Message Passing Java
(MPJ)

Intermediate Report
And
Performance Results for Java and GridTcp

Prepared By
Zhiji Huang
(3/17/2005)

1.0 Introduction

Message Passing Java (MPJ) is being developed 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. This report describes the design, implementation, and performance results throughout the midway point of this project.

2.0 Design

MPJ design is as follows:

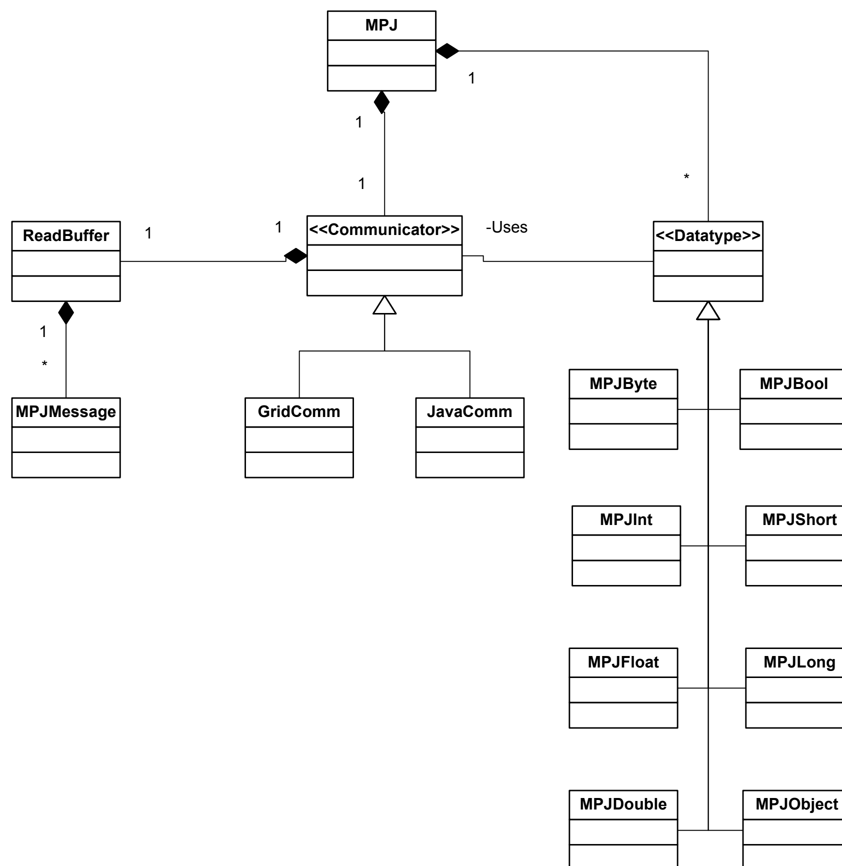

Diagram 1: Major MPJ classes currently in project.

The major classes shown in Diagram 1 (above) are the implemented clsses of MPJ throughout the first half of this project's development.

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 Send(), Recv(), Barrier(), Bcast(), Pack() and Unpack(). In the future, this class will contain all communications functionalities supported by MPJ.

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.

3.0 Algorithms

This section describes the important algorithms used in the major functions of MPJ.

3.1 MPJ.Init(String [] args) for Java

This Init() function establishes all to all connections using Java sockets. First, MPJ receives initialization commands such as rank, am slave, 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 equal the number if processe. 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.

3.2 MPJ.Init(String [] args, IpTableEntry[] ipTable, 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 3.1). However, the difference is that since the UserProgWrapper of GridTcp pre-initializes an ipTable that corresponds to each of the hosts, all processes already know about each other. Thus, the process simply utilizes the connection algorithm described in Section 3.1 without distinguishing between the master and slaves (and also no broadcast of rank-connection information by master). Each process simply connects to all lower ranking processes while accepting connections from higher ranking processes.

3.3 Communicator.Send(Object[] buf, int offset, int count, Datatype type, int dest, int tag)

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.

Note that this a blocking operations.

3.4 Commnicator.Recv(Object[] buf, int offset, int count, Datatype type, int src, int tag)

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.

3.4 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.

3.5 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.

3.5 Barrier()

Barrier() is simple in that 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().

3.6 Bcast(Object[] buf, int offset, int count, Datatype type, int root)

Bcast() will broadcast the message specified in buf 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.

4.0 PingPong Performance

Java Sockets

Optimal:

```
loop = 25, msgSize = 512bytes, elaspedTime=8msec
transfer rate = 25.6Mbps
loop = 25, msgSize = 1024bytes, elaspedTime=8msec
transfer rate = 51.2Mbps
loop = 25, msgSize = 2048bytes, elaspedTime=9msec
transfer rate = 91.0222222222223Mbps
loop = 25, msgSize = 4096bytes, elaspedTime=9msec
transfer rate = 182.04444444444445Mbps
loop = 25, msgSize = 8192bytes, elaspedTime=11msec
transfer rate = 297.89090909090913Mbps
loop = 25, msgSize = 16384bytes, elaspedTime=10msec
transfer rate = 655.36Mbps
loop = 25, msgSize = 32768bytes, elaspedTime=18msec
transfer rate = 728.1777777777778Mbps
loop = 25, msgSize = 65536bytes, elaspedTime=38msec
transfer rate = 689.8526315789475Mbps
loop = 25, msgSize = 131072bytes, elaspedTime=58msec
transfer rate = 903.944827586207Mbps
loop = 25, msgSize = 262144bytes, elaspedTime=98msec
transfer rate = 1069.9755102040817Mbps
loop = 25, msgSize = 524288bytes, elaspedTime=174msec
transfer rate = 1205.2597701149425Mbps
loop = 25, msgSize = 1048576bytes, elaspedTime=309msec
transfer rate = 1357.3799352750812Mbps
```

MPJ:

```
loop = 25, msgSize = 512bytes, elaspedTime=26msec
transfer rate = 7.876923076923076 5Mbps
loop = 25, msgSize = 1024bytes, elaspedTime=9msec
transfer rate = 45.51111111111111Mbps
loop = 25, msgSize = 2048bytes, elaspedTime=9msec
transfer rate = 91.0222222222223Mbps
loop = 25, msgSize = 4096bytes, elaspedTime=9msec
transfer rate = 182.04444444444445Mbps
loop = 25, msgSize = 8192bytes, elaspedTime=10msec
transfer rate = 327.68Mbps
loop = 25, msgSize = 16384bytes, elaspedTime=15msec
transfer rate = 436.9066666666667Mbps
loop = 25, msgSize = 32768bytes, elaspedTime=20msec
transfer rate = 655.36Mbps
loop = 25, msgSize = 65536bytes, elaspedTime=36msec
transfer rate = 728.1777777777778Mbps
loop = 25, msgSize = 131072bytes, elaspedTime=56msec
transfer rate = 936.2285714285714Mbps
loop = 25, msgSize = 262144bytes, elaspedTime=109msec
transfer rate = 961.9963302752293Mbps
loop = 25, msgSize = 524288bytes, elaspedTime=188msec
transfer rate = 1115.5063829787234Mbps
loop = 25, msgSize = 1048576bytes, elaspedTime=337msec
transfer rate = 1244.60059347181Mbps
```

GridTcp

Optimal:

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

MPJ:

```
loop = 25, msgSize = 512bytes, elaspedTime=1815msec
transfer rate = 0.1128374655647383Mbps
loop = 25, msgSize = 1024bytes, elaspedTime=1755msec
transfer rate = 0.2333903133903134Mbps
loop = 25, msgSize = 2048bytes, elaspedTime=1759msec
transfer rate = 0.46571915861284824Mbps
loop = 25, msgSize = 4096bytes, elaspedTime=1798msec
transfer rate = 0.9112347052280311Mbps
loop = 25, msgSize = 8192bytes, elaspedTime=1747msec
transfer rate = 1.875672581568403Mbps
loop = 25, msgSize = 16384bytes, elaspedTime=31msec
transfer rate = 211.40645161290325Mbps
loop = 25, msgSize = 32768bytes, elaspedTime=71msec
transfer rate = 184.60845070422536Mbps
loop = 25, msgSize = 65536bytes, elaspedTime=76msec
transfer rate = 344.92631578947373Mbps
loop = 25, msgSize = 131072bytes, elaspedTime=148msec
transfer rate = 354.2486486486486Mbps
loop = 25, msgSize = 262144bytes, elaspedTime=285msec
transfer rate = 367.9214035087719Mbps
loop = 25, msgSize = 524288bytes, elaspedTime=458msec
transfer rate = 457.8934497816594Mbps
loop = 25, msgSize = 1048576bytes, elaspedTime=793msec
transfer rate = 528.9160151324086Mbps
```

For Java sockets, PingPong performance is about 1-5% slower than optimal performance.

For GridTcp sockets, PingPong performance is significantly slower, at around 30-50% less than optimal, although the performance curve does not match.

5.0 Performance Analysis

At this point, the performance for Java sockets seem very reasonable. However, it should be noted that such performance is the optimal MPJ performance, in that no data conversion is needed.

It should also be noted that to achieve such performance, permanent send and receive buffers are used. It has been found that such buffers are very expensive to create and discard, and will cause a performance degradation if they are created and discarded for every Send() and Recv() pair, especially for buffers greater than 512kb. How such permanent buffers will affect GridTcp's checkpointing performance should be examined more closely.

GridTcp performance is much lower than optimal. However, the cause of such poor performance is likely lies somewhere in the interface between UserProgWrapper and MPJ.

5.1 Object Serialization

At this time, such algorithms are still being tuned. The idea is to manually serialize primitives by using bitwise shift. For floating point numbers, Java's ByteBuffer will be used (or alternatively the primitive wrapper classes' bits conversion function, although there seems to be some problems associated with such a conversion).

Object serialization and deserialization is done using Object IO streams. It should be noted that 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.

6.0 Continuation of MPJ

For the second half of the project, GridTcp performance will be debugged and hopefully resolved. Following further performance tuning, some of the more advanced MPJ functions will be implemented, including Gather, Scatter, Reduce, AllGatther, AllReduce, AlltoAll etc.