CSS 497 Spring'11 Benchmark Performance Testing: MPI, Sockets, MASS

Tim Clapp

Table of Contents

Introduction

This document covers all the work I completed during the course of the spring quarter in 2011. The task I undertook was to port the Java Grande Benchmark Suite of test programs based on MPI, into java TCP/ip Sockets, and again to the MASS library.

Covered in this document is the implementation details for both sockets and mass, where to find the source files, an explanation of how to run the programs, and any shortcomings or known caveats. The tests that I completed in both sockets and MASS include:

1. Ping Pong
2. Barrier
3. Broadcast
4. Reduce
5. Gather
6. Scatter
7. All Reduce
8. Sparse Matrix Multiplication

Location of Source Files

Sockets: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets
MASS – Multi-Process: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process
MASS – Multi-Thread: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Thread

Running Sockets: General

The typical sockets program will require the user to start a client on one computing node, and a server on another. The same class file is used to represent the client and server, and it is up to the arguments supplied by the user to determine which side is which.

For example, the Ping Pong program invocation is:

Client side: java –Xmx1g PingPongSockets uw1-320-17 50332
Server side: java –Xmx1g PingPongSockets 50332

In later socket programs, the typical form of the command line arguments is:

Java –Xmx1g Program <port> <server host names…>

The server side invocation should not be supplied with host names but only the port the client will try to connect on.

There is no argument check to ensure numbers are integers or strings are strings so be careful when typing arguments.

Running MASS: General

On Hercules, two versions of the MASS implementation are present: Multi-Process and Multi-Threaded. Both versions can be invoked as per usual MASS programs, but the Multi-Process version, as of now, must include MASS.jar in the Hercules home directory. Also, each program has an accompanying compile script that will compile and create the necessary .jar files. The jar file to be created, called SocketsMassBenchmark.jar, will also be copied to the Hercules home directory.

Within each folder, is a script file called run. It is easiest to modify this script in order to run the program.

Finally, in each program folder, the following items must exist per Mass in order to properly compile and run:

1. Jsch-0.1.42.jar
2. Mass.jar
3. Machine files (mfile.txt)

There is no argument check to ensure numbers are integers or strings are strings so be careful when typing arguments.

compile script

The compile script will first remove any and all class files and any temporary files *~. Next it will compile *every* .java file in the directory. This compile statement will reference Mass.jar, so that is why it must be located in the directory.

Next, the script will determine if the SocketsMassBenchmark.jar exists in the current working directory. If it does, it will simply append all of the class files it just created. If it does not exist, it will be created, and the new class files will be added.

After that, the script checks the Hercules home directory for SocketsMassBenchmark.jar. If it exists, the class files are appended. If it does not exist, the local version in the current working directory is copied over. And finally, the script will print out the current contents of the SocketsMassBenchmark.jar for verification purposes.

run script

run.sh is a simple script that relieves the user from having to type the lengthy command arguments required by MASS.

The typical run.sh script will print out the current test name followed by the command to run the program. Once complete, a 'Complete.' statement is printed to the screen.

Most implementations require one or two arguments to be supplied. To know for sure which arguments provide, the easiest and fastest way to check is looking at this run script.

Basic Program Structure for Section 1

```
For [ loopsize ] {
        generate [array] to send
        size = 1
        while [ size < MAXSIZE ] {
                start timer
                For [ k < size ]
                        perform benchmark test
                lapse timer
                size *= 2
        }
        measure performance:  (length of array * 8 * size)  / time;    /*  typical  */
}
```

loopsize = number of times to test: each successive test increases the size of the data array or object array
MAXSIZE = 1 million
size = increasing powers of 2: 1,2,4,8, … 524,288
Array length increases exponentially per loopsize: 4,7,12,21,37,… 2,851,632

A Note on JGB Performance Measuring

In most instances, The Java Grande Benchmark Suite measures performance in terms of bytes/second. A timed measurement will include multiple round trips of data, and/or whatever the aim of the particular test is.

Ping Pong

The Ping Pong performance test is a measure of the speed at which computing nodes can simply send and receive data from one another. This test is measured in bytes/second. Ping Pong can only be executed between two nodes in either implementation.

This is a simple, single-threaded implementation in sockets. In this test, an array of doubles and an array of objects are sent back and forth between two computing nodes. The object is a simple object that contains a single double value. The conceptual design is seen below in figure 1.

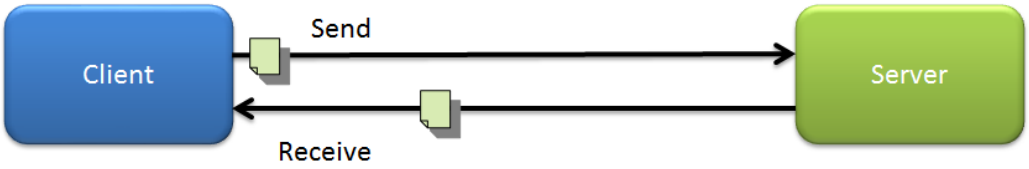


Figure 1

Sockets

Absolute path to source files: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section1/PingPong
Command line Client Side: java PingPongSockets <port> <server name>

Command line Server Side: java PingPongSockets <port>

Example:

java -Xmx1g PingPongSockets 50332	server
java -Xmx1g PingPongSockets 50332 uw1-320-22	client

To run this test, start the program at two different nodes with the arguments supplied as above. In most cases, the heap size will need to be increased: -Xmx1g.

MASS

Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section1/PingPong

In this test, it is not necessary to supply any further arguments other than what MASS requires. As can be seen in the run script:

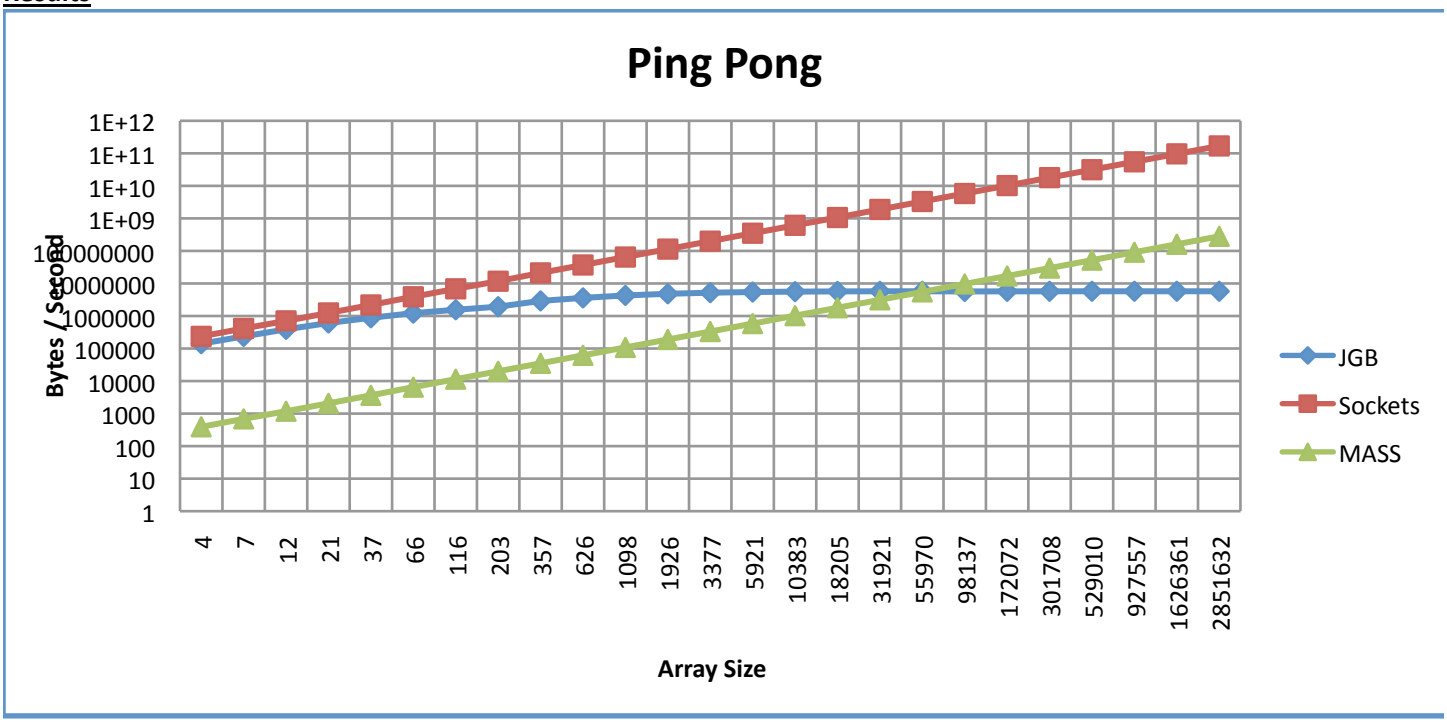
java -Xmx1g -cp './*' PingPongMass dslab ds1ab-302 mfile.txt './*'

In order to choose the computing nodes, simply update the machine file: mfile.txt

In the MASS implementation of Ping Pong, exchangeAll is used to represent the sending and receiving of data. Places is initialized with a simple two element array and exchangeAll is called with an offset [][] = 1. This ensures that only one element will call the other.

Before the exchangeAll call is made, the value to ping pong is assigned to the outObject of each element. This call is heavy since one of the elements will not actually use it. Instead, via exchangeAll, it will return the value that was sent to it.

Results



Barrier

The Barrier performance test is a measure of how quickly threads and/or processes can be synchronized. This test is measured in barriers / second. The conceptual design is seen below in figure 2.

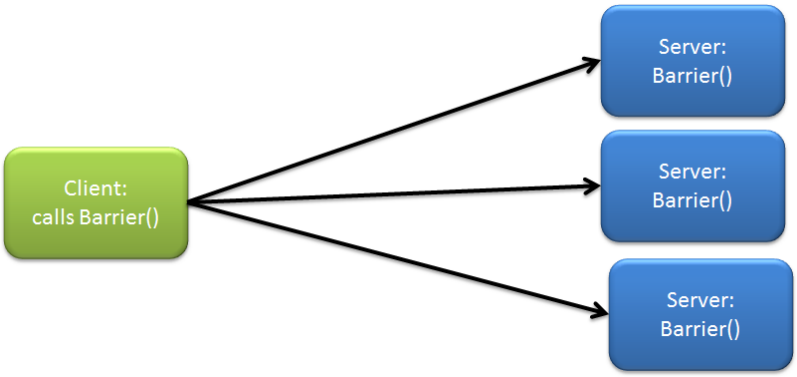


Figure 2

Sockets

Absolute path to source files: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section1/Barrier
Command line Client Side: java BarrierSockets <port> <server name…>
Command line Server Side: java BarrierSockets <port>

Example:
java -Xmx1g BarrierSockets 50332 server
java -Xmx1g BarrierSockets 50332 uw1-320-22 client

The Barrier test is scalable. The number of servers is up to the user. Note that the command line arguments have changed since Ping Pong. Instead of specifying the host name first, the port is always specified first.

With sockets, to simulate the barrier synchronization, a single byte is sent to each of the servers in a round robin fashion.

MASS

Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section1/Barrier

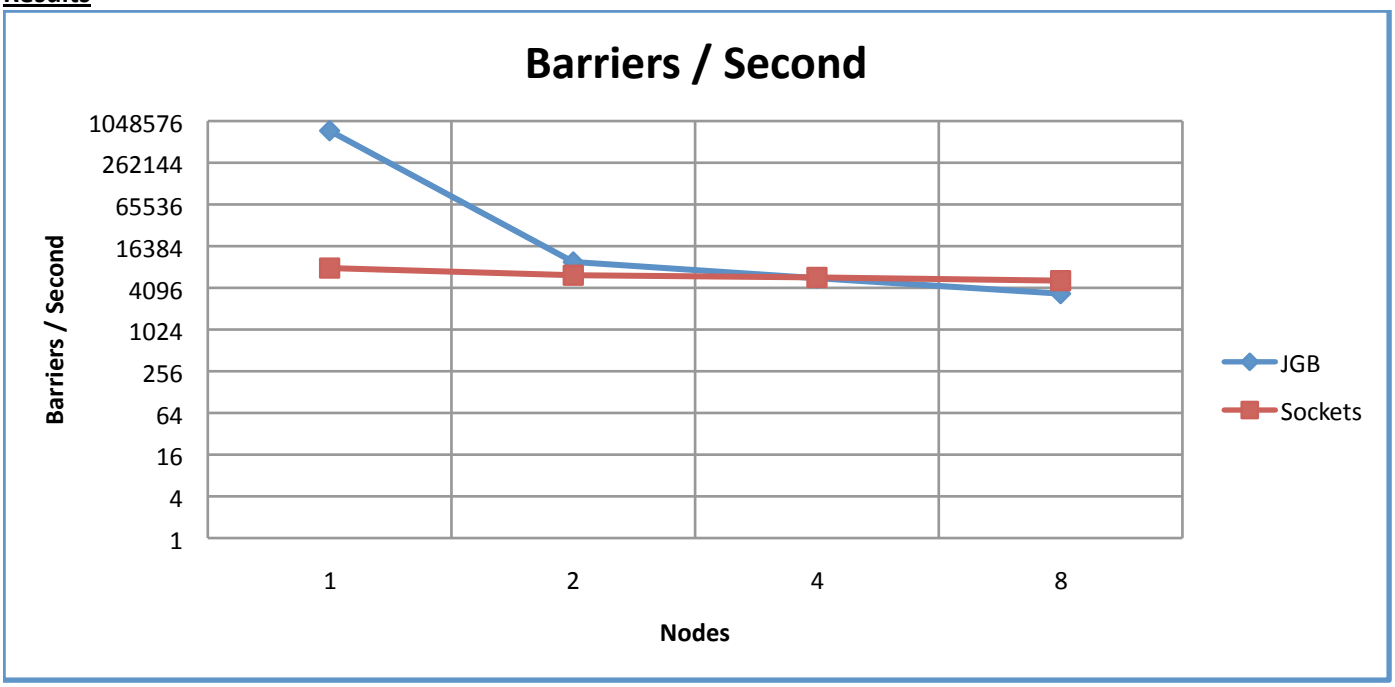
The barrier implementation in MASS is also scalable. The number of computing nodes to synchronize also corresponds to the number of Place elements to create. This number is specified in the command line arguments:

java -Xmx1g -cp './*' BarrierMass dslab ds1ab-302 mfile.txt './*' 4

The machine file must also correspond to the number of elements specified.

In order to accomplish and simulate the barrier in MASS, the callAll method is used. This call simply sends to each computing element a null value, and returns null.

Results



The results of the Barrier test for MASS are unexpected. Most times the time required to perform the barrier is 'infinite' or rather so small it cannot be represented, and thus it does not appear in the results. The cause of this may be related to the implementation. Since each element is called with a null object and returns a null object, the operation is incredible fast since no data is actually transmitted.

Broadcast

The Broadcast test measures in bytes / second the sending of an array of doubles, and also an array of objects. This test is similar to the Ping Pong test except the data transferred is one direction only: client to server, or one computing node to another.

The conceptual design is seen right in figure 3.

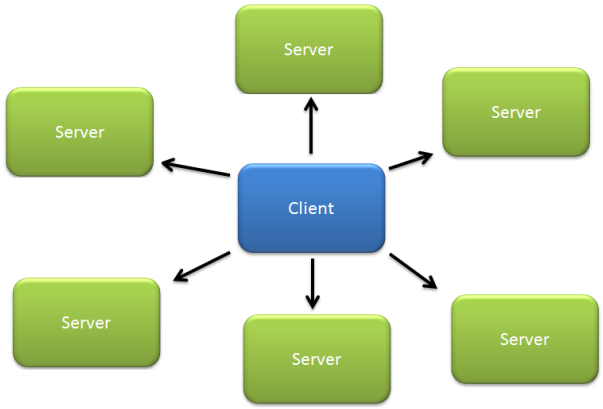


Figure 3

Sockets

Absolute path to source files: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section1/Broadcast
Command line Client Side: java BroadcastSockets <port> <server name…>
Command line Server Side: java BroadcastSockets <port>

Example:
java -Xmx1g BroadcastSockets 50332 server
java -Xmx1g BroadcastSockets 50332 uw1-320-22 client

The number of servers to broadcast to is up to the user. For benchmarking purposes, 1,2,4, and 8 servers were used. The data is broadcast out to the servers in a round robin fashion.

MASS

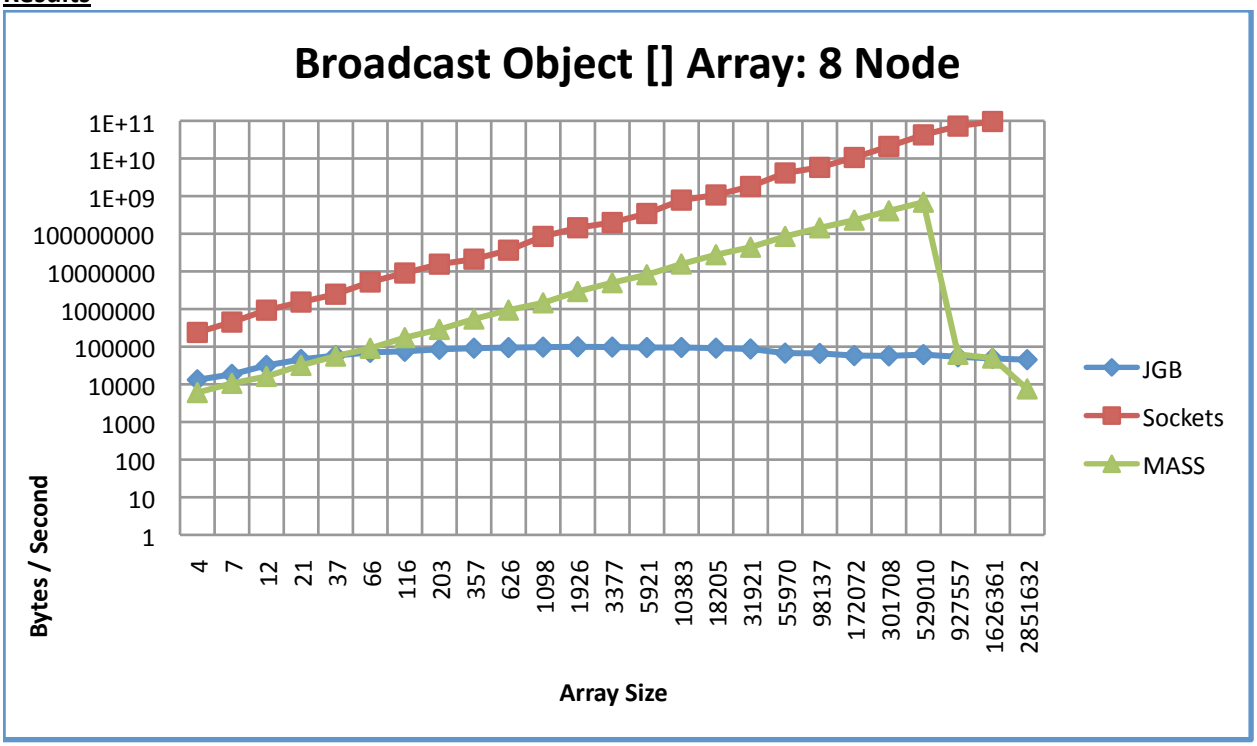
Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section1/Broadcast

Like Barrier, the number of nodes to broadcast data to must be specified in the command line arguments. This number will also correspond to the number of Place elements created.

java -Xmx1g -cp './*' BroadcastMass dslab ds1ab-302 mfile.txt './*' 4

In order to perform the broadcast, the callAll method is used. The data to send is passed in as the argument, and the return value from the called method is always null.

Results



Reduce

The conceptual design is seen right in figure 4.

In this implementation, for both sockets and mass, the server portion continually sends data to the client which then performs a summation of the received data.

The Reduce test will only send an array of doubles. The object test was not included (JGB skipped it as well).

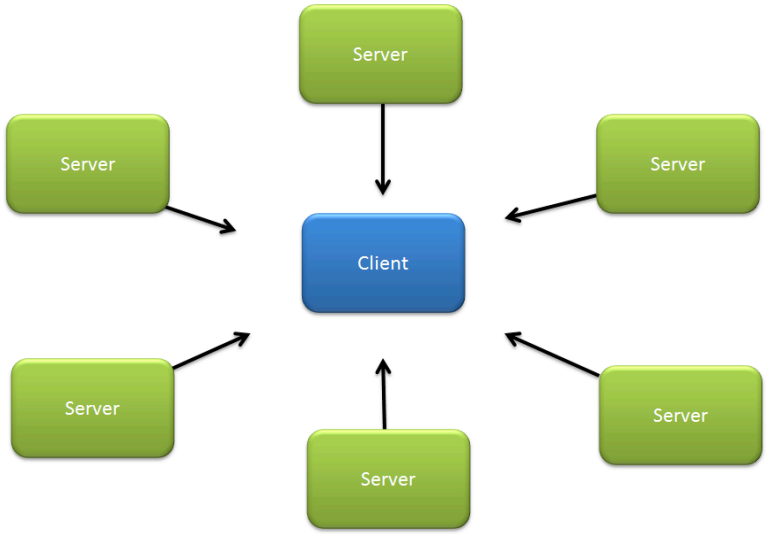


Figure 4

Sockets

Absolute path to source files:
/net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section1/Reduce
Command line Client Side: java ReduceSockets <port> <server name…>
Command line Server Side: java ReduceSockets <port>

Example:

java -Xmx1g ReduceSockets 50332 server
java -Xmx1g ReduceSockets 50332 uw1-320-22 client

In the sockets implementation, the client does not send the server any data !

MASS

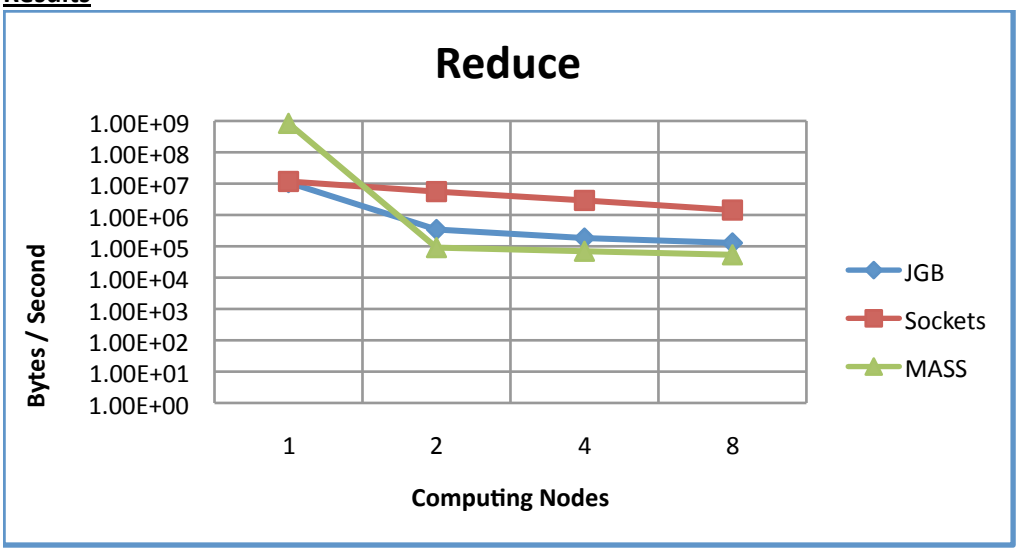
Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section1/Reduce

The 'client' in the MASS version will send data to the other computing nodes. The data that is sent is the double[] array via callAll. Later, once the actual test begins, the 'client' will again use callAll and this time it will receive back the data that it very first sent to the 'servers.' This functionality mimics the server operations in the sockets version. What is being done here, is first initializing the 'servers' with data.

The callAll method used will return an object []. This object array essentially contains the outObject results from each of the elements. This mimics the servers sending the client data. Each object within object[] array is then cast to a double[] and summed as per the Reduce operation.

Inside the directory is a runAll script that will run this test with 1,2,4,8 computing nodes. The number of nodes to use is specified on the command line arguments. Inside the directory are several machine files corresponding to the number of computing nodes specified on the command line args. See the script for further details.

Results



Gather

The Gather test works like the Reduce test with the exception that it does not sum the results from each server or computing node. The conceptual design is the same as seen in Figure 5.

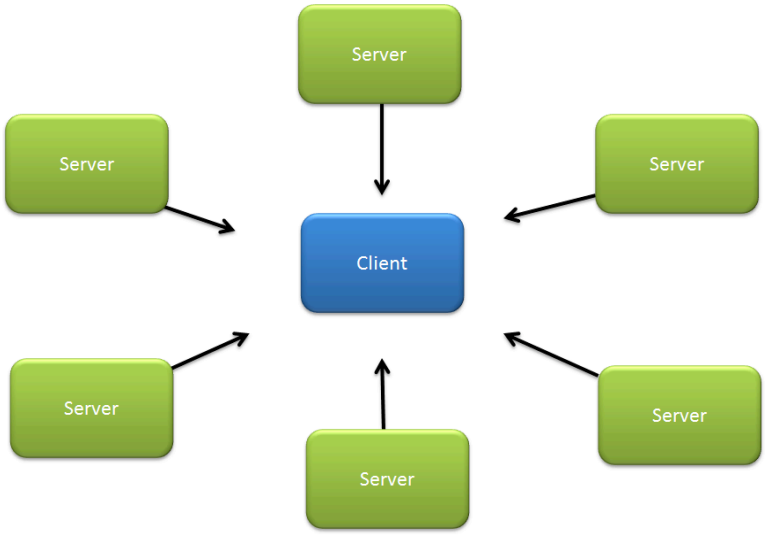


Figure 5

Sockets

Absolute path to source files: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section1/Gather
Command line Client Side: java GatherSockets <port> <server name…>
Command line Server Side: java GatherSockets <port>

Example:
java -Xmx1g GatherSockets 50332 server
java -Xmx1g GatherSockets 50332 uw1-320-22 client

Two different coding attempts were made. The first attempt was a single threaded version that worked exactly like the Reduce test with the exception that instead of summing the results, the data received from each server was copied into a unique place in a receiving double [] buffer using System.arraycopy(…).

The copy operation proved to be exceptionally detrimental to performance. Once the receiving array size became greater than 100k elements long, the test became painfully slow, and it was apparent that this was not a good approach.

The second attempt was to program the test using multiple threads, where each thread managed a socket. The main thread manages a double[][] where the first dimension corresponds to the number of threads, ie sockets, and the second dimension corresponds to data read from the server.

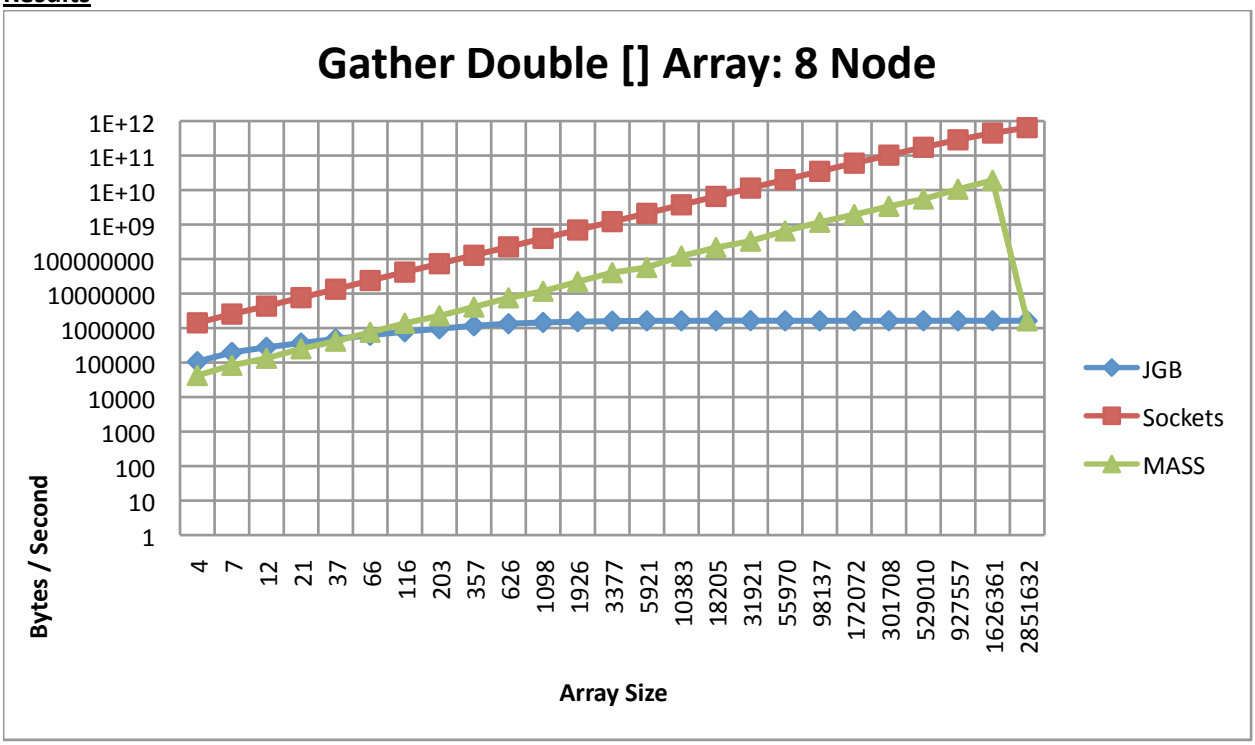
MASS

Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section1/Gather

For running, see the runAll or run script. The format is exactly the same as Reduce.

The concept works exactly the same as described in sockets, and the actual implementation works similar to the sending and receiving operation as described in Reduce.

Results



Scatter

Scatter uses a loopsize = 2. This means that the array length will only increase once. A total of two tests will be executed and the resulting array sizes are equal to: 4 & 4472

The conceptual design is the same as seen in Figure 6.

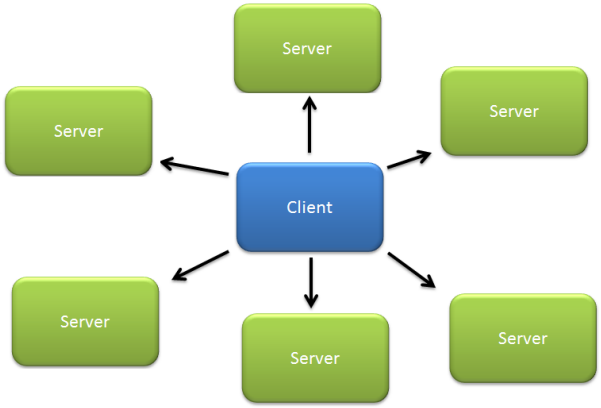


Figure 6

Sockets

Absolute path to source files: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section1/Scatter
Command line Client Side: java ScatterSockets <port> <server name…>
Command line Server Side: java ScatterSockets <port>

Example:
java -Xmx1g ScatterSockets 50332 server
java -Xmx1g ScatterSockets 50332 uw1-320-22 client

The sockets version of this test is similar to the original first attempt with Reduce. The reason it suffices for Scatter is because the loopsize is very small.

Unlike Reduce, the Scatter operation does copy a portion of unique data from one array and sends it to a single computing node. Thus, each computing node will receive unique data. The way this is accomplished is by using the System.arraycopy(…) method to copy a portion of the array into another array and sending that secondary array to a server.

As mentioned previously, System.arraycopy(), although an expensive operation, proves in this case to provide the necessary means of an copy method without hampering the performance of the test to any great degree.

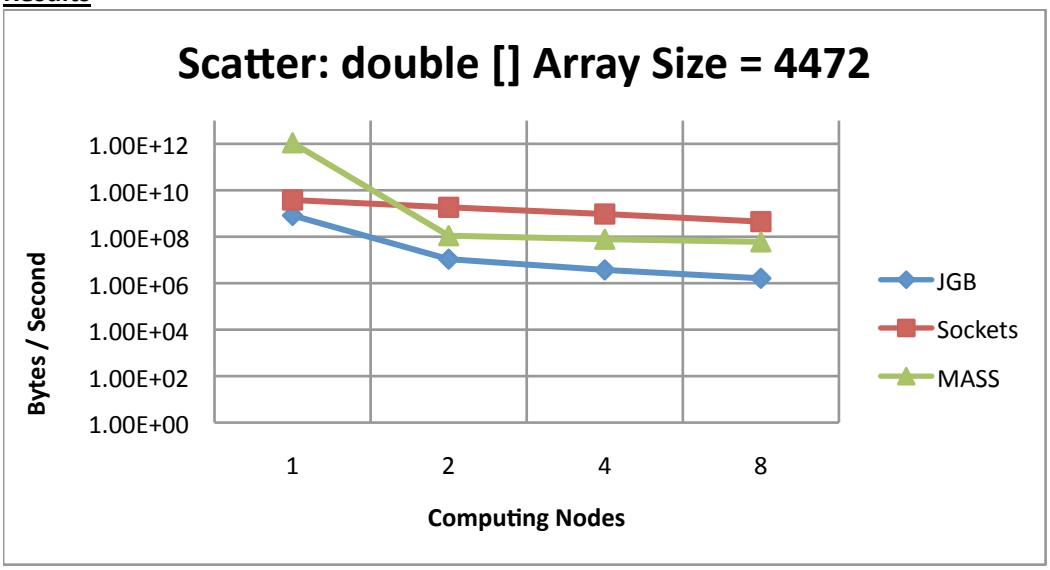
MASS

Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section1/Scatter

The MASS implementation uses the same algorithm designed in sockets. An array is partitioned, and each portion of the array is copied into a temporary buffer via System.arraycopy(…) and then sent to each computing node in a round robin fashion.

The scripts run and runAll have the exact same format as documented in earlier tests.

Results



All Reduce

AllReduce also uses a smaller loopsize: 20. The All Reduce test is a combination of the Reduce and Broadcast tests. Per iteration, data is read from each computing node to a singular, master node and the results are summed. Following that operation, the final summed results are then broadcast out in a round robin fashion to the rest of the computing nodes.

This test is not implemented in JGB. There was confusion between AllReduce and AlltoAll. JGB uses AlltoAll which is different than All Reduce. Thus, in the results comparison, JGB is not represented.

Sockets

Absolute path to source files: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section1/AllReduce
Command line Client Side: java AllReduceSockets <port> <server name…>
Command line Server Side: java AllReduceSockets <port>

Example:

```
java -Xmx1g AllReduceSockets 50332                  server
java -Xmx1g AllReduceSockets 50332 uw1-320-22       client
```

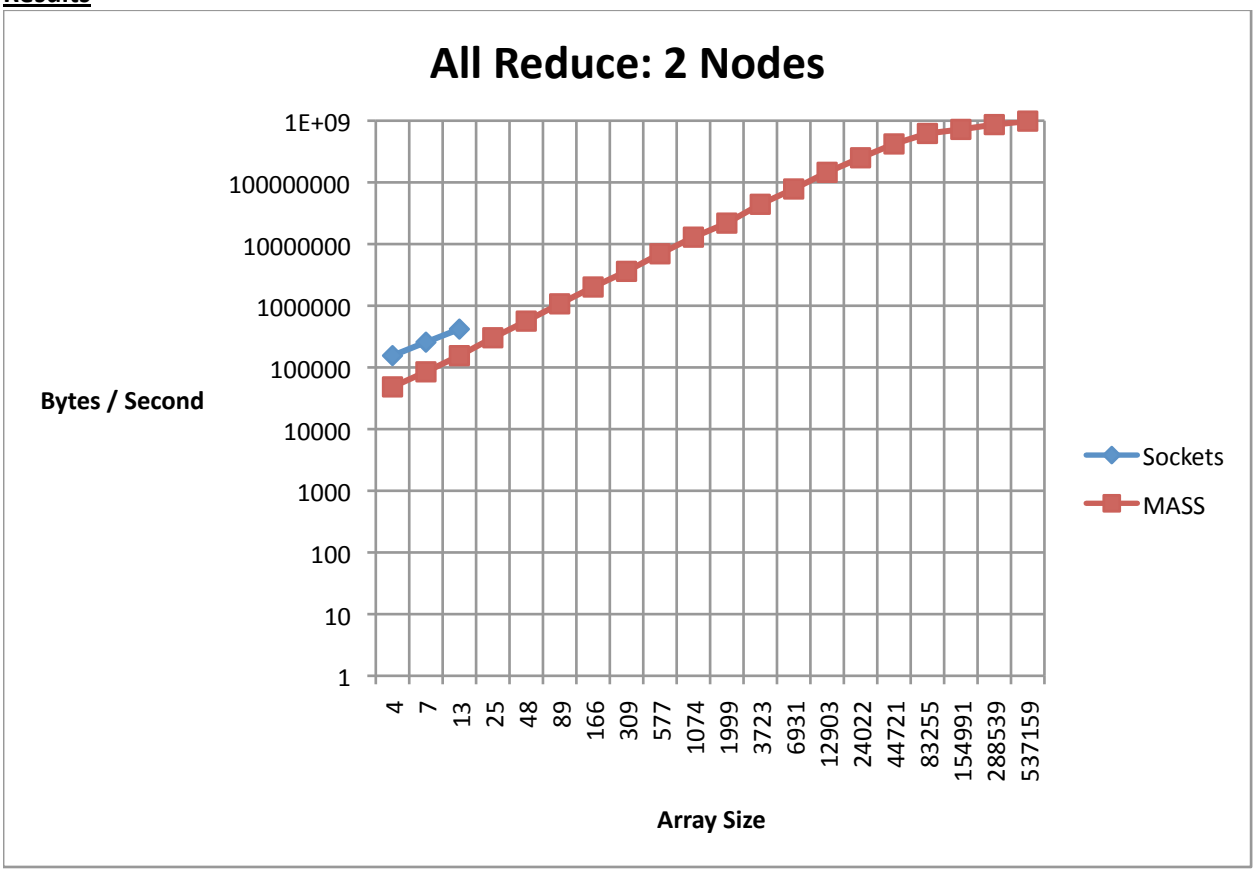
The Reduce and Broadcast operations are heavy. This test in sockets will not complete all 20 iterations.

MASS

Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section1/Scatter

Implementation is no different than previous tests – uses same sort of logic to send and receive data: ie reduce and broadcast. The algorithm of Reduce and Broadcast work the same as sockets. This test will finish all 20 iterations.

Results



Sparse Matrix Multiplication

The Sparse Matrix Multiplication test is from section 2 of the Java Grande Benchmark suite. This matrix multiplication test is a simulation of multiplication and a third matrix is not actually computed. In order to verify the accuracy of the computation, the final array storing the data is summed together and the final result is compared to a hard coded value per JGB.

There are 3 different size kernels that can be used in test:
Size A: 250,000
Size B: 500,000
Size C: 2,500,000

The size to compute is specified by the user in the command line arguments. One item of note is that the JGB test has each rank continually send each computation to every other rank. This is to facilitate the summing at the end of the program by rank 0. The sending operation is not performed in Sockets and MASS. This may account for the large performance delta.

Sockets

Absolute path to source files: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Sockets/Section2/MatrixMult/
Command line Client Side: java MatrixSockets <size> <port> <server host names....>
Command line Server Side: java MatrixSockets <size> <port>

Example:

```
java -Xmx1g MatrixSockets 0  50332                    server
java -Xmx1g MatrixSockets 0  50332 uw1-320-22         client
```

The sockets implementation is multi-threaded. Each thread manages a socket connection to the server node. The main thread will compute the sparse matrix, and determine which slice of the matrix each thread receives. Once a thread receives its portion, it sends it to the server which performs the computation. Once the computation is complete at each server, the result is sent back.

In order to validate the computation, the tolerance of the sum of the returned results is compared with a value that is hard coded per the JGB suite.

MASS

Absolute path to source: /net/metis/home3/dslab/SensorGrid/Applications/JavaGrande/Mass/Multi-Process/Section2/MatrixMult

The program begins with a call to a method: start(). Start() is the main program method. It first determines which kernel size to use, the number of computing nodes to use, and then initializes the sparse matrix.

Next the Places is created followed by code to create the slices of the matrix to send to each element in Places. This part of the code is quite heavy because three large pieces of data, unique to element, must be created and sent. Once each slice is computed, the callAll method is used to pass to each element the correct slice of the matrix.

As with sockets, each element will perform the computation, and finally send the results back. In order to accomplish this operation, callAll is used, but the argument passed to each element is just a null object. When returned, that same object will hold the value of the computation performed at each element. Once summed together, the value is compared against the target value per JGB.

To run the test using 1,2,4,8 nodes with each kernel size, see the script runAll. An excerpt can be seen below. Notice the arguments at the end have changed. The first argument after './*' corresponds to the kernel size, and the next argument corresponds to the number of computing nodes to use.

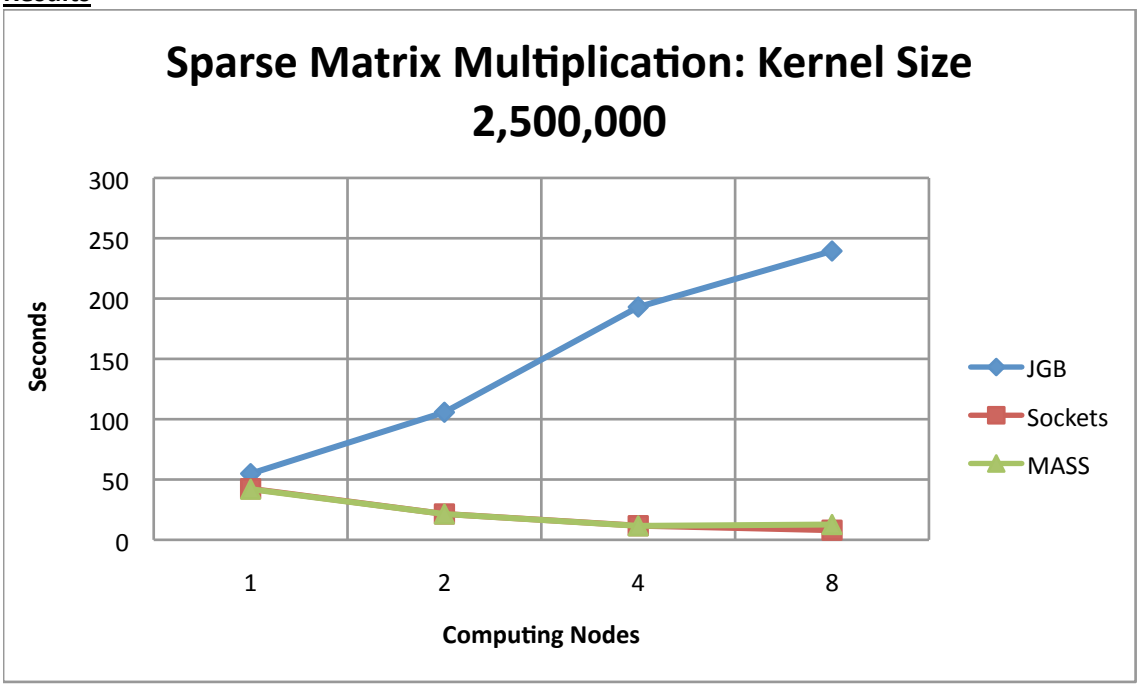
```
echo "Running Test Size 0 "
echo "Running MatrixMass on 1 nodes"
java -Xmx1g -cp './*' MatrixMass clappt wherewasit612@ mfile1.txt './*' 0 1
echo "Running MatrixMass on 2 nodes"

…….

echo "Running MatrixMass on 8 nodes"
java -Xmx1g -cp './*' MatrixMass clappt wherewasit612@ mfile4.txt './*' 2 8

echo "Complete."
```

Results



Summary Performance

The Summary Performance represents the outcome of the benchmark tests between the three different distributed computing models. As expected, sockets perform the fastest in terms of bytes / second, followed by MPI, and last MASS.

Comparison in Terms of Programmability

Usability / Paradigm	MPI	Sockets	MASS
Learning Curve	moderate	gentle	steep
Abstraction	moderate	none	high
Ease of Use	easy	moderate	difficult
Dev. Time	medium	slowest....	fastest!

In terms of Programmability, I focused on four key aspects:

1. Learning Curve
2. Level of Abstraction

3. Ease of Use
4. Development Time

Learning Curve

The learning curve for MASS compared to MPI and sockets is steep. The primary reason the curve is steep is that programming with MASS is very much different than MPI and sockets. With MPI and Sockets, it is easy to conceptualize and to focus on what each computing node will be responsible for computing. This control comes in the form of ranks for MPI and a client and server with Sockets. But with MASS, there is no concept of a central computing node, and instead, the programmer is forced to think in terms of how to make the grid of computing nodes work together in concert.

Level of Abstraction

For level of abstraction, I refer to how much of the programmer's responsibility is taken care of by libraries and thus abstracted away. For MASS, the level of abstraction is high. Using Places, the programmer simply passes in a reference to the class object and MASS takes care of the grid initialization. As part of this operation, the programmer is also free from having to set up sockets and input/output streams. So unlike sockets, which has no abstraction, MASS does a good job of abstracting the lower level communication details and frees the programmer of the responsibility of handling these operations.

Ease of Use

MASS takes a little getting used to. This is in part due to the steep learning curve and also due to the fact that for the most part, the programmer is restricted to two main methods: exchangeAll and callAll. Once these two methods are mastered, using MASS gets easier, but sometimes the use of only those two methods gives a feeling of being bounded by them.

Development Time

The redeeming quality of MASS in terms of programmability comes in the form of development time. When I was programming the tests, I would first develop the code using sockets – and this is simply because sockets took the most time to code. After I had the logic worked out with sockets, I simply lifted the test logic out, and was able to wrap a few methods around that logic with MASS and get the program up and running usually quite quickly.