CS546 "Parallel and Distributed Processing"
Homework 4

- ***Due date: 5:00pm on 04/15/2014.***
- ***You should work individually.***
- ***The instructor and TA are not responsible for debugging your code.***
- ***Please post your questions on the Blackboard.***
- ***Late penalty: 20% penalty for each day late***
- ***Please upload your assignment with a file name CS546_SectionNumber_LastName_FirstName_HW4.tgz on the Blackboard by due date. Please do NOT email your assignment to the instructor and TA!***

---

1. MPI Environment: the objective of this problem is to provide an introduction on compiling and running MPI programs.

- *Setting up your account:*

We will use the CS department's **jarvis** cluster for class assignments. You should have an account on the machine already. If you, for whatever reason, can't access the cluster, please contact the TA as soon as possible.

Step1 > ssh iit-username@jarvis.cs.iit.edu

- *What is MPI and which MPI are we using?*

Well, it's a long story. MPI stands for Message Passing Interface. It basically contains a communication library and other supportive software components. You can call message passing routines such as MPI_Send and MPI_Recv to deliver messages among different processes. MPI is an industrial standard created from the nationwide MPI-forum. More details of the MPI standard can be found at http://www.mpi-forum.org/docs/docs.html. Please, be aware that we are using MPI-3.0.

In terms of implementation, the communication library will actually use TCP/IP to communicate your data. You may wonder how MPI manages process creation and termination on different machines. This has a lot to do with the MPI design. Roughly speaking, the rlogin and rsh access commands allow MPI to execute processes on remote machines.

- *Compile and Run get_data.c*

The file get_data.c uses parallel trapezoidal rule to estimate the integral of function f(x)=x*x.

Input: a, b: limits of integration; n: number of trapezoids. Output: n trapezoids estimated where f(x) =x*x

Now you can write your MPI program and store it in your home directory. To compile the program, use mpicc command. For example, you can download a copy of program get_data.c to your directory. To create an object file get_data.o, you can use:

Step2 > mpicc -c get_data.c

Then to build an excutable file get_data, use:

Step3 > mpicc -o get_data get_data.o

Before running your program, you have to create a submit script file. This script file will be submitted to the queue manager on the cluster to run on the compute nodes. Let's call this script *run_get_data.bash*.

```
#!/bin/bash

mpirun –npernode 8 ./get_data
```

This script tells the cluster to run your code spawning 8 processes per node. This should be ideal, since the nodes in the cluster have 8 cores each. However, you can select any number of processes you want. For example, if you want to run your program with just 4 processes, you can specify 4. You can also use more processes than cores too.

Now you can run your code submitting it to the cluster as follows:

Step4 > qsub –p cs546_s14_project –pe mpich 2 run_get_data.bash

The first parameter indicated the project under which your code will run. In this case, you will run it under the project for this class. The next parameter indicated the parallel environment and the number of nodes that will be used (2*8 = 16 processes in total). For the parallel environment, we are using the mpich MPI implementation developed at Argonne National Laboratories. This implementation is open source, although it is not the only one out there (OpenMPI for example). The good news is that you don't need to worry about providing a machine file (IPs of machines where the code will run) to the parallel environment. The cluster will prepare all that for you.

You can check the progression of your job with *qstat*:

Step5 > qstat –u iit-username

When your job has finally run, you will get the results in 4 different files in your home directory:

run_get_data.bash.e1234
run_get_data.bash.o1234
run_get_data.bash.pe1234
run_get_data.bash.po1234

These 4 files correspond to application's standard error, application's standard output, parallel environment's standard error, and parallel environment's standard output. The numbers attached to the end correspond to the job ID. Application's standard error is everything your print to the special file descriptor *stderr:*

fprintf (stderr, …)

Application's standard output is everything you print to the special file descriptor *stdout*:

fprintf (stdout,…)
or

printf (…)

If everything goes well, your results should be printed to the right file:

$ cat run_get_data.bash.o1234
with n = 100 trapezoids, our estimate
.....

You need to modify 3 macros at the beginning of the code before compiling and running if you want to use different values for the parameters A, B and N. For example:

```
#define A 0
#define B 1
#define N 100
```

- *Modify get_data.c*

  In program get_data.c, it is process0 who is going to collect data from other processes. **You are required to modify the program** to make the process with the largest rank to collect data and output the final result.

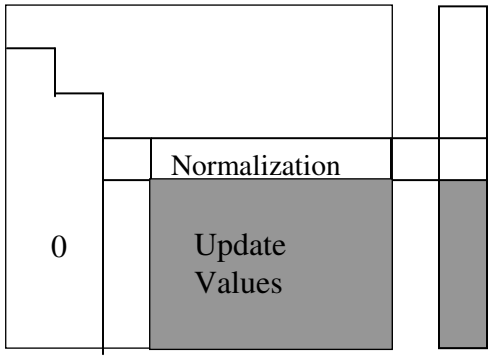
2. In this problem you are asked to write a parallel algorithm using MPI for solving a set of dense linear equations of the form A*x=b, where A is an n*n matrix and b is a vector. You will use Gaussian elimination without pivoting. You had written a shared memory parallel algorithm in previous assignments (using pthread and OpenMP). Now you need to convert the algorithm to a message passing distributed memory version using MPI.

   Assume that the data for the matrix A and the right hand-side vector x is available in processor 0. You can generate the data randomly as was done in HW2.

   But note that you will need to distribute the data to all other processors. Finally the results will have to be collected into processor 0 which will print the final results.

   The algorithm has two parts:
   - *Gaussian Elimination*: the original system of equation is reduced to an upper triangular form Ux=y, where U is a matrix of size N*N in which all elements below the diagonal are zeros which are the modified values of the A matrix, and the diagonal elements have the value 1. The vector y is the modified version of the b vector when you do the updates on the matrix and the vector in the Gaussian elimination stage.
   - *Back Substitution*: the new system of equations is solved to obtain the values of x.

   The Gaussian elimination stage of the algorithm comprises (N-1) steps. In the algorithm, the ith step eliminates nonzero subdiagonal elements in column i by subtracting the ith row from row j in the range [i+1,n], in each case scaling the ith row by the factor A_{ji}/ A_{ii} so as to make the element A_{ji} zero. See figure below:



Hence the algorithm sweeps down the matrix from the top corner to the bottom right corner, leaving subdiagonal elements behind it.

(Part a) Write a parallel algorithm using MPI using static interleaved scheduling. The whole point of parallel programming is performance, **so you will be graded partially on the efficiency of your algorithm.**

Suggestions:

- Consider carefully the data dependencies in Gaussian elimination, and the order in which tasks may be distributed.
- Gaussian elimination involves $O(n^3)$ operations. The back substitution stage requires only $O(n^2)$ operations, so you are not expected to parallelize back substitution.
- The algorithm should scale, assuming N is much larger than the number of processors.
- There should be clear comments at the beginning of the code explaining your algorithm.

(Part b) Report running times for 1, 2, 4, 8, 12 and 16 processors. Evaluate the performance of the algorithm for a given matrix size (2000*2000) on the cluster. USE THE MPI TIMERS FOR PORTABILITY, i.e. the MPI_Wtime() calls even though this measures elapsed times and not CPU times.