Introduction:

For this project we are tasked with implementing parallelism using p-threads, MPI and openMP. All of these methods have slightly different implementations but overall have the same functionality of dividing a large workload across multiple threads of execution. The general trend for this project is as more threads are added the execution time per thread and the total execution time should decrease as more threads are added. This also comes with diminishing returns as it takes time to open each thread.

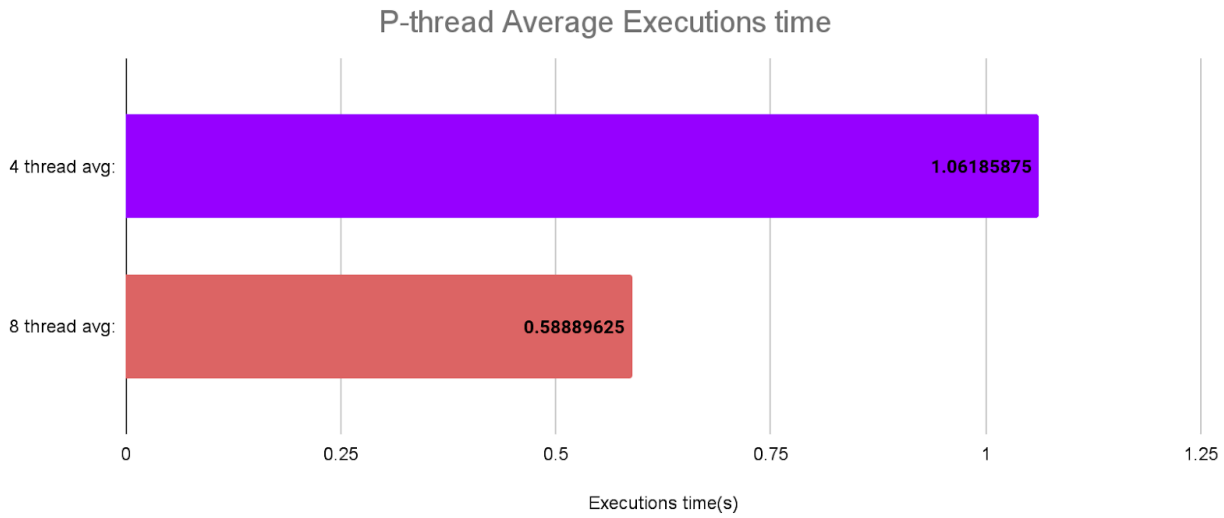
Implementation:

PThreads:

4 Threads:
Thread: 0, Time: 1.062770
Thread: 1, Time: 1.060879
Thread: 2, Time: 1.061981
Thread: 3, Time: 1.061805
8gb per node

8 Threads:
Thread: 0, Time: 0.589675
Thread: 1, Time: 0.584328
Thread: 2, Time: 0.589301
Thread: 3, Time: 0.586483
Thread: 4, Time: 0.591049
Thread: 5, Time: 0.586999
Thread: 6, Time: 0.591674
Thread: 7, Time: 0.591661
8gb per node

Thread Number:	4 Executions time(s)	8 Execution time(s)
0	1.062770	.589675
1	1.060879	.584328
2	1.061981	.589301
3	1.061805	.586483
4		.591049
5		.586999
6		.591674
7		.591661



Analysis:

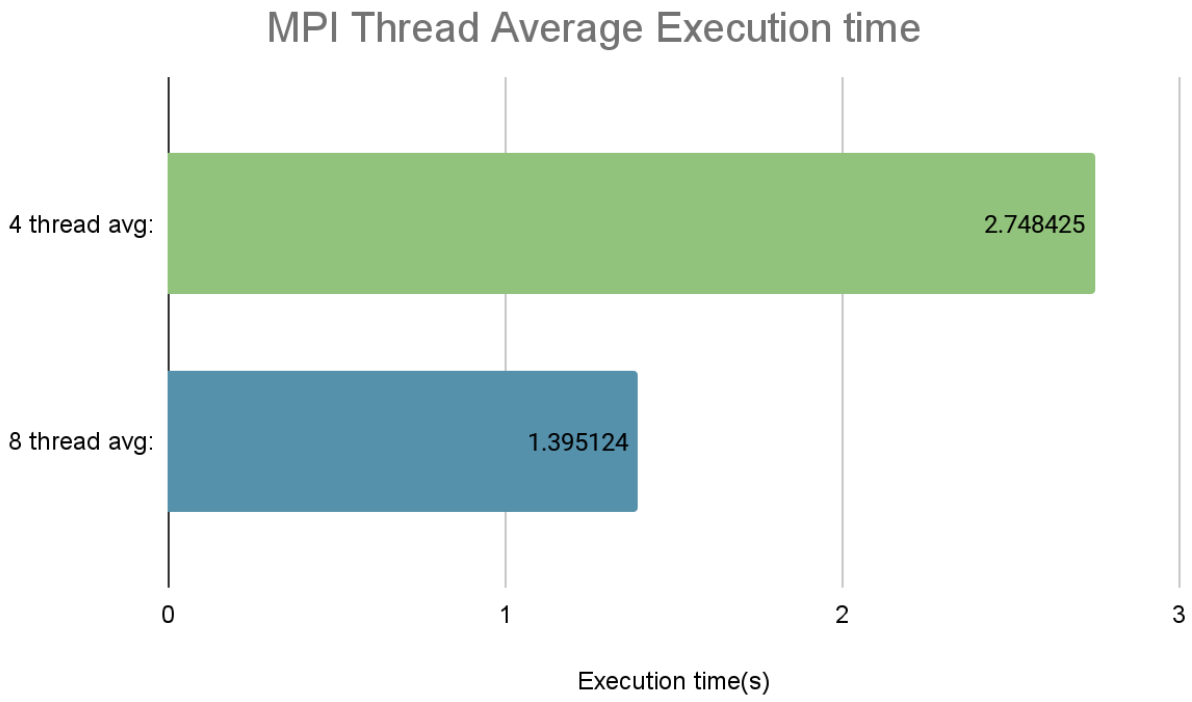
We had many different iterations for pthreads. This was the most difficult part about the project as we had to come up with strategies for parallelism on our own. Even though we understand the idea that “create” makes a new thread and “join” brings waits till the threads come back to continue, we struggled actually implementing it. In the end, we got the result we were looking for. By increasing the number of threads, we got a faster execution time.

MPI:

4 Threads:
Rank 2 processing time: 2.741170 seconds
Rank 0 processing time: 2.740158 seconds
Rank 1 processing time: 2.741407 seconds
Rank 3 processing time: 2.770966 seconds
2gb per node

8 Threads:
Rank 3 processing time: 1.376495 seconds
Rank 7 processing time: 1.376892 seconds
Rank 0 processing time: 1.380738 seconds
Rank 6 processing time: 1.384257 seconds
Rank 1 processing time: 1.382309 seconds
Rank 4 processing time: 1.384641 seconds
Rank 2 processing time: 1.430015 seconds
Rank 5 processing time: 1.445647 seconds
4gb per node

Thread Number:	4 Executions time(s)	8 Execution time(s)
0	2.740158	1.380738
1	2.741407	1.382309
2	2.74117	1.430015
3	2.770966	1.376495
4		1.384641
5		1.445647
6		1.384257
7		1.376892



Analysis:

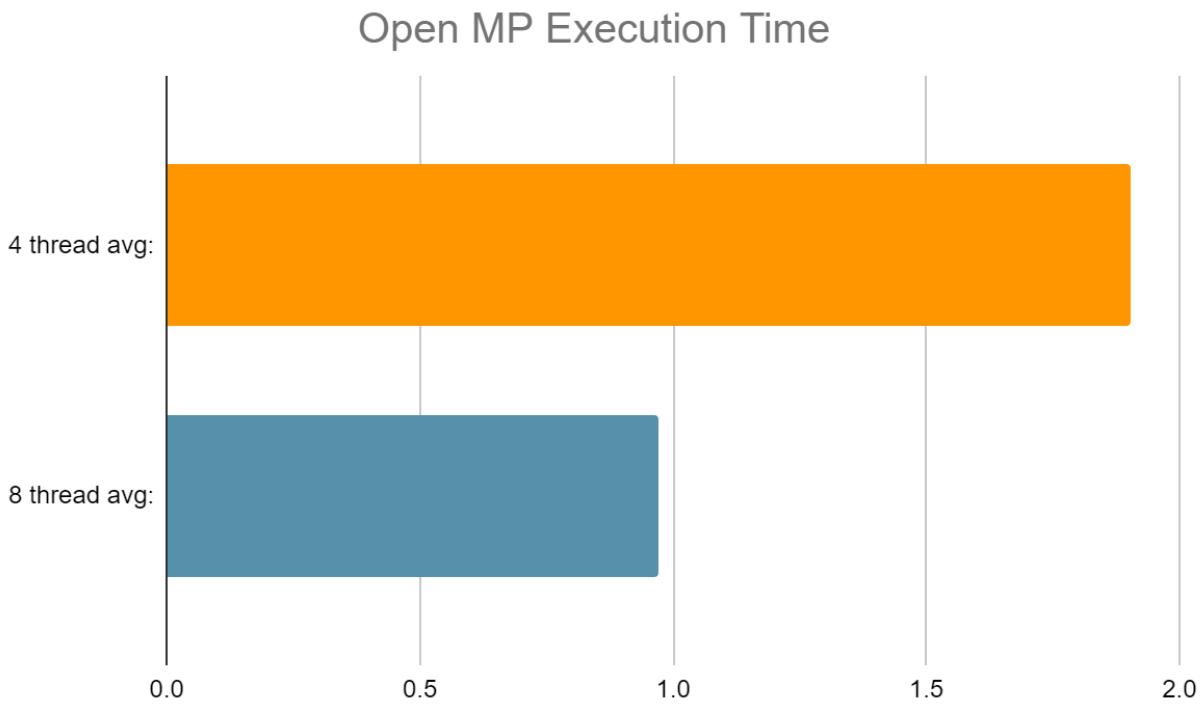
This raw data decreases as we add threads. This is the type of idea that we want when using parallelism. We can also see that it's about half the time per thread as it's double the thread numbers. This means that more threads equate to faster runtime. If we were running a bigger experiment then we could keep adding threads to see how many threads help us versus hurting us in the time that it takes to open all them.

OPEN-MP:

4 Threads:
Thread: 1 time: 1.905675
Thread: 3 time: 1.905674
Thread: 0 time: 1.905675
Thread: 2 time: 1.905674
2gb per node

8 Threads:
Thread: 2 time: 0.971157
Thread: 3 time: 0.971158
Thread: 7 time: 0.971158
Thread: 0 time: 0.971158
Thread: 6 time: 0.971158
Thread: 1 time: 0.971158
Thread: 5 time: 0.971158
Thread: 4 time: 0.971158
4gb per node

Thread Number:	4 Executions time(s)	8 Execution time(s)
0	1.905675	0.971158
1	1.905675	0.971158
2	1.905674	0.971157
3	1.905674	0.971158
4		0.971158
5		0.971158
6		0.971158
7		0.971158



Analysis:

This runs way better than both pthreads and mpi. This was also way easier to implement and our code was shorter than the rest as well. Overall, open-mp is our personal favorite. One nice part that we figured out about open-mp is the fact that we don't have to go into our script and initialize an amount of threads. Instead when we run it with beocat it utilizes what it's given.

Conclusion:

Overall this project was much more involved then the previous ones. This is because the threaded and parallelism concept is more involved than most of the concepts in the class and the implementation of these concepts is also difficult. The pthreads implementation was definitely the hardest to implement. MPI was much easier to implement as it handles some of the parallelism for the user. This is nice but also comes at the cost of more overhead and not having as much control as the other 2 options. OpenMP seems to be the best of both worlds with a simpler implementation than pthreads but more control than the MPI. The ease of switching back from one multithreaded task back to a threaded task in OpenMP is also very convenient.