Programming Projects and Related Materials

Experimental platforms:

The main experimental platform for this class is the *Pleiades* cluster.

The *Pleiades* cluster is an 8-node, 64-core cluster. Each compute node has 8 Intel Xeon cores and 4GB of shared RAM.

Pleiades cluster: Please follow these instructions while using the Pleiades cluster.

Recommended usage: Please use the Pleiades cluster as your first preference. If you want to use a different cluster that you already have access to, that is fine as long as: a) the cluster is comparable (if not larger) in its configuration compared to Pleiades; and b) you log in your project report the system configuration of that cluster (under experimental setup section).

Examples:

- MPI Hello World helloworld.c
- MPI Send Receive test send_recv_test.c
- OpenMP: Simple for loop parallelization: loop.c
- OpenMP: Sum of n numbers using p threads: sumcomp.c
- OpenMP: Matrix vector parallelization using p threads: matrix_vector.c
- OpenMP: Synchronization primitives (critical section, atomic, locks): sync.c

PS: To compile an OpenMP code use:

```
gcc -o {execname} -fopenmp {source file names}
```

COVER PAGE: PDF Word
(please include this with every project and homework submission)

PROGRAMMING PROJECTS

- **Programming Project #1:** Due September 17, 2019 (11:59pm PDT)

Assignment type: Team of size up to 2 encouraged
Where to submit? Blackboard dropbox for Programming Project #1

The goal of this project is to empirically estimate the network parameters **latency** and **bandwidth**, and the **network buffer size**, for the network connecting the nodes of the compute cluster. Note that these were described in the class in detail (refer to the lectures on August 27th and September 3rd).

You are expected to the Pleiades cluster for this project.

To derive the estimates write a simple MPI send receive program involving only two processors (one sends and the other receives). Each MPI *send communication* should send a message of size *m* bytes to the other processor (which will invoke *receive*). By increasing the message size *m* from 1, 2, 4, 8, ... and so on, you are expected to plot two runtime curves, one for send and another for receive. The communication time is to be plotted on the Y-axis and message size (m) on the X-axis. For the message size, you may have to go on up to 1MB or more to observe a meaningful trend. Make sure you double *m* for each step.

You will need to test using two implementations:

Blocking test: For this, you will implement the sends and receives using the blocking calls MPI_Send and MPI_Recv. As an alternative to MPI_Send and MPI_Recv, you are also allowed to use MPI_Sendrecv. This is blocking as well.

Nonblocking test: For this, you will implement the receives using the nonblocking call MPI_Irecv. For the send, it doesn't matter whether you use MPI_Send or MPI_Isend - it shouldn't matter much as explained in the class.

Please refer to the API for MPI routines for function syntax and semantics. An example code that does MPI_Send and MPI_Recv along with timing functions is given above (send_recv_test.c). You can reuse parts of this code as you see fit. Obviously modification is necessary to fit your needs.

Deriving the estimates: From the curves derive the values for latency, bandwidth and network buffer size. To ensure that your estimates are as precise and reliable as they can be, be sure to take an average over multiple iterations (at least 10) before reporting. Report your estimated values, one set for the Blocking Test, and another for the Nonblocking Test. (It is possible you might get slightly varying values between the two tests.)

Deliverables (zipped into one zip file - with your names on it):
Note, for those of you who worked in teams of size 2, both of you should submit, but only one of you should submit the full assignment along with the report and cover page stating who your other partner was, and the other person simply submits the cover page.
i) ***Source code*** with timing functions,
ii) ***Report*** in PDF that shows your tables and charts followed by your derivation for the network parameter estimates. Make sure add a justification/explanation of your results. Don't dump the raw data or results. Your results need to be presented in a professional manner.

- **Programming Project #2:** Due October 3, 2019 (11:59pm PDT)

Assignment type: Team of size up to 2 encouraged
Where to submit? Blackboard dropbox for Programming Project #2

For a detailed description of this project, please refer to this PDF.

- **Programming Project #3 - PARALLEL PSEUDORANDOM NUMBER GENERATION:** Due November 5, 2019 (11:59pm PDT)

(Total points: 20)
Assignment type: Team of size up to 2 encouraged
Where to submit? Blackboard dropbox for Programming Project #3

In this project you will implement a parallel **pseudo-random number series** generator, using the Linear Congruential Generating model we discussed in class. Here, the ith random number, denoted by x_i, is given by:
$x_i = (A*x_{i-1} + B) \bmod P$, where, A and B are some positive constants and P is a big constant (typically a large prime); all three parameters {A,B,P} are user-defined parameters.
Your goal is to implement an MPI program for generating a random series up to the nth random number of the linear congruential series (i.e., we need *all* n numbers of the series, and not just the nth number). We discussed an algorithm that uses parallel prefix in the class. You are expected to implement this algorithm. Refer to the Lecture notes website to look for more references if you need.
Operate under the assumption that n>>p. Your code should have your own explicit implementation the parallel prefix operation. Your code should also get parameter values {A,B,P} and the random seed to use (same as x_0), from the user.
All the logic in your code for doing parallel prefix should be written from scratch. Use of MPI_Scan is *not* allowed for doing parallel prefix. Write your own parallel prefix function. I have already provided a lot of tips in the class.
It should be easy to test your code by writing your own simple serial code and comparing your output. If your parallel implementation is right, its output should be identical to that of the serial output (for the same parameter setting).
It is expected that your code has two serial implementations:
i) serial_baseline: which is a function that simply is the most straightforward serial implementation that directly implements the mathematical recurrence using a simple for loop.
ii) serial_matrix: which is a function that is also a for loop but it uses the matrix formulation (in other words, this is equivalent to your parallel algorithm on a single process).

Both these serial baseline pseudocodes are provided in the lecture notes. The above two versions are important for correctness check.

Performance analysis:

a) Study total runtime as a function of n. Vary n from a small number such as 16 and keep doubling until it reaches a large value (e.g., 1 million).
b) Generate speedup charts (speedups calculated over your serial_baseline implementation), fixing n to a large number such as a million and varying the number of processors from 2 to 64.

Compile a brief report that presents and discusses your results/findings. Quality of the presentation style in your report is important.

Deliverables (zipped into one zip file - with your names on it):
Note, for those of you who worked in teams of size 2, both of you should submit, but only one of you should submit the full assignment along with the report and cover page stating who your other partner was, and the other person simply submits the cover page.
i) Cover page
ii) Source code,
iii) Report in PDF
Name your zip folders as Project3_MemberNames.zip. No whitespace allowed in the folder or file names. Follow the naming convention strictly. Otherwise you stand to lose points.
Rough grading rubric: code (30%), testing (40%), reporting (30%)

- **Programming Project #4: Pi Estimator (using OpenMP multithreading):** Due November 14, 2019

Total points: 10
Assignment type: Team of size up to 2 encouraged
In this project you will implement an OpenMP multithreaded PI value estimator using the algorithm discussed in class. This algorithm essentially throws a dart *n* times into a unit square and computes the fraction of times that dart falls into the embedded unit circle. This fraction multiplied by 4 gives an estimate for PI.
Here is the generation approach that you need to implement: PDF

Your code should expect two arguments: <n> <number of threads>.
Your code should output the PI value estimated at the end. Note that the precision of the PI estimate could potentially vary with the number of threads (assuming n is fixed).
Your code should also output the total time (calculated using *omp_get_wtime* function).

Experiment for different values of n (starting from 1024 and going all the way up to a billion or more) and p (1,2,4..).

Please do two sets of experiments as instructed below:

1) For speedup - keeping n fixed and increase p (1,2,4, 8). You may have to do this for large values of n to observe meaningful speedup. Calculate relative speedup. Note that the Pleiades nodes have 8 cores per node. So there is no point in increasing the number of threads beyond 8. In your report show the run-time table for this testing and also the speedup chart.
PS: If you are using Pleiades (which is what I recommend) you should still use the Queue system (SLURM) to make sure you get exclusive access to a node. For this you need to run "sbatch -N1" option (i.e., run the code on a single node).

2) For precision testing - keep n/p fixed, and increase p (1,2,.. up to 16 or 32). For this you will have to start with a good granularity (n/p) value which gave you some meaningful speedup from experiment set 1. The goal of this testing is to see if the PI value estimated by your code improves in precision with increase in *n*. Therefore, in your report make a table that shows the PI values estimated (up to say 20-odd decimal places) with each value of n tested.

Deliverables (zipped into one zip file - with your names on it):
Note, for those of you who worked in teams of size 2, both of you should submit, but only one of you should submit the full assignment along with the report and cover page stating who your other partner was, and the other person simply submits the cover page.
i) Cover page
ii) Source code,
iii) Report in PDF
Name your zip folders as Project4_MemberNames.zip. No whitespace allowed in the folder or file names. Follow the naming convention strictly. Otherwise you stand to lose points.
Rough grading rubric: code (40%), testing (20%), reporting (40%)