Lab Report 2

Parallelism with MPI

CSC 452-001

Andrew Almond

10/08/19

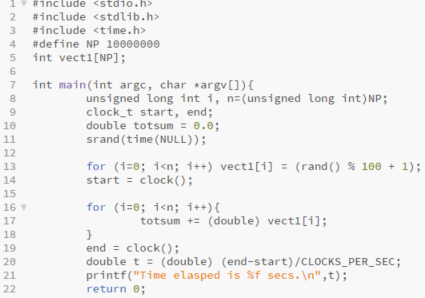
**Introduction**

For the majority of the time of computers, sequential programming dominated most of all computational programs. However, today, distributed systems have been making a dramatic climb in use. Because of this, parallel programs are in high demand now. One example of this is MPI that this lab covered. Its use allows developers to write code for one, ten or hundreds of programs to be executed on various grand amounts of systems at the same time. This is done through an approach covered before SPMD (single data, multiple data). Using MPI through this lab, a greater understanding of communication between threads, reduction operations, SPMD programs and more can be made.

**Methods & Algorithms**

Question 1:

Sequential C Code: In the given C program, a vector is made of NP = 10 million numbers valued (random# % 100)+1. The program then starts a timer while the total sum is added up. The total time is then made by the difference in the end time and start time divided by the # of clocks per second.



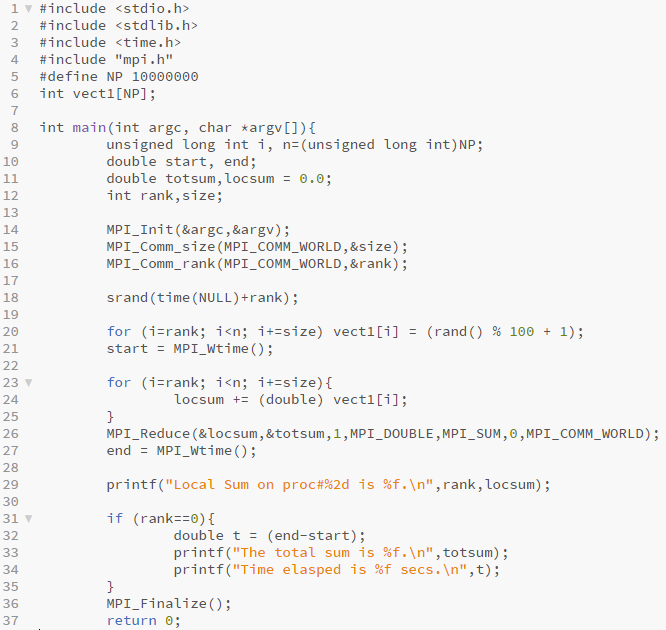
Question 2:

MPI Hello World: In this MPI program, we include MPI routines to initialize the parallel threads; have each thread find its own rank and the size of the pool, then print its message. The “MPI\_Barrier” command acts as a wall to prevent the first done threads from ending until the last ones finish so that a completion time can be collected. This then prints it at command line and terminates the program.



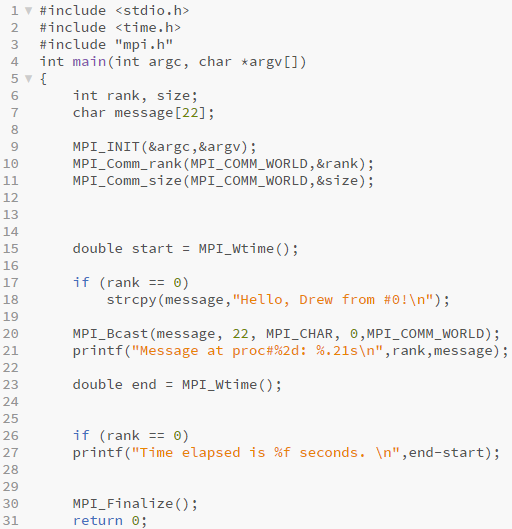
Question 3:

MPI Distributed Summation: In this MPI program, like program 1, an array of 10 million integers is initialized across each parallel thread given through a striped manner. The local sum (locsum) is computed with a “for” loop for each thread where it iterates by the pool size. Using the “MPI\_Reduce” (reduction) function, each thread will combine its result with the rest as the “MPI\_SUM” function. All of the local sums, the total sum and the completion time are printed before exiting the loop.



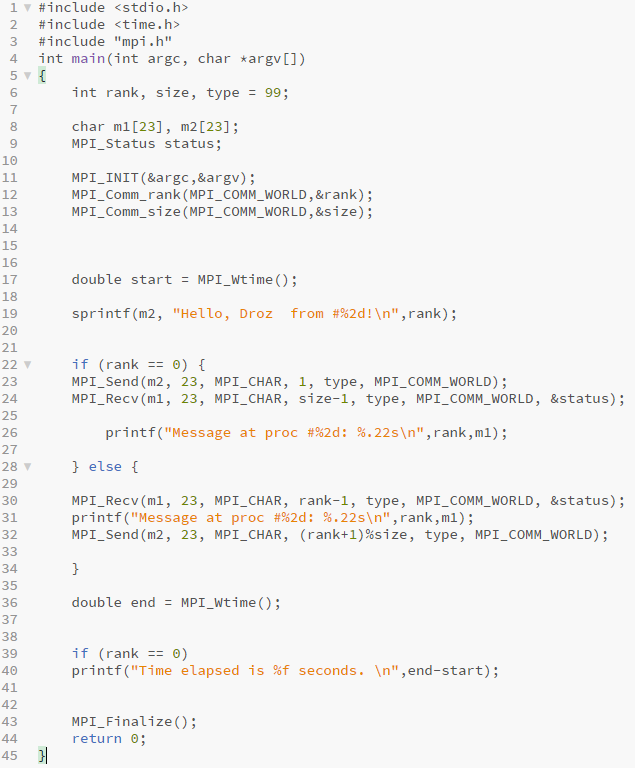
Question 4:

Distributed Broadcast: Unlike the previous programs, this one doesn’t have each thread print off their own ranks. In this case, the thread with rank 0 will broadcast its message to the remaining threads. With this, the other threads will follow and print the broadcasted message before the pool ends.



Question 5:

Distributed MPI Ring: This MPI program utilizes ring communications. However, deadlock should be avoided. To prevent deadlock, thread 0 will send its message to thread 1 upon starting before it tries to receive. All the others will receive their message before they receive. The messages will be received from thread #(n-1), printed off and sent to thread #(n+1)mod0. Once all this is over, the final execution time is displayed.



**Experiments**

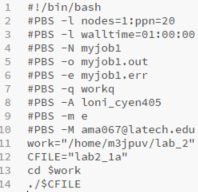
All of the following tests were run on the LONI cluster with 1 node, 20 physical cores and 64 GB of RAM. All lab programs but program 1 required changes in one variable, which was the number of threads to utilize.

|  |  |  |  |
| --- | --- | --- | --- |
| **File Name** | **Variables** | **Job Script** | **Job Output** |
| lab2\_1a.c | Npoints = 1E7 | submit\_lab2\_1a | myjob1a.out |
| lab2\_2.c | Nthreads = 4, 8 | submit\_lab2\_2 | myjob2.out |
| lab2\_3.c | Nthreads = 4, 8 | submit\_lab2\_3 | myjob3.out |
| lab2\_4.c | Nthreads = 4, 8 | submit\_lab2\_4 | myjob4.out |
| lab2\_5.c | Nthreads = 4, 8 | submit\_lab2\_5 | myjob5.out |

**Results**

Before running every program besides the one from question 1 through the LONI cluster using “qsub script\_name”, the code should be compiled using “mpicc -o lab2\_x lab2\_x.c”.

Question 1:

A screenshot of a cell phone

Description automatically generated Submit Script:

|  |  |
| --- | --- |
| **t** | 0.02 |

Question 2:

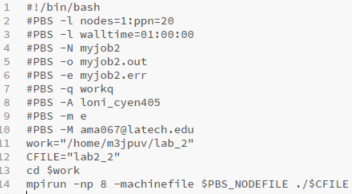
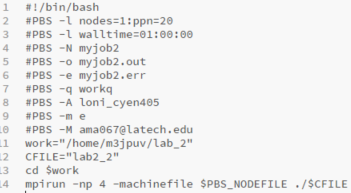
For the solutions to questions 2 through 5, the submit script should be altered on the value of np on the bottom like between 4 threads or 8 threads.

A screenshot of text

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Submit Scripts:



|  |  |  |
| --- | --- | --- |
|  | **4 Threads** | **8 Threads** |
| **t** | 0.000133 | 0.000065 |

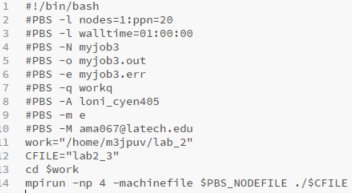
A screenshot of text

Description automatically generatedQuestion 3:

A screenshot of a cell phone

Description automatically generated

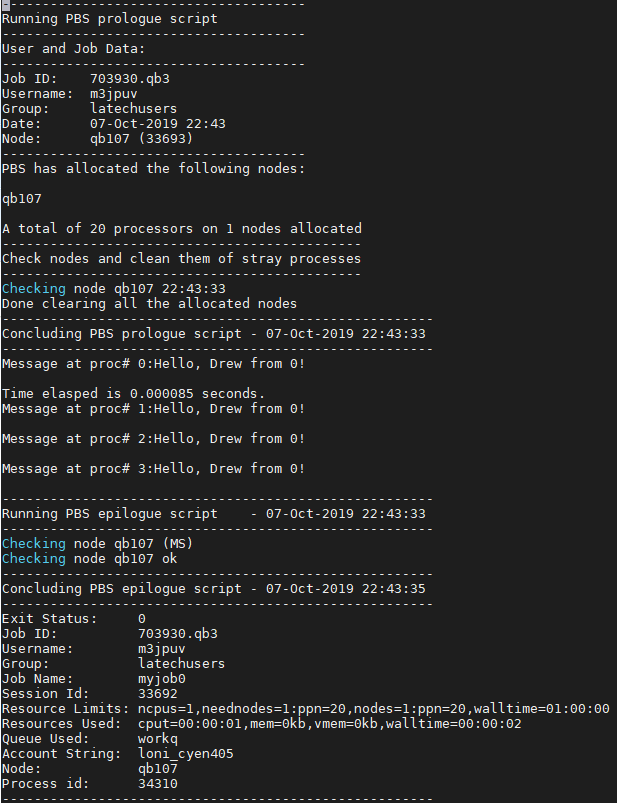
Submit Scripts:



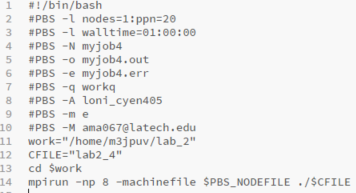
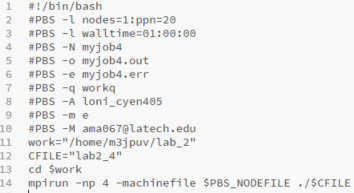
|  |  |  |
| --- | --- | --- |
|  | **4 Threads** | **8 Threads** |
| **t** | 0.004615 | 0.007412 |

A screenshot of a cell phone

Description automatically generatedQuestion 4:



Submit Scripts:



|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | **4 Threads** | **8 Threads** |
| **t** | 0.000085 | 0.007412 |

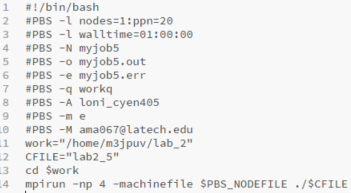
A screenshot of a cell phone

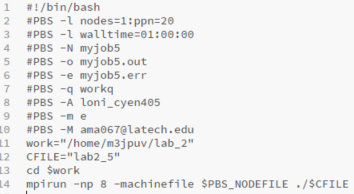
Description automatically generatedQuestion 5:

A screenshot of a cell phone

Description automatically generated

Submit Scripts:





|  |  |  |
| --- | --- | --- |
|  | **4 Threads** | **8 Threads** |
| **t** | 0.000144 | 0.000221 |

**Conclusion:**

Given the results from each use of MPI, an assumption could be made on which iterations seem to be more efficient for the given number of threads. In these cases, there are comparisons between each thread printing their own message, printing a received broadcast or receiving, printing and sending messages. Each has its own advantages and disadvantages, but the solution for lab2\_5 happened to be the best for speed due to the passing of messages. Given more time in the future, a further understanding and adaptation of such programming could be discovered, increasing speed and efficiency through parallelism.

**Github Link:**

<https://github.com/M3JPUV/CSC452_Labs.git>