ECE 451/566 Introduction for Parallel and Distributed Programming

Assignment 1: MPI Programming and Performance Evaluation

Due: Wednesday, October 23, 20013

Important notes on submission:

- Assignments are individual efforts. SUBMIT YOUR OWN WORK!!!
- Submission of the assignment is electronic (send it using **Sakai**).
- Any submission received beyond midnight on the due date will be considered late and will be graded with a 25% penalty. Submissions more than 2 days late will not be graded.
- The final submission should be a single zipped file named as ECE###_YourFirstName_YourLastName_Assign#.zip. To submit your assignment, place all files and programs into a single directory named ECE###_YourFirstName_YourLastName_Assign# (eg. - ECE451_John_Doe_Assign1 or ECE566_Jane_Smith_Assign1) and zip the folder into a single zipped file. Make sure you follow this naming convention.
- For programming questions: Within the zipped contents, supply your source codes as separate files (not pasted in the assignment solution file) using an appropriate name to identify the program source file. Comments explaining your logic make programs readable and easy to understand and are a good programming habit. For any programming question, supply a snapshot of the user input (if needed) and output obtained in the assignment solution file along with analysis and graphs (if needed).
- Clearly state any assumptions that you make for any program or theory question.

Instructions:

PART 1: MPI ping-pong (**20%**)

The first part of this assignment consists on familiarizing with the environment and being able to compile and run MPI programs. Steps to compile and run a simple "hello world" program in MPI implementing message-passing between two processes are listed below:

1. Download the `hello.c` and `hello_any.c` source codes.
2. Login to `excalibur` or `ee103-pc24.engr.rutgers.edu` (`172.16.69.38`) (you can monitor using Ganglia at: `http://ee103-pc24.engr.rutgers.edu/ganglia/`).
3. Compile your `hello.c` program with `mpicc` (see course slides for further detail).
4. Run the MPI hello world by using "`mpirun -np NUM_PROCS BINARY`".
5. Use different options such as "`-nolocal`", "`-machinefile machines.txt`", etc.

Example of `machines.txt`:

```
compute-0-0
compute-0-2
compute-0-3
...
compute-0-16
```

A key issue in the performance analysis of programs for a distributed memory parallel system is the cost of communication relative to the cost of computation. Write an MPI program to experimentally estimate the costs of computation and communication. The cost of communication between two MPI processors can be approximately divided into two parts:

(i) Start-up time `Tstartup` and
(ii) Transmission Time `Tt = (Data size)/Bandwidth`

Start-up time is the duration needed for MPI to set up the communication links. Transmission time is the time needed to send the message to its destination. The most commonly used model of the cost of communication is linear:

```
Tcommunication = Tstartup + (Data size)/Bandwidth
```

Ping-pong messaging is often used to measure communication times. In this measurement, messages are repeated sent from process 0 to process 1 and back to process 0 as shown in the code below:

```
/* Ping-Pong message */
if (myRank == 0)  {
    /* send to process 1 */
    MPI_Send(message, count, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

    /* receive from process 1 */
    MPI_Recv(message, count, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &status);

}  else /* myRank == 1 */
{
    /* receive from process 0 */
    MPI_Recv(message, count, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

    /* send to process 0 */
    MPI_Send(message, count, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
 }
```

You can estimate `Tstartup` and the bandwidth by sending messages of various sizes and using the least squares to fit a line to the data. Computational time can be measured as follows:

```
float x, y, z;
double start, elapsed;
start = MPI_Wtime();
z = x * y;                  // DO YOUR OWN COMPUTATION HERE
elapsed = MPI_Wtime() - start;
```

The resolution of `MPI_Wtime()` is given by the MPI function

```
double MPI_Wtick(void);    /* for MPI time resolution */
```

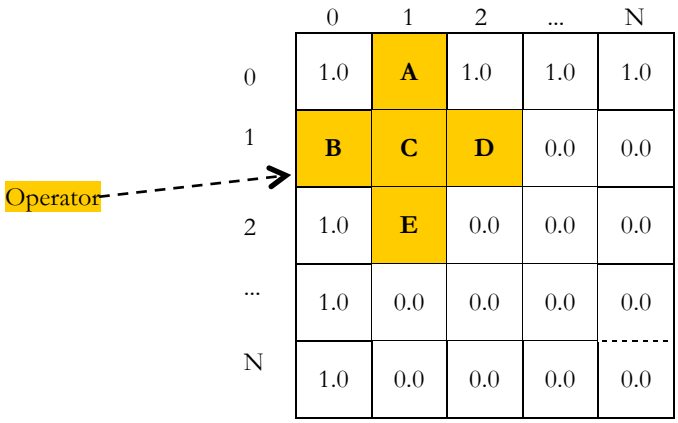
Note that if the computation takes less time than the resolution, `MPI_Wtime()` will return zero. As a result, to measure the timings, you will have to execute the operations repeatedly and compute the average time. For computation timings, you should use arrays of very large size rather than scalars (single variables).

Output: The program should print the average cost of a computation, and a series of average costs for communication: the cost for a message of 0 byte, 1 byte, 2 bytes ... 4 bytes. 2^{20} bytes. There will be no input to the program. Use these measurements to compute `Bandwidth` and `Tstartup`. You might want to make a scatter plot of your results. Include a discussion of your results. The discussion should include the following:

(a) How did you measure the computation times? Did you consider the loop overhead? Why?
(b) What do you think is the effect of the call to `MPI_Wtime()` itself? Does it skew the result (i.e. what about the time spent on calling `MPI_Wtime()`)?
(c) Was there much variance in the ping-pong times for messages of fixed size? If so can you offer a suggestion about why this might be the case?

PART 2: MPI solver (**80%**)

The second part of this assignment consists on parallelizing a sequential program that implements a solver. First, the program initializes a matrix and then applies an operator over the whole matrix a number of times looking for convergence (see figure below and convergence details in the provided code). Note: for every element, the operator applies the formula C'=0.2(C+A+B+D+E).



A sequential program that implements the solver is available in the course web site.

For this part of the assignment:

1. Implement the solver using MPI. Explain your decisions/observations. **(20%)**
2. Evaluate the performance of the application in terms of: **(20%)**
    a. Its scalability with the number of cores used.
    b. Percentage of the parallel fraction.
    c. Imbalance and overhead (e.g., messages).
3. Compare different data partition strategies and problem (matrix) sizes. **(20%)**
4. Explore different types of MPI calls (e.g., asynchronous, buffered) and compare with your initial implementation (e.g., scalability, percentage of the parallel fraction). **(20%)**

Explain your observations and provide plots for your performance evaluation.