MPJ

Message Passing Interface

In **J**ava for AgentTeamwork

Distributed Systems Laboratory
University of Washington, Bothell
Computing and Software Systems

Implemented By: Zhiji Huang
Debugged and Documented By: Etsuko Sano
Faculty Advisor: Munehiro Fukuda
March 17, 2006

Table of Contents

page

1. Introduction

MPI (Message Passing Interface) which is the original concepts of Message Passing Java (MPJ) supports communications for distributed programs. It allows programmers to create an environment with parallel programming and shared memory. MPI is able to use some programming languages, FORTRAN, C/C++, and Java, for the implementations. The current implementations in Java (mpiJava) are actually Java wrappers around native C code. However, it has some disadvantages with the portability and is not suitable to the concept of AgentTeamwork [Refer to 2.1.1.]. In other words, the original MPI has no check-pointing feature.

Therefore, the AgentTeamwork project has developed its own version of Message Passing Java (MPJ) as a middleware between the user program and various communications protocols. Currently, MPJ supports GridTcp sockets as well as Java sockets. The objective is to provide a set of communication functionalities supporting distributed computing. The most remarkable function in MPJ is a snapshot algorithm. It will take snapshots using some computer nodes over the Internet during the operations.

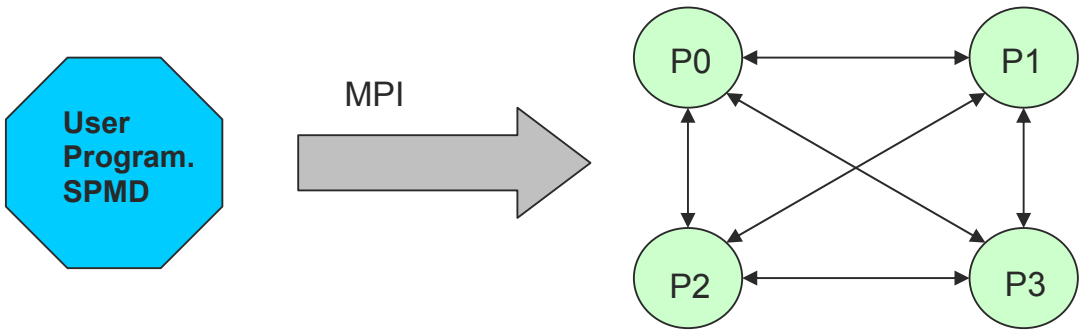


Figure1: The MPI data flow as a distributed program

This report describes the design, implementation, and performance results of the project.

2. Algorithm and Design

2.1. AgentTeamwork

MPJ is used on top of the AgentTeamwork system. The UWB Distributed Systems Laboratory has been developing the AgentTeamwork grid-computing middleware. AgentTeamwork coordinates remote grid-computing jobs using mobile agents. A user requests AgentTeamwork for some computing nodes. AgentTeamwork manages the resources for better performance and fault tolerance automatically.

Figure2: The basic concept of AgentTeamwork

A mobile agent is assigned to each process and moves to a low-loading machine. It also monitors each process and takes its periodical execution snapshot supported by *user program wrapper* and *GridTcp*. If a machine is crashed, the agent recovers the system with the latest snapshot. Thus the system can restore broken processes involved in the same job. The key feature in MPJ is to allow those agents to migrate or to resume a user program to a new idle computer if their current computer are overloaded or even powered off.

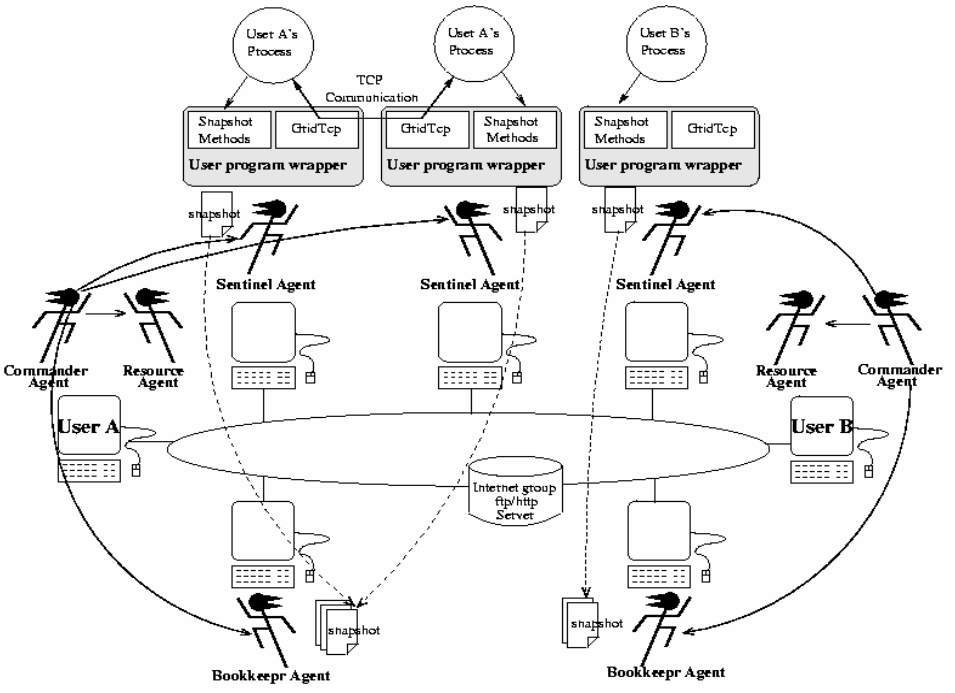


Figure3: AgentTeamwork Snapshot Algorithm

2.2. MPJ

Table1 shows the main structure of AgentTeamwork and MPJ:

<table>
<tr><td colspan="3">User Applications in Java</td></tr>
<tr><td>mpiJava</td><td colspan="2">MPJ API</td></tr>
<tr><td rowspan="4">Java Socket</td><td>mpjrun</td><td>User Program Wrapper</td></tr>
<tr><td>mpiJavaSocket</td><td>mpiJavaAteam</td></tr>
<tr><td rowspan="2">Java Socket</td><td>GridTcp</td></tr>
<tr><td>AgentTeamwork</td></tr>
<tr><td colspan="3">Java Virtual Machine</td></tr>
<tr><td colspan="3">Operating Systems</td></tr>
<tr><td colspan="3">Hardware</td></tr>
</table>

Table1: Regular MPI and the main frame of MPJ Project

AgentTeamwork's MPJ distinguishes two different sockets. One is mpiJavaSocket [*mpiJava-S*] using a regular Java Socket, and the other is mpiJavaAteam [*mpiJava-A*] that uses a check-pointing error-recoverable GridTcp Socket for the message passing.

2.2.1. **mpiJava-S** – Java Socket

mpiJava-S uses the regular Java Socket to implement MPJ functions. For convenience, there is a class which automatically opens SSH windows, executes commands, and creates processes at once. The class is called *mpjrun* and helps less typing for command executions.

2.2.2. **mpiJava-A** – GridTcp

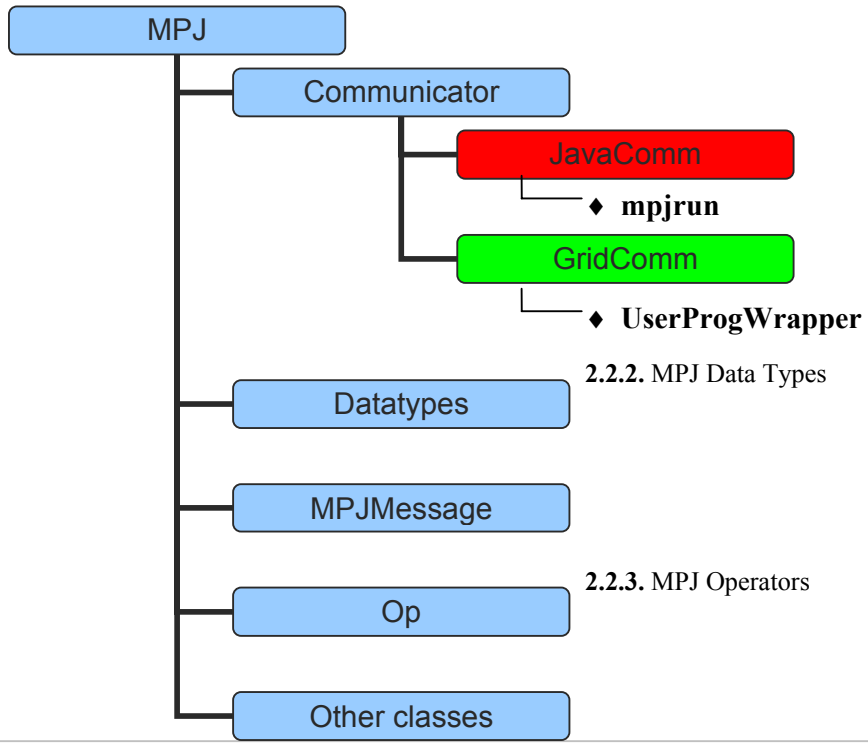
mpiJava-A extends TCP by adding message-saving and check-pointing features. It automatically saves messages and performs a checkpoint (or takes a snapshot) of a program execution. It allows programs to recover from errors, if their current computing node crashes.

2.3. MPJ Architecture

2.3.1. Class Diagram

The major classes shown in Diagram1 are the implemented classes of MPJ.

Diagram1: Class Hierarchy in MPJ



MPJ is the main application. It contains a communicator of either JavaComm or GridComm, depending on the type of socket used. It also contains various data types supported by MPJ. In addition, MPJ provides initialization and finalization methods for the network connections.

JavaComm and GridComm hold Java sockets and GridTcp sockets, respectively. These two classes do not provide any major functionality other than maintaining the input and output streams of their respective sockets.
Communicator is the class that provides the primary communications capabilities of MPJ. As of the writing of this report, Communicator contains the major functions like Send(), Recv(), Barrier(), Bcast(), Pack() and Unpack().

MPJMessage is a wrapper around each message received by the Recv() functions. It holds the message's status and the actual message itself.
The various Datatype subclasses provide serialization and deserialization of their respective types for Communicator.
Various Operation classes are wrappers around operations that are implemented in the Datatype subclasses.

2.3.2. MPJ Data Types [Datatype]

Datatype has nine major data types and provides serialization and deserialization services. In addition, each data type calls and implements Op functions inside the class.

Data Type	***Description***
MPJBool	Boolean (true/false)
MPJByte	Byte
MPJChar	Character
MPJDouble	Double
MPJFloat	Float
MPJInt	Integer
MPJObject	Object
MPJShort	Short

Table2: Data types in MPJ

2.3.3. MPJ Operators [Op]

Op has 12 different operation types. Valid arguments for the op parameter are the following:

Op Name	***Description***
MPJBAND	**B**itwise **And**
MPJBOR	**B**itwise **Or**
MPJBXOR	**B**itwise E**x**clusive **Or**
MPJLAND	**L**ogical **And**
MPJLOR	**L**ogical **Or**
MPJLXOR	**L**ogical E**x**clusive **Or**
MPJMAXLOC	**Max**imum and **Loc**ation of Maximum
MPJMAX	**Max**imum
MPJMINLOC	**Min**imum and **Loc**ation of Minimum
MPJMIN	**Min**imum
MPJPROD	**Prod**uct
MPJSUM	**Sum**

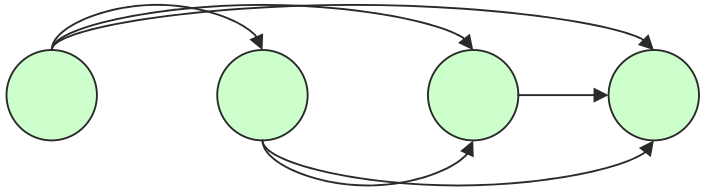
Table3: Operators in MPJ

* MAX/MIN LOC operations require a DataLoc array as the root's buffer.

2.4. Functions and Syntax

2.4.1. MPJ

MPJ contains main MPI operations. Traditional Init(string[]) initializes Java socket-based connections. Another Init(string[], IpTable, GridTcp) initializes connections with GridTCP. It also provides Rank(), Size(), Finalize(), etc.



2.4.1.1. **Init(String[] args):** for **Java Socket**

This Init() function establishes all to all connections using Java sockets. First, MPJ receives initialization commands such as rank, amslave, master node, number of processes, etc. Such arguments are mainly used to identify each process.

On the master process, Init() creates a ServerSocket and accepts connections from slaves until the number of connections equals to the number of processes. Following each connection, the master process reads the connecting process's rank and identifies each connection with that rank, storing the rank-connection information in JavaComm. Then, the master broadcasts the ranks and their corresponding hostname to all slave processes. At this point, the master process's Init() is complete.

For the slave processes, they first connect to the master process, send their rank, and then receive a table with other slaves' ranks and their hostnames. Once such information is exchanged with the master, the slaves will connect with each other. First, all of the slaves except for the highest ranking slave will create ServerSockets. Afterwards, the lowest ranking slave will accept connections from higher ranking slaves. The lowest ranking slave will then receive a rank from the connection it received and update its rank connection table (much like the master). When the lowest ranking slave has received all connections, its Init() is complete. The second lowest ranking slave then accepts connections from higher ranking slaves, and the process repeats until the second highest ranking slave has accepted a connection from the highest ranking slave, indicating all slaves are connected to all other slaves. At this point, the Init() process is complete for Java sockets.

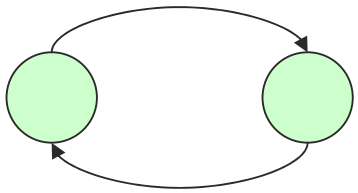
2.4.1.2. **Init(String[] args, IpTableEntry[] ipTable, GridTcp):** for **GridTcp**

This Init() function establishes all to all connections for GridTcp sockets. The algorithm for this case is very similar to the Init() for Java sockets (Section 2.4.1.1). However, the difference is that since the UserProgWrapper of GridTcp pre-initializes an ipTable corresponding to each of the hosts, all processes have already known each other. Thus, the process simply utilizes the connection algorithm described in Section 2.3.1.1 without distinguishing between the master and slaves (and also no broadcast of rank-connection information by the master). Each process simply connects to all lower ranking processes while accepting connections from higher ranking processes.

2.4.2. **Communicator [JavaComm & GridComm]**

Communicator provides all communications functions. A basic communication algorithm for MPJ is to receive and send messages between [or among] processes. Point-to-point communication is one of the key mechanisms in MPJ. Blocking communications such as Send(), IRecv() allow to . On the other hand, Isend() and Recv() are used for NonBlocking communications. Collective communications are able to carry out some operations over a group of processes. Gather(), Scatter(), Reduce() are collective communication related functions.

Communicator class is extended by two classes: JavaComm and GridComm. JavaComm is designed for communication using Java Sockets and SocketServers. GridComm is for GridTcp Sockets and requires GridTcp object and IpTable. Both JavaComm and GridComm allow socket communications using bytes and need InputStreamForRank[] and OutputStreamForRank[] to accomplish the connections. The interface between the two layers is clean and well-organized since same communication algorithms can be used for both JavaSocket and GridTcp Communications.



2.4.2.1. **Send(Object[] buf, int offset, int count, Datatype type, int dest, int tag)** – Blocking Communication

The Send() function takes in various parameters describing the datatype, the send count, the send buffer, the offset, the destination rank, and the message tag. The send buffer must be an array. The datatype is actually a Datatype object from MPJ, such as MPJ.SHORT.

Before sending a message, the Send() function first creates a header for the message, including the message's type, size in bytes, count, and the tag. Send() then serializes the message using the Datatype specified in the function parameters, and writes the header along with the serialized message to the output stream corresponding to the destination rank.

2.4.2.2. **Recv(Object[] buf, int offset, int count, Datatype type, int src, int tag)** – Blocking Communication

When a user calls the Recv() function, Recv() will perform a blocking read operation on the input stream corresponding to the source rank. First, Recv() reads 16 bytes of the message header and then reads the rest of the body with respect to the size defined in the header. Then, Recv() will deserialize the message using the correct Datatype. The completed message is stored in an MPJMessage, which is then checked against the parameters of Recv(). If the tag or datatype does not match, the message is stored in a message queue, and Recv() will read again for a new message.

If MPJ.ANY_SOURCE is specified, Recv() will poll each socket and read from the first socket with available data, and then check the tag.

If MPJ.ANY_TAG is specified, then Recv() will return the message if the datatype matches the parameter.

Recv() will crash if the count parameter is smaller than the actual message's count.

2.4.2.3. **Pack(Object[] inbuf, int offset, int incount, Datatype type, byte[] outbuf, int position)**

Pack() is similar to Send(). It uses the Datatype to serialize the message into the provided output buffer, and then returns the updated position. If the Datatype is an MPJ.OBJECT, each object is serialized individually (rather than as a buffer), so they can be deserialized or extracted individually.

2.4.2.4. **Unpack(byte[] inbuf, int position, java.lang.Object outbuf, int offset, int outcount, Datatype datatype)**

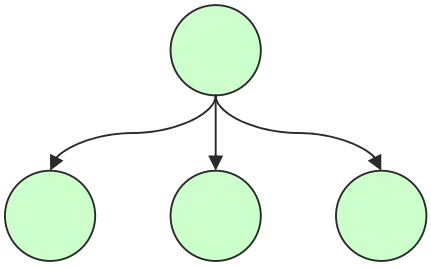
Unpack() performs the opposite of the Pack() operation. It will deserialize the input buffer into the output buffer using the corresponding Datatype's deserialize operation.

2.4.2.5. **Barrier()**

Barrier() is simple. Rank 0 will receive a message from all other ranks and broadcast another message once it has received from all other ranks. Thus, each process is blocked until all processes have called Barrier().

2.4.2.6. **Bcast(Object[] buf, int offset, int count, Datatype type, int root)**

Bcast() will broadcast the message specified in buffer from rank root to all other processes. Bcast currently follows a tree-structured algorithm, which reduces the number of send stages to $\log_2(n)$. Such an algorithm should be much better performance when broadcasting large messages. Bcast() on the root process will send the buffer to all other processes in the communicator, while the non-root processes will receive.



2.4.2.7. **Reduce()**

Reduce() is a collective communication call, meaning all processes within the communicator are involved. Reduce() will perform an operation defined by the op parameter. Operations are done element-wise on every send buffer, and the result is stored in the root's buffer.

Example with MPJ.SUM as the operation:

	send[0]	send[1]	send[2]	send[3]
p0	1	2	3	4
p1	1	2	3	4
p2	1	2	3	4

recvbuf [0] = 3, [1] = 6, [2] = 9, [3] = 12

2.4.3. Other Communication Functions

Other communications algorithms are compatible with the standard mpiJava. All of these functions require an array as the input and output buffers (even though the function prototype may just require an Object, this is type-casted into an array of appropriate type later). Their documentation can be found at http://www.hpjava.org/mpiJava.html. Please see online examples or the provided example code on how to use the following functions:

2.4.3.1. Isend(Object buf, int offset, int count, Datatype type, int dest, int tag)

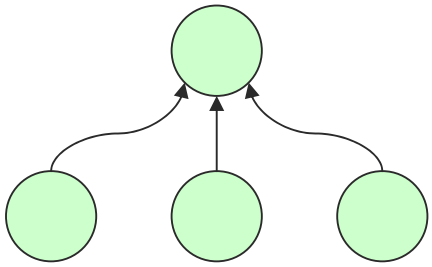
Spawns a thread to send a message, like send()

2.4.3.2. Irecv(Object buf, int offset, int count, Datatype type, int src, int tag)

Spawns a thread to recieve a message, like recv()

2.4.3.3. Gather(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)

Gathers all data in the inbuffers at all the processes to the outbuf of the root in rank order



2.4.3.4. Gatherv(Object sendbuf, Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int[] recvcounts, int[] displs, Datatype recvtype, int root)

Variation of Gather(), where different displaced input elements are allowed

2.4.3.5. **AllGather(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype)**

Like Gather(), but result of outbuf is sent to all processes

2.4.3.6. **AllGatherv(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int[] recvcounts, int[] displs, Datatype recvtype)**

Like Gatherv(), but result of outbuf is sent to all processes

2.4.3.7. **Scatter(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)**

Opposite of Gather(). Scatter the buffer on the root to all processes in rank order

2.4.3.8. **Scatterv(Object sendbuf, int sendoffset, int[] sendcounts, int[] displs, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)**

Variation of Scatter() where differently displaced input elements are allowed

2.4.3.9. **AllToAll(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype)**

Sends each element of inbuf to the corresponding ranking process. Does on all processes, and stores in rank order

2.4.3.10. **AllToAllv(Object sendbuf, int sendoffset, int[] sendcount, int[] sdispls, Datatype sendtype, Object recvbuf, int recvoffset, int[] recvcount, int[] rdispls, Datatype recvtype)**

Variation of AllToAll() where differently displaced input elements are allowed

2.4.3.11. SetBufferSize(int numbytes)

MPJ uses permanent buffers to help serialization and deserialization performance. As such, Communicator has a SetBufferSize(int numbytes) function.

Example:
MPJ.COMM_WORLD.SetBufferSize(655365);

3. Initialization and Usage

MPJ files are currently not in any sort of package, so the users will need to either put their program files into a folder with MPJ files or use the –classpath to point to a folder containing MPJ files.

3.1. How to create MPJ programs

3.1.1. mpiJava-S: Java Socket

e.g.) *MyApplication_S.java*

```
public class MyApplication_S {

  public int rank;
  public int nProcesses;

  public static void main (String[] args){

    MPJ.Init(args);
    rank = MPJ.COMM_WORLD.Rank();
    nProcesses = MPJ.COMM_WORLD.Size();
    char message[] = new char[] { 'H', 'e', 'l', 'l', 'o', '!' };
    int i, j;

    if(rank == 0){   //master

       MPJ.COMM_WORLD.Send(message, 0, message.length, MPJ.CHAR, 1, 1);
       System.out.println("Master sending: ");

       for(i=0; i<message.length; i++)
          System.out.print(message[i]);

       ……            //more statements will be inserted

    }
    else{             //slaves

       for(j=1; j<nProcesses; j++){
          MPJ.COMM_WORLD.Recv(message, 0, message.length, MPJ.CHAR, 0, 1);
          System.out.println("Slave " + j + " Received: ");
          for(i=0; i<message.length; i++)
             System.out.println(message[i]);
       }

       ……            //more statements will be inserted
    }

    MPJ.Finalize();
  }
}
```

This is a simple program example of mpjJava-S and will be run on Java sockets. For mpiJava-S, the code must have a main() in the code as well as a regular Java program. The MPJ program first invokes its MPJ instance using Init() function. The master sends a message, which is an array, to all other slave nodes. Each slave receives the message. At last, it finishes the MPJ operation using Finalize().

3.1.2. **mpiJava-A: GridTcp**

e.g.) *MyApplication_A.java*

```
public class MyApplication_A {

  public GridIpEntry ipEntry[];              // used by the GridTcp socket library
  public int funcId;                         // used by the user program wrapper
  public GridTcp tcp;                        // the GridTcp error-recoverable socket
  public int nprocess;                       // #processors
  public int rank;                           // processor id ( or mpi rank)

  public int func_0( String args[] ) {       // constructor
    MPJ.Init( args, ipEntry, tcp );          // invoke mpiJava-A
    .....;                                   // more statements to be inserted
    return 1;                                // calls func_1( )
  }

  public int func_1( ) {                     // called from func_0
    if ( MPJ.COMM_WORLD.Rank()== 0 )
      MPJ.COMM_WORLD.Send( ... );
    else
      MPJ.COMM_WORLD.Recv( ... );
    .....;                                   // more statements to be inserted
    return 2;                                // calls func_2( )
  }

  public int func_2( ) {                     // called from func_2, the last function
    .....;                                   // more statements to be inserted
    MPJ.Finalize( );                         // stops mpiJava-A
    return -2;                               // application terminated
  }
}
```

mpiJava-A is run on GridTcp sockets using UserProgWrapper class. UserProgWrapper creates a byte-presented stream as an execution snapshot. To serialize a program counter and stack which Java does not support, mpiJava-A uses a collection of methods. Each method is named "**func_*n*" (*n* is an integer starting from 0) and returns the index of the next methods. The User Program Wrapper saves the index value and retrieves the last index from the corresponding snapshot if a process crushes.

Program statements are partitioned into those functions (func_0 to func_n). Init() is called to invoke mpiJava-A inside the first function, "***func_0***". The last function returns **"-2"** to terminate MPJ. After that, the program calls Finalize() and finishes the mpiJava-A operations.

3.2. How to use

3.2.1. Create links to MPJ.jar and GridTcp.jar

```
[me@medusa mydir] $ ln –s ~uwagent/MA/MPJ.new/MPJ.jar MPJ.jar
[me@medusa mydir] $ ln –s ~uwagent/MA/GridTcp/GridTcp.jar GridTcp.jar
```

Users may use "MPJ.jar:GridTcp.jar:." as a class path for the following execution [3.2.2. and 3.2.3], if they create links above in the same folder with a MPJ program file.

3.2.2. mpiJavaS - MPJ

3.2.2.1. Java Socket

< Master >
java [**–cp** classPath] programName masterHostName [port#] [progArgs] **–np** #ofProcessors

< Slave >
java [**–cp** classPath] programName masterHostName [port#] **-amslave** **-yourrank** rank **-yourname** yourHostName **-np** #ofProcessors

---

e.g.) Master: *machine name - medusa*

```
[me@medusa mydir] $ java –cp
~uwagent/MA/MPJ.new/MPJ.jar:~uwagent/MA/GridTcp/GridTcp.jar:.
myMPJProg medusa 12345 –np 2
```

e.g.) Slave: *machine name – mnode0*

```
[me@mnode0 mydir] $ java –cp
~uwagent/MA/MPJ.new/MPJ.jar:~uwagent/MA/GridTcp/GridTcp.jar:.
myMPJProg medusa 12345 –amslave –yourrank 1 –yourname mnode0 –np
2
```

3.2.2.2. Java Socket with **mpjrun**

mpjrun performs automatic initialization of MPJ programs. The drawback to using mpjrun is that it requires multiple threads per process to monitor stderr and stdout.

- *Create a machine file:*

[me@medusa mydir]$ **vi hostfile**

e.g.) hostfile:

```
mnode0
mnode1
mnode2
mnode3
mnode4
mnode5
```

A machine file may not include a host name for a master. mpjrun will be run from the master machine ("medusa" in this example).

- *Run a program using mpjrun:*

java [**–classpath** pathForMPJrun] **mpjrun** [programName] [**-port** port#] [**-np** #ofProcessors] [**-machinefile** machineFileName] [**-cp** classPath]

e.g)Run a MPJ program (on a master node only):

```
[me@medusa mydir]$ java –classpath ~uwagent/MA/MPJ.new/MPJ.jar:.
mpjrun myMPJProg -port 12345 -np 4 -machinefile hostfile –cp
/home/uwagent/MA/MPJ/MPJ.jar:/home/uwagent/MA/GridTcp/GridTcp.jar
```

This will automatically open new SSH windows and run the program using commands for Java sockets [Refer to 3.2.1.1]. Before executing, the system might ask users a password for authentication per processor. The drawback to using mpjrun is that it requires multiple threads per process to monitor stderr and stdout. This will result in significant performance hits.

3.2.3. **mpiJavaA - GridTcp**

3.2.3.1. GridTcp Socket with **UserProgWrapper**

< Master >
java [**–cp** classPath] **UserProgWrapper -** #ofProcessors masterRank masterHostName **-** slaveRank slaveHostName **-** programName masterHostName [port#] **-np** #ofProcessors

< Slave >
java [**–cp** classPath] **UserProgWrapper -** #ofProcessors slaveRank slaveHostName **-** masterRank masterHostName **-** programName masterHostName [port#] **-np** #ofProcessors –yourrank slaveRank –yourname slaveHostName

---

e.g.) Master: *machine name - medusa*

```
[me@medusa mydir] $ java -cp
~uwagent/MA/MPJ.new/MPJ.jar:~uwagent/MA/GridTcp/GridTcp.jar:.
UserProgWrapper - 2 0 medusa - 1 mnode0 - myGridProg
medusa 12345 -np 2
```

e.g.) Slave: *machine name – mnode0*

```
[me@mnode0 mydir] $ java -cp
~uwagent/MA/MPJ.new/MPJ.jar:~uwagent/MA/GridTcp/GridTcp.jar:.
UserProgWrapper - 2 1 mnode0 – 0 medusa - myGridProg
medusa 12345 -np 2 -yourrank 1 -yourname mnode0
```

4. Performance Evaluation

Though our AgentTeamWork project including MPJ is still on-going, we have observed GridTcp library and mpiJava-A API and compared it with original mpiJava. The performance has been evaluated with our own test programs as well as with the Java Grande Forum MPJ Benchmark Suite produced by the University of Edinburgh.

For Java Sockets, performance of MPJ seems very reasonable. The performance should be considered as the optimal mpiJava performance, and no data conversion is needed. When data conversion is required, MPJ performance with Java sockets drops to slower levels than the original mpiJava. Unfortunately this means that when dealing with data that needs to be serialized, mpiJava-A (GridTcp) runs even slower.

Java byte arrays/buffers are very expensive to create and discard and greatly reduce the performance if they are created every Send() and Recv() pair, especially buffers greater than 512kb. To achieve the best performance with Java sockets, permanent send and receive buffers are used for serialization.

Object serialization and de-serialization is done using Object IO streams. Any user objects should overload the default serialization methods if object serialization performance is critical as the default methods provided by java are very slow. However, object serialization in mpiJava-S (Java socket) is still 20-30% faster than mpiJava object serialization.

Even after some turning of primitive serializations, the performance cost to convert into buffer is still very expensive. This is due to the high number of instructions needed to serialize a piece of data. Look at the following Java code for example.

```
int x;                              //for just 1 integer
byte[] arr[4];                      //extra memory cost
arr[3] = (byte) (x);
arr[2] = (byte) (x >>> 8);          //shift, cast, copy
arr[1] = (byte) (x >>> 16);         //repeat
arr[0] = (byte) (x >>> 24);
```

Like an example above, to serialize a single integer, there needs to be extra storage and quite a few instructions including assignment, cast, and shift used multiple times. In C/C++, this operation would be simply a pointer to the int followed by the number of bytes of the int.

This creates a major problem with collective communications where multiple Send() and Recv() pairs may be needed. There is a trade off between creating byte buffers and repeatedly serializing/de-serializing into the user's buffers. They are extremely slow.

4.1. PingPong Performance

<< Java Sockets >>

Optimal:

loop = 25, msgSize = 512bytes, elapsedTime=8msec
transfer rate = 25.6Mbps
loop = 25, msgSize = 1024bytes, elapsedTime=8msec
transfer rate = 51.2Mbps
loop = 25, msgSize = 2048bytes, elapsedTime=9msec
transfer rate = 91.02222222222223Mbps
loop = 25, msgSize = 4096bytes, elapsedTime=9msec
transfer rate = 182.04444444444445Mbps
loop = 25, msgSize = 8192bytes, elapsedTime=11msec
transfer rate = 297.89090909090913Mbps
loop = 25, msgSize = 16384bytes, elapsedTime=10msec
transfer rate = 655.36Mbps
loop = 25, msgSize = 32768bytes, elapsedTime=18msec
transfer rate = 728.1777777777778Mbps
loop = 25, msgSize = 65536bytes, elapsedTime=38msec
transfer rate = 689.8526315789475Mbps
loop = 25, msgSize = 131072bytes, elapsedTime=58msec
transfer rate = 903.944827586207Mbps
loop = 25, msgSize = 262144bytes, elapsedTime=98msec
transfer rate = 1069.9755102040817Mbps
loop = 25, msgSize = 524288bytes, elapsedTime=174msec
transfer rate = 1205.2597701149425Mbps
loop = 25, msgSize = 1048576bytes, elapsedTime=309msec
transfer rate = 1357.3799352750812Mbps

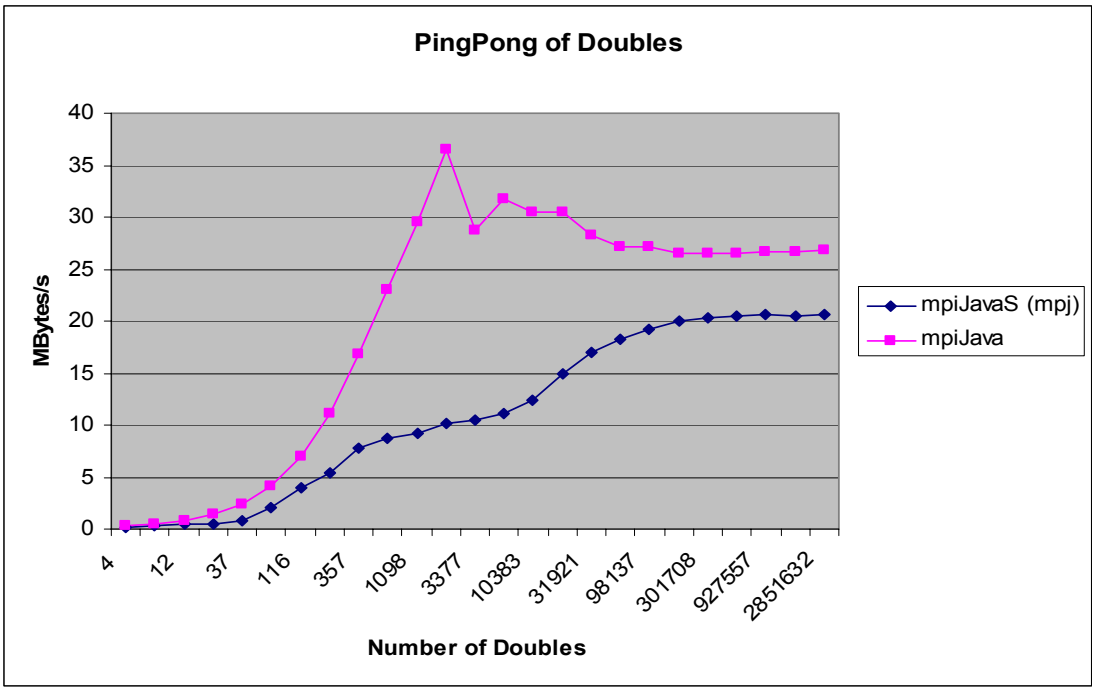
MPJ:

loop = 25, msgSize = 512bytes, elapsedTime=26msec
transfer rate = 7.876923076923076 5Mbps
loop = 25, msgSize = 1024bytes, elapsedTime=9msec
transfer rate = 45.51111111111111Mbps
loop = 25, msgSize = 2048bytes, elapsedTime=9msec
transfer rate = 91.02222222222223Mbps
loop = 25, msgSize = 4096bytes, elapsedTime=9msec
transfer rate = 182.04444444444445Mbps
loop = 25, msgSize = 8192bytes, elapsedTime=10msec
transfer rate = 327.68Mbps
loop = 25, msgSize = 16384bytes, elapsedTime=15msec
transfer rate = 436.9066666666667Mbps
loop = 25, msgSize = 32768bytes, elapsedTime=20msec
transfer rate = 655.36Mbps
loop = 25, msgSize = 65536bytes, elapsedTime=36msec
transfer rate = 728.1777777777778Mbps
loop = 25, msgSize = 131072bytes, elapsedTime=56msec
transfer rate = 936.2285714285714Mbps
loop = 25, msgSize = 262144bytes, elapsedTime=109msec
transfer rate = 961.9963302752293Mbps
loop = 25, msgSize = 524288bytes, elapsedTime=188msec
transfer rate = 1115.5063829787234Mbps
loop = 25, msgSize = 1048576bytes, elapsedTime=337msec
transfer rate = 1244.60059347181Mbps

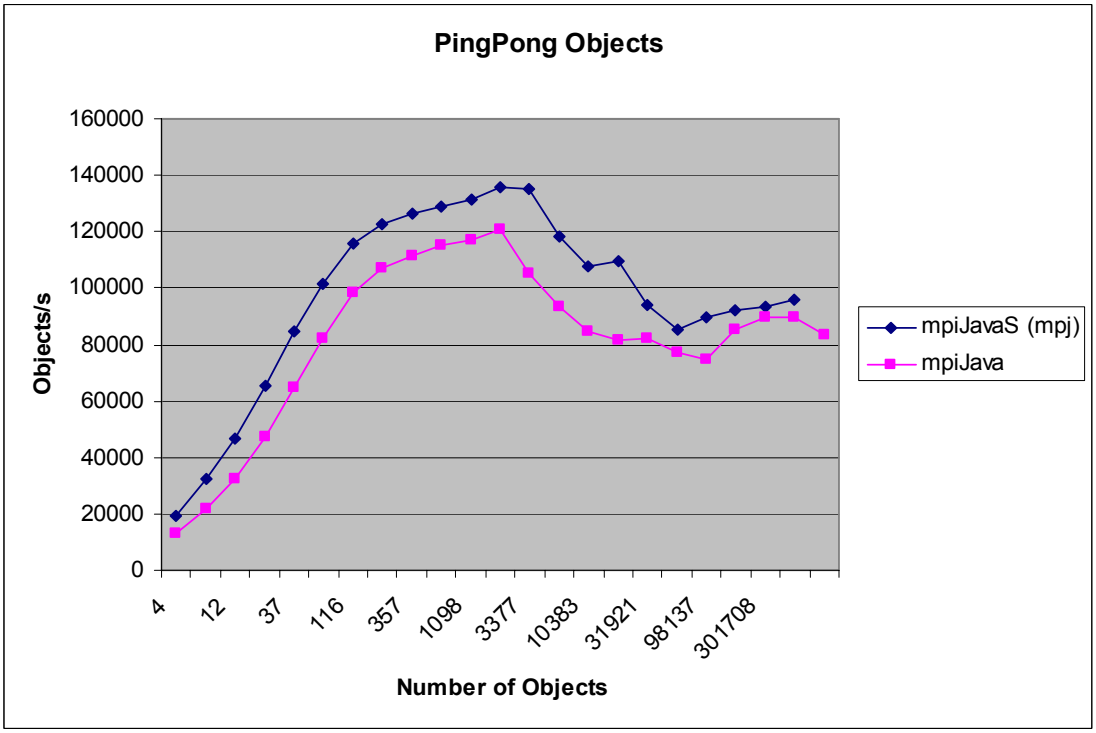
<< GridTcp >>
Optimal:
loop = 25, msgSize = 512bytes, elapsedTime=1789msec
transfer rate = 0.1144773616545556Mbps
loop = 25, msgSize = 1024bytes, elapsedTime=1749msec
transfer rate = 0.23419096626643798Mbps
loop = 25, msgSize = 2048bytes, elapsedTime=1789msec
transfer rate = 0.4579094466182224Mbps
loop = 25, msgSize = 4096bytes, elapsedTime=1743msec
transfer rate = 0.9399885255306942Mbps
loop = 25, msgSize = 8192bytes, elapsedTime=1758msec
transfer rate = 1.8639362912400457Mbps
loop = 25, msgSize = 16384bytes, elapsedTime=22msec
transfer rate = 297.89090909090913Mbps
loop = 25, msgSize = 32768bytes, elapsedTime=22msec
transfer rate = 595.7818181818183Mbps
loop = 25, msgSize = 65536bytes, elapsedTime=41msec
transfer rate = 639.3756097560976Mbps
loop = 25, msgSize = 131072bytes, elapsedTime=61msec
transfer rate = 859.4885245901639Mbps
loop = 25, msgSize = 262144bytes, elapsedTime=128msec
transfer rate = 819.2Mbps
loop = 25, msgSize = 524288bytes, elapsedTime=864msec
transfer rate = 242.7259259259259Mbps
loop = 25, msgSize = 1048576bytes, elapsedTime=1045msec
transfer rate = 401.3688038277512Mbps

MPJ:
loop = 25, msgSize = 512bytes, elapsedTime=1815msec
transfer rate = 0.1128374655647383Mbps
loop = 25, msgSize = 1024bytes, elapsedTime=1755msec
transfer rate = 0.2333903133903134Mbps
loop = 25, msgSize = 2048bytes, elapsedTime=1759msec
transfer rate = 0.4657191586128482 4Mbps
loop = 25, msgSize = 4096bytes, elapsedTime=1798msec
transfer rate = 0.9112347052280311Mbps
loop = 25, msgSize = 8192bytes, elapsedTime=1747msec
transfer rate = 1.875672581568403Mbps
loop = 25, msgSize = 16384bytes, elapsedTime=31msec
transfer rate = 211.40645161290325Mbps
loop = 25, msgSize = 32768bytes, elapsedTime=71msec
transfer rate = 184.60845070422536Mbps
loop = 25, msgSize = 65536bytes, elapsedTime=76msec
transfer rate = 344.92631578947373Mbps
loop = 25, msgSize = 131072bytes, elapsedTime=148msec
transfer rate = 354.2486486486486Mbps
loop = 25, msgSize = 262144bytes, elapsedTime=285msec
transfer rate = 367.9214035087719Mbps
loop = 25, msgSize = 524288bytes, elapsedTime=458msec
transfer rate = 457.8934497816594Mbps
loop = 25, msgSize = 1048576bytes, elapsedTime=793msec
transfer rate = 528.9160151324086Mbps

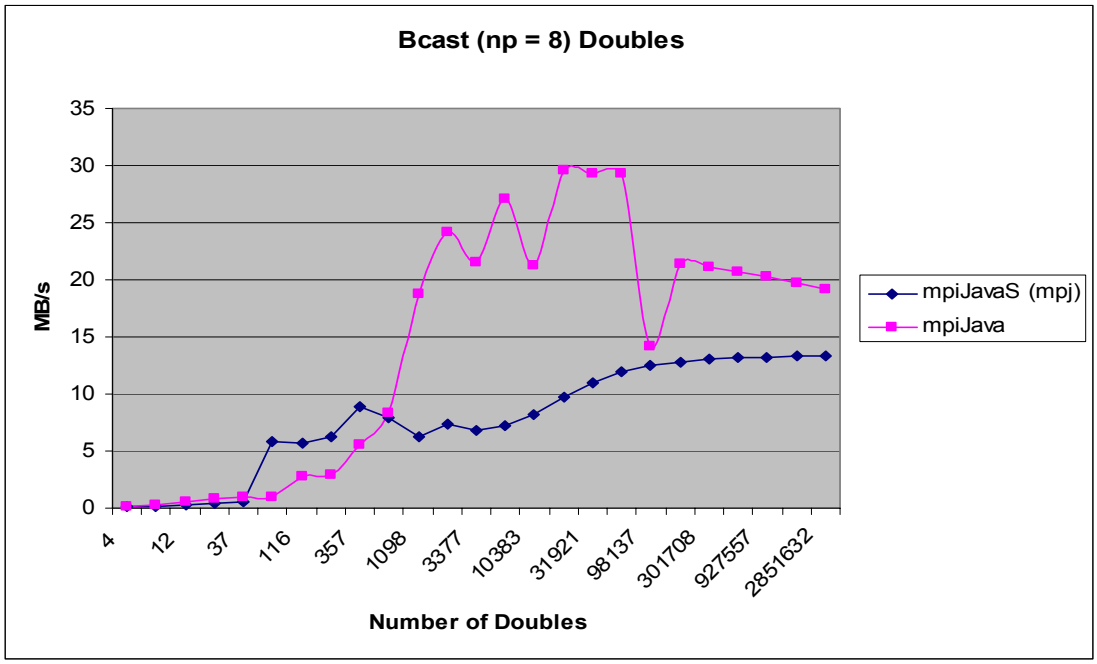
PingPong (send and recv) - Doubles



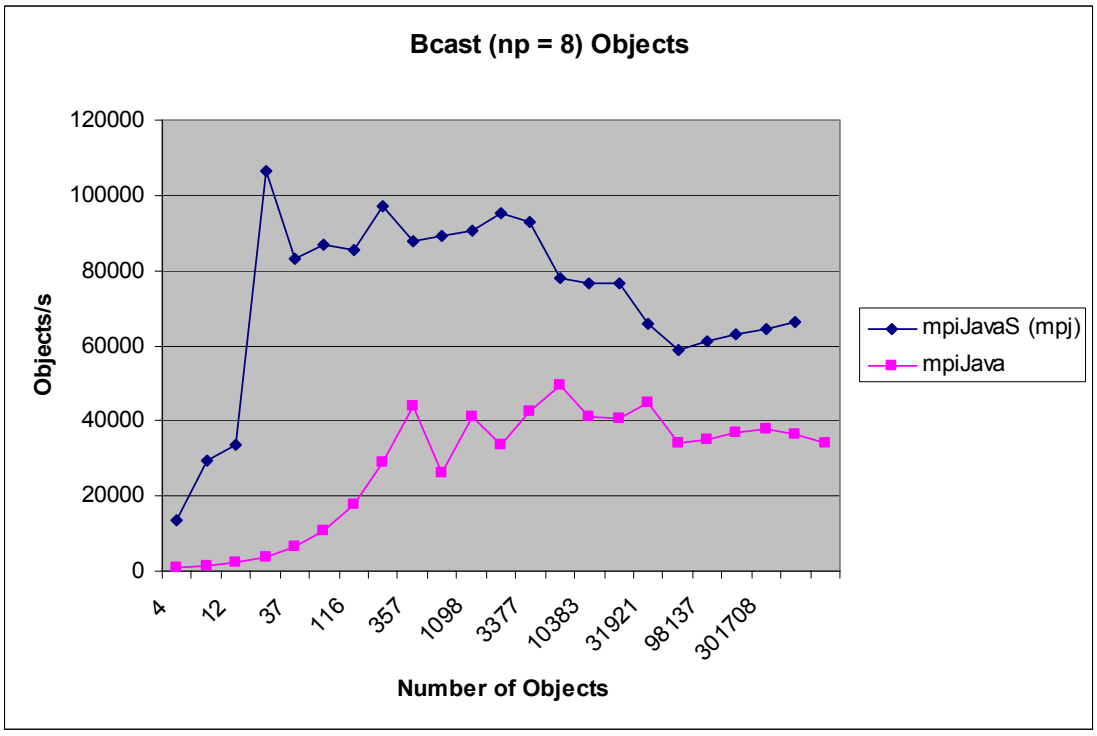
Ping Pong – Objects



Bcast – 8 Processes: Doubles



Bcast – 8 Processes: Objectes



5. Summary and Known Bugs

Next step is to develop a tool to automatically parse a user program into GridTcp functions for the best performance such as automating user job distribution, management, and error recovery using UserProgWrapper class.

Table4 shows the result of running several different test programs using mpjrun. Unfortunately, MPJ has memory issues and causes out of memory errors when collective communications, such as Gather(), has been used. Some of the test programs fail with the certain array size when it is run on a certain number of processors.

	2	4	8	16	32
PingPong	○	-	-	-	-
JGFPingPong	× (529010<)	-	-	-	-
JGFReduceBench	○	○	○	○	○
JGFAlltoallBench	○	× (288539<)	× (288539<)	× (154991<)	× (83255<)
JGFGatherBench	× (1626361<)	× (927577<)	× (529010<)	× (301708<)	× (172072<)
JGFBarrierBench	○	○	○	○	○

○ -> Passed more than 5 times
△ -> Failed a few times out of 5 times or more
× -> None succeeded(Array Size: passed)

Table4: test result of MPJ-S using mpjrun

In addition, mpiJava-A results in "out of memory" as well. Also, *GridPP* class stops and hangs there during the middle of the execution. Further observation will be needed to evaluate the performance of mpiJava-A using UserProgWrapper.

Future improvements include fixing the problems above and implementing an MPJException class to handle various exceptions. We are also planning to implement additional communications algorithms and expand the parallel operations of MPJ.