616
[4]: Process 4 sub sum = 6790
[0]: List sum = 33168
[0]: Time taken = 36
```

A different run of the same program:

The contents of listb.pj:

```
MainClass list1
Classes list1.class listClass.class
Agent agent1  2
Agent agent2  2
Agent agent5  1
NoLocal
```

To run the program:

```
$ pjava listb.pj
```

Output:

```
=============================================
Reading the parallel java job file
Loading user classes
Starting Job ID 87243 with 5 threads
Running 2 threads at agent agent1
Running 2 threads at agent agent2
Running 1 threads at agent agent5
=============================================
[0]: Process load = 1200
[1]: Process 1 sub sum = 6657
[2]: Process 2 sub sum = 6630
[3]: Process 3 sub sum = 6735
[0]: Process 0 sub sum = 6553
[4]: Process 4 sub sum = 6654
[0]: List sum = 33229
[0]: Time taken = 40
```

The first six lines of the output are general information to show the user what is happening and where the program threads are going to execute. The rest of the lines are the result of the *mp.print* statements in the program. The number between the square brackets is added to show which process initiated the print statement.

Acknowledgements

This project was partially supported by a National Science Foundation grant, Nebraska University Foundation grant, and the Priority Initiative in Simulation and Modeling grant. This project was supervised by Dr. Hong Jiang and Dr. David Swanson, UNL.

Please note: If you find any errors or typos, please send an e-mail to jopi@cse.unl.edu

Index

Bold Page number is most significant mention of keyword