The link to the github repository is `https://github.com/yajurahuja/High-Performance-Computing`

3.1 Pitch your final project

Project proposal sent and approved.

3.2 Approximating Special Functions Using Taylor Series Vectorization

Used the snappy server on cims as it was Intel architecture.
I have improved accuracy of sin function upto 12 digits. The code is in the **fast-sin.cpp**. I have completed both **sin4_intrin()** and **sin4_vector()** functions and have observed quite a speed up in both.

```
[[ya2109@snappy1 HPCSpring2022]$ cd homework3
[[ya2109@snappy1 homework3]$ make clean
rm -f omp-scan fast-sin fast-sin_ *~
[[ya2109@snappy1 homework3]$ make
g++ -std=c++11 -O3 -march=native -fopenmp -fno-tree-vectorize omp-scan.cpp  -o omp-scan
g++ -std=c++11 -O3 -march=native -fopenmp -fno-tree-vectorize fast-sin.cpp  -o fast-sin
g++ -std=c++11 -O3 -march=native -fopenmp -fno-tree-vectorize fast-sin_.cpp  -o fast-sin_
[[ya2109@snappy1 homework3]$ ./fast-sin
Reference time: 17.2122
Taylor time:    3.3068      Error: 6.928125e-12
Intrin time:    0.9041      Error: 6.928125e-12
Vector time:    0.9019      Error: 6.928125e-12
[[ya2109@snappy1 homework3]$ ./fast-sin_
```

Figure 3.2.1: Fast-Sin

After loading the gcc-9.2 module, I observed

```
[[ya2109@snappy1 homework3]$ module load gcc-9.2
[[ya2109@snappy1 homework3]$ ./fast-sin
^C
[[ya2109@snappy1 homework3]$ make
make: Nothing to be done for `all'.
[[ya2109@snappy1 homework3]$ make clean
rm -f omp-scan fast-sin fast-sin_ *~
[[ya2109@snappy1 homework3]$ make
g++ -std=c++11 -O3 -march=native -fopenmp -fno-tree-vectorize omp-scan.cpp  -o omp-scan
g++ -std=c++11 -O3 -march=native -fopenmp -fno-tree-vectorize fast-sin.cpp  -o fast-sin
g++ -std=c++11 -O3 -march=native -fopenmp -fno-tree-vectorize fast-sin_.cpp  -o fast-sin_
[[ya2109@snappy1 homework3]$ ./fast-sin
Reference time: 0.3623
Taylor time:    3.2471      Error: 6.928125e-12
Intrin time:    0.0026      Error: 6.928125e-12
Vector time:    0.0026      Error: 6.928125e-12
[ya2109@snappy1 homework3]$
```

Figure 3.2.2: Fast-Sin

We see that the accuracy is upto 12 digits and there is a good speed up.

3.2.1 Extra Credit

The idea I used to solve for solving this problem was to get range in the input angle in the range $[-pi/4, pi/4]$. But that range is not enough. So also calculate Cos in the range$[-pi/4, pi/4]$. As $\sin(\theta) = \cos(pi/2 - \theta)$, we essentially get the range $[-pi/2, pi/2]$ for sin. Then use the vectorized already defined sin and cos functions to get the result. The rest of the angles can be reduced to this range. Which is what the function does. The steps I have take are below.

1. First get the angle in the $[0, 2pi]$ range using the modulo function. This also handles the negative cases.
2. Angle in the range$[\frac{3}{2}pi, 2pi]$ has same sin value as $[-pi/2, 0]$.
3. The values from $[\frac{1}{2}pi, \frac{3}{2}pi]$ are mirror images of values of sine in range $[\frac{-1}{2}pi, \frac{1}{2}pi]$.

Now any of the values in the reduced range are either $[-pi/4, pi/4]$ for either cos or sin. So this way we are able to calculate any range. I have implemeted this **fast-sin_.cpp** file which I have created. It also gives a 12 digit accuracy. The intrin version of the function is also there in the same file. There is some bottleneck in allocating and deallocating memory within the function but the sin and the cos function for small range are really fast.

3.3 Parallel Scan in OpenMP

I have replaced N by $N + 1$ because the question starts states to calculate sums from $1 \ldots N$ and the first index is 0. I created the parallel version of scan where I diveded the calculation of parallel sums into tasks where each thread is assigned one task. And one thread is assigned to that task. I used the single and task constructs in OpenMP to do that.

System used: linserv1.cims.nyu.edu
Architecture: AMD Opteron 6272 (2.1 GHz) (64 cores)

Below is that table summarizing the time taken.

Number of Threads	Seq Scan(Sec)	Parallel Scan(Sec)
1	3.58	3.30
2	3.63	1.73
4	3.64	1.01
6	3.5	0.62
8	3.05	0.46

After increasing to higher number of threads than 8, improvement observed was not that significant.