Khoa Nguyen - Lab 6 Notes - 03/26/2022

1. Analysis on mmm-naive
   I ran the mmm-naive with the default arguments a couple times. On average, the running time to allocate memory for the matrices is 0 sec; the running time to initialize the values of the matrices is 0.025 sec; and the running time to calculate the matrix multiplication is 10.6 sec (106 sec to do it repeatedly 10 times). As expected, the most expensive task in this code is the implementation of the matrix multiplication repeatedly. Thus, it will be efficient to parallelize only this part.

   One of the running results is shown below:

```
# itemsPerDimension: 1000, repeats: 10, platform: unknown,
coreSpeed: 1.000
# platform, totalBytes, matmulTimeForAll, matmulTimeForOne,
MatrixKBPerSecond, mallocTime, initTime
unknown, 4000000, 106.454, 10.645, 366.941, 0.000, 0.025
```

2. Design of parallel solution
   My design will only focus on parallelizing the matrix multiplication for one time, then run the parallelized matrix multiplication repeatedly in serial (similar to task parallelism). Because the resulting n by n matrix has n*n=h items/sum calculations, I will split these calculations among the threads. Imagine transform the n*n matrix into a list of h index, notice that h=i*n+j, where i is the row index and j is the column index. Therefore, instead of having two for loops with i and j (in the mmm-naive.c), I will derive i using h/n and derive j using h mod n. To split h items evenly, I will use a simple parallel for loop because all calculations will cost the same regardless of h.

3. Analysis on parallelized version
   I ran the code with the default arguments a couple times. On average, the running time to allocate memory for the matrices is 0 sec; the running time to initialize the values of the matrices is 0.025 sec; and the running time to calculate the matrix multiplication is 1.1 sec (11 sec to do it repeatedly 10 times). Again, the code spent most of the time of matrix multiplication implementation, but it has been improved significantly with a speed up of 10 using 16 threads.

4. Visualization
   Link:
   https://docs.google.com/spreadsheets/d/1Z_fElHmsz3Zg08XgxBEW1UFemSiuKQ4Hsl8a_VpPNTE/edit?usp=sharing