Assignment 4

Code is included under HW4 on our class Github site.

Use this code as a basis for a calculation of PI, using 100000000 iterations of Nilakantha's accelerated series.

Convert the code in 4 ways to see which is fastest and most efficient method for this code. Make 3 versions of the pi.cpp code, call them pi1.cpp, pi2.cpp, pi3.cpp and pi4.cpp. You can use "cp pi1.cpp piX.cpp". Keep the same calculation for pi1-3, but modify it to be spread across n number of processors. With pi4.cpp, you can use any method you like, including changing the calculation as long as it works. Only pi4.cpp could have the calculation changed, other than just breaking it up across processors.

The versions should use the following methods in OpenMPI:

- pi1.cpp with two-sided communication (OpenMPI-2)
- pi2.cpp with one-sided communication (OpenMPI-3 one-sided with fencing)
- pi3.cpp with one-sided communication (OpenMPI-3 one-sided without fencing)
- pi4.cpp with a method of your own choosing (could be newer OpenMPI, OpenMP, Pthreads, etc.)

All of the items above should be run across 1, 12, 24 and 48 processors. You will end up with 16 timed results (runtime) to log, along with 16 error rates. You can use my results below to compare the default code which required 400 ms.

Example of the unmodified code running on Bridges with 1 processor:

```
$ g++ pi.cpp -o pi -m128bit-long-double      # to compile
$ srun ./pi 800000  # 800K is about as large a number as we can use with a 128b long double. Note:
                      eventually you may need other methods to run it for best results

srun: job 23351996 queued and waiting for resources
srun: job 23351996 has been allocated resources
PI is approx 3.14159265358979321747928681318740018468815833330154, Error is
0.00000000000000001014813233446432150230975821614265 4
14.9684 Runtime ms
```

The time is nearly 15 ms to compute with 800,000 iterations of Nilakantha's accelerated series for PI calculation.

See the papers listed at the bottom for good reference material.

Items to turn in:

- All code written/modified and used in the testing
- Any required commands to compile and run on Bridges
- Your code should compile and run based on your documentation
- A 2+ page write-up on the methods applied in each of your pi1-4 programs
- What differences you saw in performance between them
- Which method and number of processors (1, 12, 24, 48) provided the best performance on Bridges2
- Explain what decisions you made and why you chose the methods that you picked
- Include a chart (can be done in Excel and do a copy/paste into your document, or could be hand-drawn if required) of the performance differences between the program runs including the unmodified code
- Include any reference material you used (nothing formal, links are fine) to determine methods to use. If AI helped, let me know about it and how helpful it was

Your grade will be based on following the requirements (one-sided, two-sided, other method) and increases in performance. You should be able to find a way to increase the performance well beyond the un-modified code. I recommend you start off with lower numbers of processes before increasing to higher numbers of processes. Lower numbers of processors also get through the HPC queue faster. You can test with 1-4 processors which will enable lower queue times until you have something viable to test with increasing numbers of processes.

The following academic papers may help. They have some example pseudo code for multiple one and two-sided communication using OpenMPI.

An Evaluation of Implementation Options for MPI One-Sided Communication

https://web.cels.anl.gov/~thakur/papers/rma-impl.pdf

Covers options such as: fence synchronization, post-start-complete-wait synch, and lock-unlock synch. Has a good and simple explanation of one-sided communication and discusses the pros/cons of each implementation.

An Implementation and Evaluation of the MPI 3.0 One-Sided Communication Interface

https://www.mcs.anl.gov/uploads/cels/papers/P4014-0113.pdf

This paper discusses one-sided vs two-sided communications. On page 14-15 there is a comparison of code utilizing the MPI-2 exclusive lock “Epoch” method vs the MPI-3 Shared-lock “Accumulate” method.

One-Sided Interface for Matrix Operations using MPI-3 RMA: A Case Study with Elemental

https://pavanbalaji.github.io/pubs/2016/icpp/icpp16.elemental.pdf

This paper has a lot of charts showing the performance difference between different types of communication.