Parallel Computing 2012
MPI Programming Assignments
Due Date: Dec 29th, 2012

On Lonestar, an implementation of MPI called MVAPICH is installed. In MVAPICH, there are two commands that will be used mainly: mpicc, the command (a script) to compile MPI programs, and mpiexec the command to execute a MPI program. Check with "modules" for the availability of a different implementation of MPI.

Task 1 Preliminaries
export lonestar=129.114.53.21
ssh *-o PreferredAuthentications=password* username@lonestar *Or* ssh username@lonestar

Exercise 1 "Hello World"
Task 1: Executing a simple Hello World program.
This program is given below:

```
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv ) {
char message[20];
int i,rank, size, type=99;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank == 0) {
strcpy(message, "Hello, world");
for (i=1; i<size; i++)
MPI_Send(message, 13, MPI_CHAR, i, type, MPI_COMM_WORLD);
}
else
MPI_Recv(message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
printf( "Message from process =%d : %.13s\n", rank,message);
MPI_Finalize();
}
```

The program sends the message "Hello, world" from the master process, (rank = 0) to all the slave processes (workers, rank != 0). The slave processes receive the messages and then all print the messages out to stdout.

Compilation:
Compile and execute the hello program using four processes in total. To compile the program use the script:

```
mpicc -o hello hello.c
```

which uses the gcc compiler (probably) to link in the libraries and create an executable hello, and hence all the usual flags that can be used with gcc can be used with mpicc.

Execution:
For the MPI installation on Lonestar, it is necessary to submit the application to **ibrun** with the specific path to the executable, i.e.:

```
ibrun -n 4 ./hello
```

Note : ibrun is a wrapper for mpirun/mpiexec that is exclusive to TACC resources

So far, four instances of the program will execute just on one node. You should get the output:

```
Message from process =0 : Hello, world
Message from process =2 : Hello, world
Message from process =1 : Hello, world
Message from process =3 : Hello, world
```

The order of messages may be different; it will depend upon how the four processes are scheduled to use IO.

Task 2 Modification so that master prints out all messages

Modify the hello.c program so that the slave processes do not issue the print statements, but each sends a message back to the master containing their rank after the master receives each message, it print out a message containing the slave's rank. The program should produce output such as:

```
Master: Hello slaves give me your messages
Message received from process 1 : Hello back
Message received from process 2 : Hello back
Message received from process 3 : Hello back
```

Now, all print statements are issued by the master.

Clue: MPI_Comm_rank returns the processes rank using the argument rank.

Task 3: Different messages from each process

Alter the code so that each process send a different message back, e.g.:

```
Master: Hello slaves give me your messages
Message received from process 1 : Hello, I am John
Message received from process 2 : Hello, I am Mary
Message received from process 3 : Hello, I am Susan
```

Task 4: Experiments with tags

In the previous programs, the tag of 99 is used in messages. Modify the program from task 3 so that the master process sends a messages to each slave, but with the tag of 100 and the slave waits for message with a tag of 100. Confirm program works.

Repeat but make the slaves wait for tag 101, and check program hangs. Why?

Exercise 2 Using multiple computers

In the previous exercise, only one computer was used, with multiple processes timeshared on the single processor. In the following exercise, multiple compute nodes will be used. Also the time of execution will be measured.

2A: Identifying the machine names

Modify the program hello.c to also output the machine name, and generate output such as follows:

```
Message from c341-303.ls4.tacc.utexas.edu process =0 : Hello, world
Message from c342-303.ls4.tacc.utexas.edu process =3 : Hello, world
Message from c343-303.ls4.tacc.utexas.edu process =1 : Hello, world
Message from c344-303.ls4.tacc.utexas.edu process =2 : Hello, world
.....
```

A C statement to obtain the hostname is **gethostname(&myname,80)** where **myname** is declared as

char myname[80];

Two ways to run this job: interactive mode using idev or batch mode using qsub.

idev -A TG-ASC120007 -pe 12way24 -q normal -N hello -m e -l h_rt=01:00:00

Example job scripts are available online in /share/doc/sge on TACC Ranger system.

(http://www.tacc.utexas.edu/user-services/user-guides/lonestar-user-guide#running)

2B: Broadcast algorithms

In this hands-on lab, you will use MPI point-to-point operations to implement the two broadcast algorithms. One of these uses a minimum spanning tree approach and is latency-optimal; the second uses a scatter + all-gather approach and is bandwidth-optimal (to within a constant factor). The algorithm sketches are available in the slides by Prof. Sathish Vadiyar (http://www.serc.iisc.ernet.in/~vss/courses/PPP/CollectiveCommunicationImplementations.ppt). For this part, Please work in teams of two.
Download and unpack this week's source tarball:

Use the MPI.zip file for the sample codes available in Piazza.

This tarball contains the following:

- serial.c: An implementation of a "naive" broadcast algorithm.
- tree.c: A latency-optimal implementation of a minimum spanning tree-based algorithm.
- bigvec.c: A partial implementation of the scatter + all-gather algorithm, which you will complete and compare against the tree-based algorithm.
- A bunch of other files, described below as needed.

Creating a communication model for point-to-point operations

We wish to devise, model, and implement algorithms for *broadcasting* the data from one MPI process to all others, using only the MPI point-to-point communication operations, such as MPI_Send, MPI_Recv, and/or their non-blocking counterparts, MPI_Isend, MPI_Irecv, and MPI_Wait.
The file serial.c, in the tarball, implements a "naive" broadcast algorithm. The data resides initially on MPI rank 0; this rank sends its data to each of the other processes, one by one. Inspect serial.c and verify that the bcast() routine implemented therein implements such a scheme.
We ran this benchmark on the Lonestar. The results appear in the graph below. More specifically, we show the average time per broadcast (y-axis) as a function of the size of the data size (x-axis).

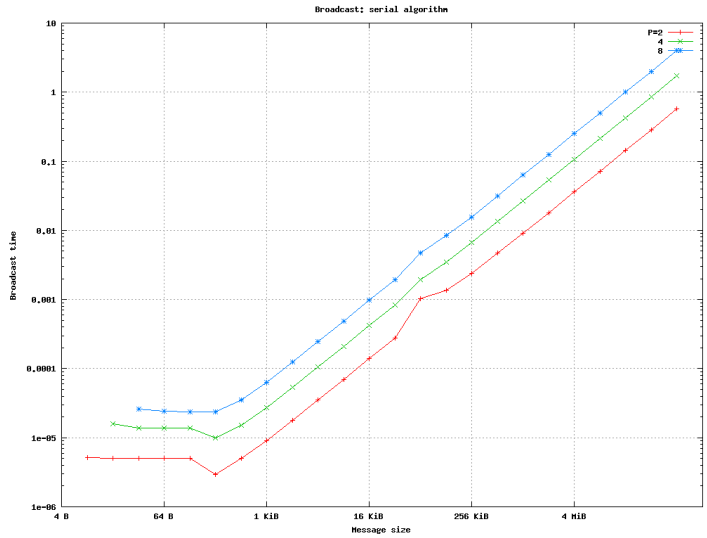

Figure 1: Performance of the naive algorithm on Lonestar. Note that the x-axis is on a $\log_2$ scale, the y-axis on a $\log_{10}$ scale.

Question 1: Assuming that the time $T(n)$ to send a message of size n is $T_{\text{msg}}(n) = \alpha + \frac{n}{\beta}$, use these data to estimate α and β. You may either eyeball the plot or use the raw data used to generate these plots, which appear at the following links for two processors, four processors, and eight processors. Explain how you derived your estimate and report α in microseconds and β in megabytes (10^6 bytes) per second. (Note that the plot shows message sizes along the x-axis in bytes, kibibytes, and mebibytes.)

The raw data is stored in a tab-delimited files. Each file has four columns. The first column is the number of MPI processes (ranks). The second column is the size of the data being broadcast, in Bytes. The third column is the average time to perform the broadcast, in seconds. The fourth column is the number of timing trials used to compute the average. (You can basically ignore the last column.)

Modeling the "small" and "large" vector broadcast algorithms

Recall from the last class that we described two algorithms for broadcast. The first algorithm is a *minimum spanning tree* approach. We argued this method is good for*small* messages because it is optimal with respect to latency (number of messages). The other uses a two-stage "scatter" plus "all-gather" technique, which we argued was good for large messages ("big vectors") because it trades a higher latency component for a nearly-optimal bandwidth component.

Look at the bcast() code in tree.c. Convince yourself that it implements a minimum spanning tree algorithm like the one described in class. Note that it assumes a power-of-two number of processes and, furthermore, that the number of processes divides the number of data elements.
Question 2: Using your model of message time from Question 1, write down a model for the tree-based algorithm.

Running the tree-based algorithm

Let's see how accurately the model from Question 2 predicts actual execution time by running the tree-based algorithm. We have provided a Makefile to simplify compiling and running this code (and the others we will use in this lab).
To compile, simply execute:

```
make
```

If it succeeds, it will produce the binary, tree.
We have provided Makefile rules that make it easy to run this code and generate a performance plot like that shown in Figure 1 (above). The command,

```
make pbs ALG=tree P=2 ; qstat -a
```

will submit a batch job that runs the tree program with two nodes (P=2) and various message sizes. Go ahead and run this command now; the "qstat -a" part peeks at the job queue to verify the job is in the queue. (Repeat "qstat -a to monitor the progress of this job, as in previous labs.) Once it begins running, it should complete in about a minute and generate a file called, tree-2.dat, which will have the same format as the serial-?.datfiles above. Repeat this command with P=4 and P=8 to collect data for the tree-based algorithm running with four and eight nodes, respectively.
Once you've collected this data, you can generate a plot by executing the command,

```
make plot-alg ALG=tree
```

This will run a *gnuplot* script to plot the tree-2.dat, tree-4.dat, and tree-8.dat data. If successful, it will create a file called tree.png. You can download or use display as in theprevious lab to see it.
Question 3: Compare these data to your model from Question 2. How well do they agree?

Implementing the "scatter + all-gather" approach for "big vectors"

Take a look at bigvec.c, which is a partial implementation of the algorithm designed for large vectors. In particular, we've provided the "scatter" step; you will need to complete the "all-gather" step.
Question 4: Complete the "all-gather" code, using only point-to-point MPI operations (either blocking or non-blocking, as you wish). See the notes below on compiling and testing your code. You may make the same assumptions as we do in the tree-based algorithm: power-of-two number of processes and the number of processes evenly dividing the message length. Once you've gotten it working, be sure to upload your final implementation onto Github.

To compile your implementation, you can use the command,

```
make bigvec
```

This will generate the executable, bigvec. You may choose to test it interactively on a single node; once you've gotten it debugged, perform a timing run on eight nodes using the command,

```
make pbs ALG=bigvec P=8
```

When this run succeeds, it will create a data file called, bigvec-8.dat.
Question 5: Plot the serial-8.dat, tree-8.dat, and bigvec-8.dat together. (See below for a make rule we have provided that uses *gnuplot* script to do it; alternatively, use any plotting software you wish.) You should see that the tree-based method is faster for "small" messages, whereas the scatter+allgather method if faster for "large" messages. What is the cross-over point (message size) between the tree and scatter+allgather methods?
We have provided a *gnuplot* script that can generate this plot. Use make to run it by issuing the following command:

```
make plot-all P=8
```

Assignment Submission

Submit at github site (dmacsassignments.github.com) by the due date using your own github used id. Combine all your insights, observation and sample outputs into one PDF file. The code and pdf together should be tarred and the submission should be named "name_mpi.tar". (Use just simple tar and not tar.bz2, tar.bz etc). *All students must work individually.*

Note: The files required for this assignments are provided at the "here" links. A file at this link can be downloaded on Lonestar using the command "wget link".

References:
1. Lecture slides used in the class (Par-Lab bootcamp): Tim Mattson
http://parlab.eecs.berkeley.edu/sites/all/parlab/files/MPI_bootcamp_2012.pdf
2.Introduction to MPI, TACC training material:
http://www.tacc.utexas.edu/c/document_library/get_file?uuid=6a3a0b19-d102-480f-b290-4a1cc8684f2d&groupId=13601
(slide 25 for a sample batch job template)
3. MPI Lab (TACC training material): http://www.tacc.utexas.edu/c/document_library/get_file?uuid=a7a38aa3-3c0e-4dbe-aa8e-179199aa0b23&groupId=13601
4.MPI Collective communication algorithms:
http://www.serc.iisc.ernet.in/~vss/courses/PPP/CollectiveCommunicationImplementations.ppt