CS546 “Parallel and Distributed Processing”
Homework 4

Submission:

- ***Due by 11:59pm of 04/21/13***
- ***Late penalty: 10% penalty for each day late***
- ***This is a programming assignment.***
- ***Please upload your assignment on the Blackboard with the following name: CS546_SectionNumber_LastName_FirstName_HW4. The submission should include:) the C source files, binary file, README.txt file***
- ***Please do NOT email your assignment to the instructor and TA!***

1. MPI Environment setup: the objective of this problem is to provide an introduction on setting up the MPI programming environment.

- *Setting up your account:*

We will use the CS department's **csrocks** cluster for class assignments. The cluster consists of one server machine (csrocks.cs.iit.edu) and 15 computer nodes (compute-0-0 to compute-0-14). Every node is connected through Ethernet and runs the Centos linux OS. They use the same Network File System. This system configuration serves well for programming assignments. However, due to limited memory space, it is not recommended to run other applications on the nodes while you are running your MPI programs.

Your class account has already been assigned to you. You can use it to run this assignment and get used to the system. You are not allowed to run MPI program on csrocks server itself. So first login to **csrocks.cs.iit.edu** and then **ssh to compute-0-0 (or compute-0-2, compute-0-7, compute-0-9)**.

Step1 > ssh username@csrocks.cs.iit.edu
Step2 > ssh compute-0-#

The MPI path is /share/apps/mpich2/bin. Please, set your path variable to point to mpich-3.0.3 first (if not, the openmp implementation will be used):

> export PATH=/share/apps/mpich2/bin:$PATH

Be aware that you will need to change PATH every time you login to csrocks. In order to avoid that, you can add the above like to you .bashrc file located in your home directory:

> echo export PATH=/share/apps/mpich2/bin:$PATH >> ~/.bashrc

- *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.

The utilities you will use to compile and run your programs are in */share/apps/mpich2/bin* directory.

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:

Step3 > mpicc -c get_data.c

Then to build an excutable file get_data, use:

Step4 > mpicc -o get_data get_data.o

Before running your program, you have to create a machine file, a list of machines on which you want to run your application processes. For examples, you may have a file called mymachines containing:

```
compute-0-0
compute-0-2
compute-0-7
compute-0-9
```

This file tells the MPI execution environment that the application processes can be executed on these machines. The allocation of the processes to the machines is up to the MPI execution environment. We don't need to consider resource management at this time.

Now that you get an executable get_data, you can use the mpirun command to run it as follows:

Step5 > mpirun -f mymachines -np 4 ./get_data

The command tells MPI that get_data will be executed on a machine pool (as listed in the mymachines file). The option -np 4 indicates that 4 processes will be executed concurrently. The number of machines in mymachine file and the number of processes indicated by the -np option does not have to match each other. MPI will decide the number of processes to be put on each machine. Some machines could host more processes than the others. You will get something like this as the results:

```
mpirun -machinefile mymachines -np 4 get_data
Enter a, b, and n
0 1 100
with n = 100 trapezoids, our estimate
..................................................
```

MPI User's Manual is located at http://www.mpich.org/static/docs/guides/mpich2-1.5-userguide.pdf

- *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.

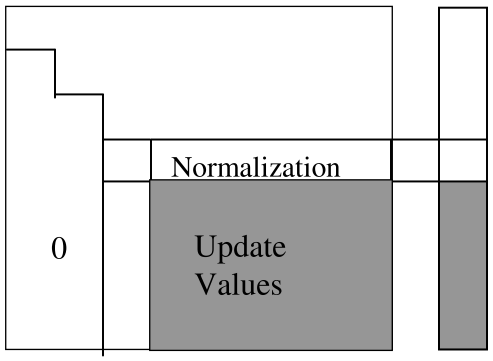
2. In this problem you are asked to write a parallel algorithm using MPI for solving a set of dense linear equations of the form A*x=b, where A is an n*n matrix and b is a vector. You will use Gaussian elimination without pivoting. You had written a shared memory parallel algorithm in HW1 and a pthread program in HW2. 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, and 12 processors. Evaluate the performance of the algorithm for a given matrix size (1000*1000) 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.