Andrew Sexton (asexton2)

CSC 458 Project 3

March 27, 2020

Run Instructions

There are three different make files corresponding to each type of parallelization.

For pthreads and openMP:

Compile:

```
make -f Makefile.pthread

make -f Makefile.omp
```

Run:

```
./p_gauss -s 1024 -n 4

./omp_gauss -s 1024 -n 4
```

-s: size of matrix (default is 128)

-n: number of processors (default is 1)

For MPI:

Compile:

```
module load mpi          (required before make)

make -f Makefile.mpi
```

Run:

```
mpirun -np 4 ./mpi_gauss -s 1024
```

-np: number of processes to run (required)

-s: size of matrix (default is 128)

Explanation:

The purpose of this project was to implement MPI on the Gaussian code from project 1, as well as implement OMP and compare each version along with pthreads.

The pthread code iterates through every row of the matrix, and spawns threads that share memory to do computations on a defined subset of the remaining rows.

OMP is identical to the sequential version with the exception that the inner two for loops of the "computeGauss()" function are wrapped in an OMP Pragma block to break the work up among the number of processes specified.

MPI required the most modification to the code. The way I implemented it was to iterate over every row of the matrix, the root process then computes the pivot, and then use the "MPI_Scatterv" function to spread the remaining rows to every other process, along with a few other needed variables. The processes then each did work on the sub-matrix they were given, and the matrix was put back together with the "MPI_Gatherv" function. This was tested both on a system with a large number of processor cores, as well as over the network across six nodes.

<u>Difficulties:</u>

OMP was the easiest to implement, as it only required minor modification of the code. Pthread was already implemented from a previous assignment, although some bugs were fixed that increased performance very slightly.

MPI was extremely difficult to get working correctly. One of the biggest challenges with MPI is that, while it utilizes multiple processes to achieve parallelization, the code must be sequential. This means that every process made by the MPI library follows the same code as all the others, but must also have a way to determine the unique segments of the work that each process must work on. For this, the root process does some specific work, while all the other processes must receive the subsets to work on via message passing.

Although it took much time to implement correctly, MPI also suffers from significantly slow performance time than the other two, especially on larger array sizes. I am not clear if this is due to bad implementation on my part or due to inherent slowness in the MPI message passing when trying to move large amounts of data around.

The MPI version was so slow, that I did not run it for the 2048 matrix, because it was already taking over ten minutes to complete at 1024. I've included the times below for the MPI on the "node2x18a". As seen, for a single process the time is not terrible, the same for the 128 matrix. However, the times grow almost exponentially as more processes are used.

The 1024 times go from 1.5 seconds with one process, to 73 seconds with two, to 10 minutes with four or more. Again, I am not sure what the reason or this is.

MPI	1	2	4	8	16	32	64
128	0.002604	0.174206	0.248616	0.251699	0.286855	0.301815	0.390725
256	0.015381	1.283718	28.15416	20.80185	63.96348	82.33935	104.4227
512	0.114412	9.431424	107.3335	121.3248	126.4456	204.7211	230.5949
1024	1.452592	73.9568	635.5891	583.9067	583.8104	581.2718	673.512

Another issue is that MPI only worked when the number of processes were powers of 2. My home desktop has six cores, but this would cause SEGFAULT errors when six cores were specified. However, I was able to run it with four processes.

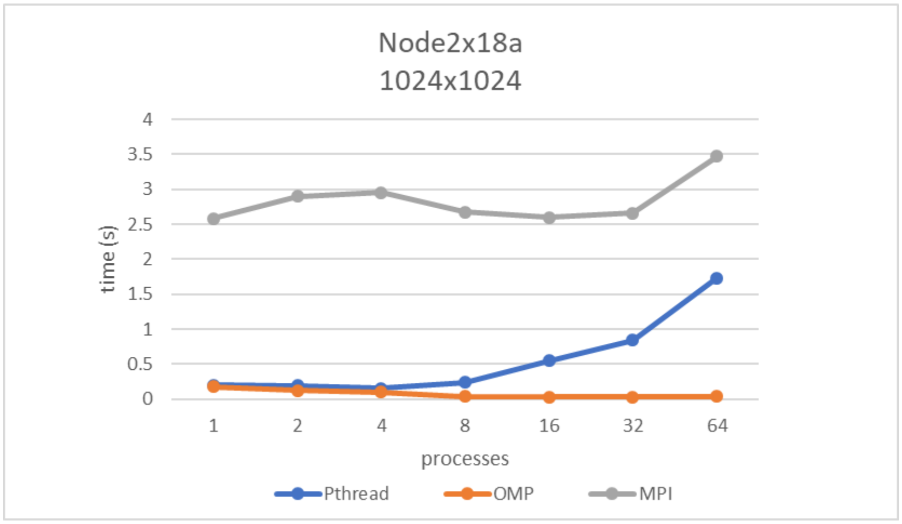
Results:

For this assignment I tested the code on three systems. First was my home desktop which has a AMD Ryzen 1600 6 core/12 thread processor. Second was the "node2x18a" computer which has two Intel Xeon processors with a total of 72 cores. Lastly was over the network on the computers "node01" through "node06", each of which has one Intel Xeon processor with 6 cores/12 threads each.

As stated above, one issue with the MPI code was that it failed if the number of processes specified was not a power of two. To be consistent, I only tested each version with numbers of cores that met this odd requirement.

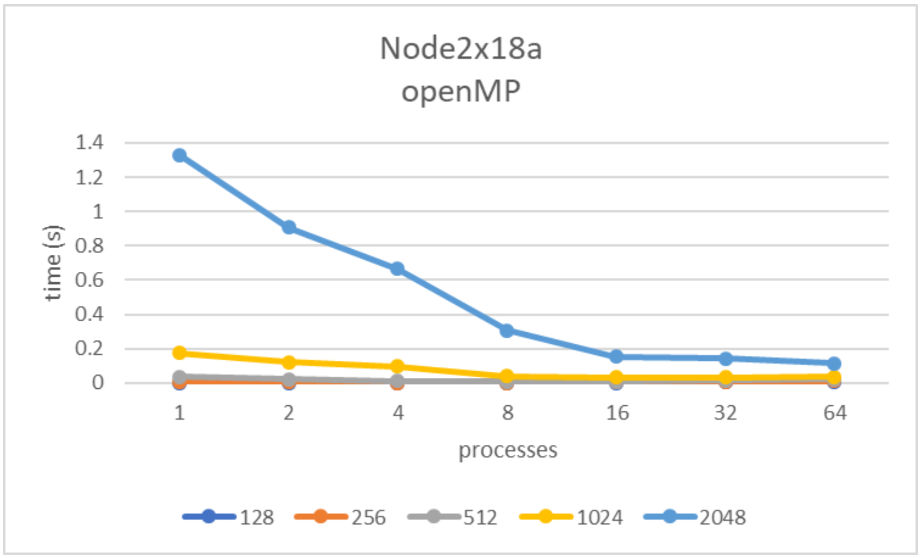
Additionally, most of the plots do not include the times for the 2048 sized matrix. This is because the times were usually large enough that it suppressed the smaller arrays in the plot (3 seconds compared to 0.003 seconds).

Below shows a comparison between the three types of parallelization on the "node1x18a" computer, for the case of matrix size 1024. OMP was the best performing at all points, while pthread became worse as more threads were created.

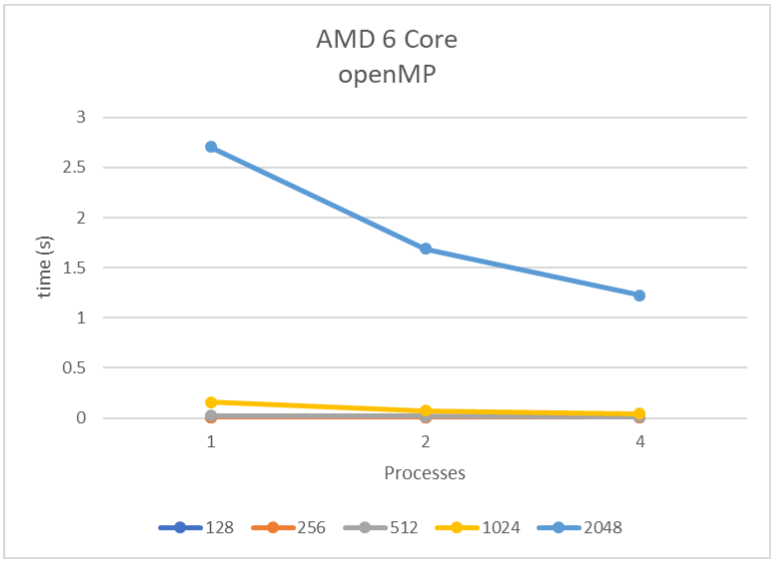


MPI on the other hand, shows inconsistent behavior. Slowing down and speeding up again. Is this due to network traffic? Perhaps unexpected load on a particular node?

OMP was by far the best in every metric. It was the easiest to implement as well as the returned the best results. The obvious drawback of OMP compared to MPI is that it does not allow for multiple nodes, so all calculations are local to the processes.



OMP show consistently better performance at every matrix size with more processes. It does not have the same overhead associated with pthreads. The performance is evident even on smaller systems as shown below.



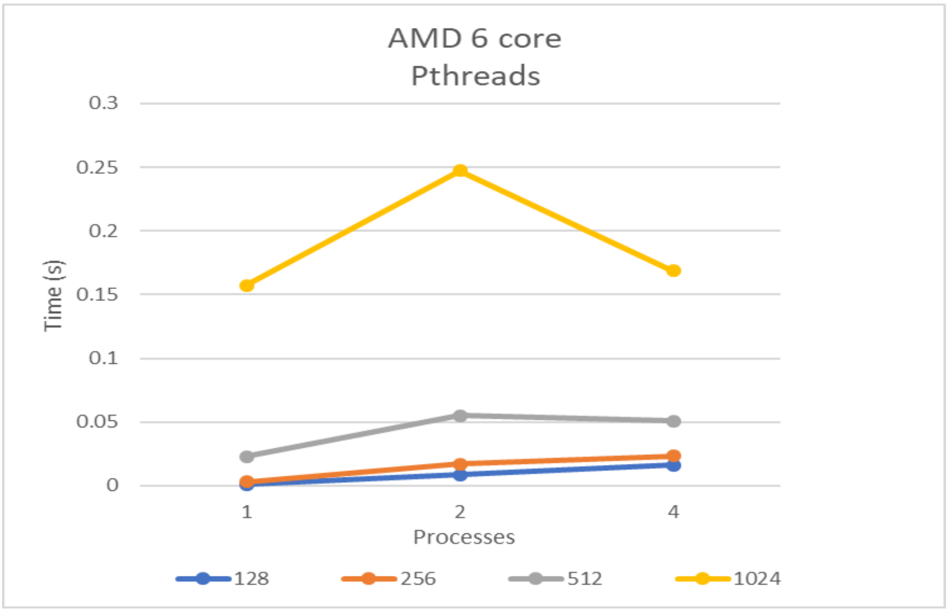
Discussion:

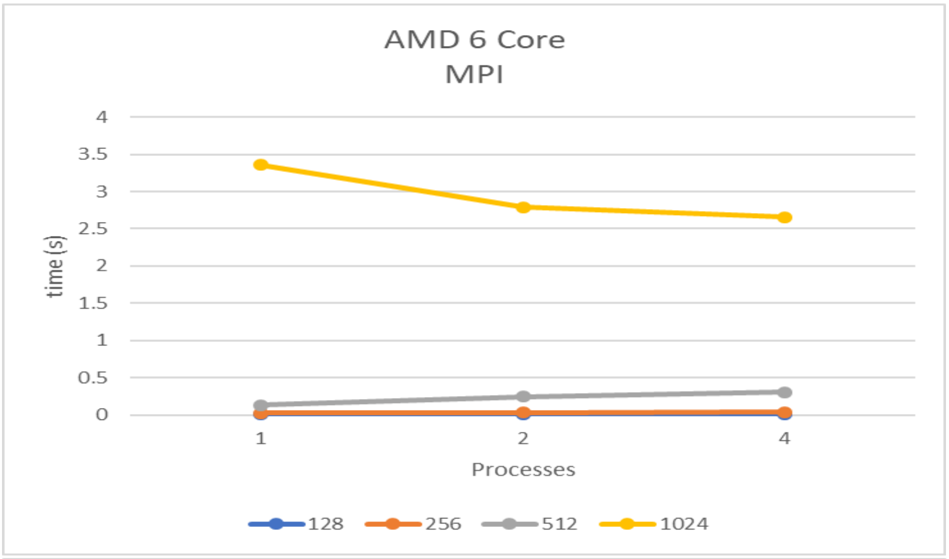
In terms of performance, OMP was best followed by Pthreads. MPI generally performed poorly. Without a better understanding of what's going on in my code, I can't be sure if MPI is just inherently slower (or had network issues), or if my code is flawed. It almost every case it was slower than a sequential solver on one core.

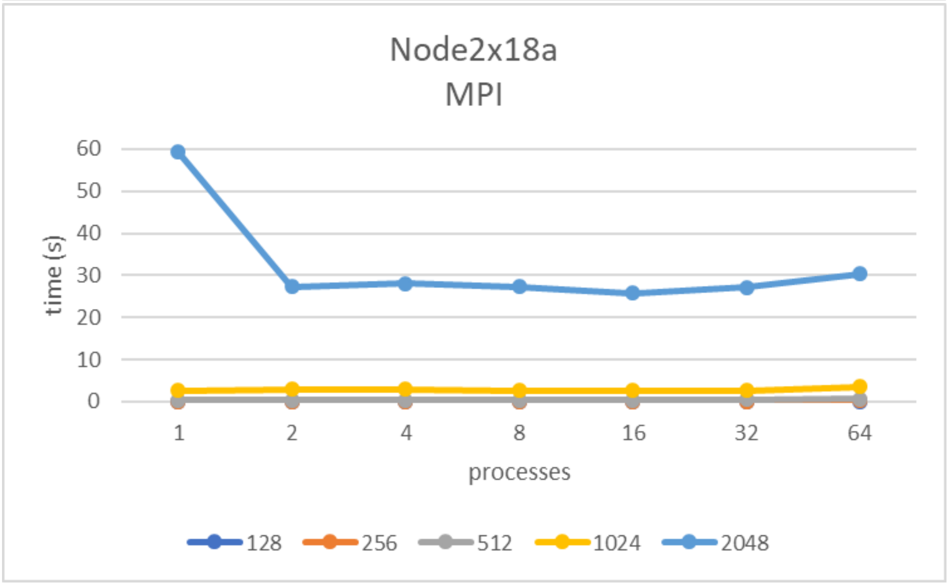
OMP was a surprise, as it was extremely easy to implement and had the best performance at every step. It also does not require any special steps or add on except for a compiler flag, just like pthread. Unless I was needing distributed processing on distributed nodes, I would use OMP. Unfortunately, real world parallelization solutions are often so large that distributed nodes are necessary. This means that MPI is most often what is needed.

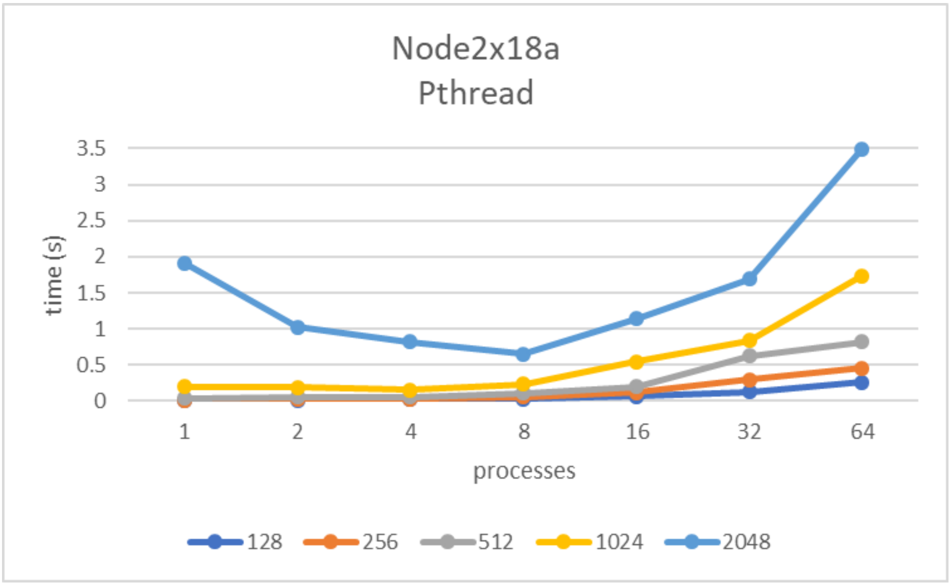
I would like to compare my times (and code) to other students who utilized Send/Recv commands instead of Scatter/Gather. This would go a long way to confirming whether my implementation is the problem.

Additional Plots




AMD 6 Core
MPI
time (s)
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
4
Processes
128
256
512
1024


Node2x18a
MPI
time (s)
60
50
40
30
20
10
0
1
2
4
8
16
32
64
processes
128
256
512
1024
2048


Node2x18a
Pthread
time (s)
3.5
3
2.5
2
1.5
1
0.5
0
1
2
4
8
16
32
64
processes
128
256
512
1024
2048