ECS 158 Project 1

Program 1:

For program 1, we implemented the multiply function within mmm_omp.c. The rest of the code (i.e. the main function) is nearly identical to that of mmm.c from Project 1. We parallelized the ikj loop using the OMP parallel for directive with static scheduling. We also tried dynamic scheduling, but this was marginally slower (1.173855 seconds vs 1.1593 seconds). The other clauses we did not try, as they all relate to variable sharing and our code does not write to any shared variables (aside from the C array, but even then, no two threads ever write to the same index). Following are the results of running the OMP implementation against Project 1's pthread:

order	50	200	500	1000	2000
pthread	0.0007	0.003	0.0196	0.1582	1.2594
omp	0.0032	0.0016	0.0185	0.1433	1.1622

Pthread only had the advantage for the smallest order, which suggests that OMP has higher overhead for thread creation but distributes work more efficiently. Perhaps the higher overhead comes from the fact that OMP must dynamically determine the number of threads whereas our pthread implementation is hard coded to use eight. Next, we ran the OMP implementation with various numbers of threads. The results were as follows:

threads	1	2	4	8	16	32
omp	5.3915	2.8776	1.4969	1.1758	1.5289	1.8335

This is not particularly surprising. The machine has eight cores so with fewer threads the hardware is not being fully utilized and with more OMP has to waste time context switching to different threads running on the same CPU.

Program 2:

For the serial portion of program 2, we implemented the room as a static array of two dynamic arrays, with flipping indices to refer to the current and next array in the run function. We traverse the arrays in standard ij order. For this part, we determined the average temperature statically, and therefore our initialization loops do not need to perform any arithmetic to calculate it. We used one loop for all of the walls because splitting the initialization into multiple "cache friendly" loop with contiguous memory did not seem to have a material impact on the initialization time.

For the parallel portion, we are instead computing the average dynamically in order to use a reduction clause (each thread computes its own total, all of which are added together then divided by the number of wall cells). The run function is largely the same as before, except for the parallelized outer loop with max reduction clause.

Program 2 Timing:

The timing results were as follows (horizontal axis is number of threads, vertical is input matrix order):

heat_distribution_serial

Epsilon 0.001

50	0.0083
200	0.4282
500	5.7497

Epsilon 0.01

50	0.0018
200	0.0981
500	0.6435

Epsilon 0.1

50	0.0006
200	0.0116
500	0.0621

Epsilon 100

50	5E-06
200	7E-05
500	0.0006

heat_distribution_omp

Epsilon 0.001

	1	2	4	8	16	32
50	0.0054	0.0046	0.003	0.0027	0.0361	0.0701
200	0.4288	0.2331	0.1268	0.1329	0.3308	0.5262
500	5.7871	2.9244	1.6739	1.5497	2.0854	2.5615
1000	28.499	20.107	19.36	19.672	19.519	19.954
2000	115.88	83.321	80.397	81.119	82.063	82.451

Epsilon 0.01

	1	2	4	8	16	32
50	0.0021	0.0019	0.0019	0.0016	0.0218	0.0379
200	0.0979	0.0503	0.053	0.0272	0.0776	0.1225
500	0.6084	0.3092	0.1919	0.1699	0.2202	0.2738
1000	2.9068	2.0082	1.9542	1.9718	1.9552	1.9842
2000	11.493	8.415	8.0516	8.0631	8.2474	8.3906

Epsilon 0.1

	1	2	4	8	16	32
50	0.0007	0.0005	0.0005	0.0004	0.007	0.014
200	0.0128	0.0051	0.0054	0.0027	0.0075	0.0131
500	0.0635	0.037	0.0321	0.0176	0.0229	0.0281
1000	0.2881	0.2022	0.1966	0.1958	0.1962	0.2012
2000	1.1557	0.8708	0.8238	0.8094	0.8279	0.829

Epsilon 100

	1	2	4	8	16	32
50	7E-06	5E-06	5E-06	5E-06	5E-05	9E-05
200	0.0001	5E-05	3E-05	3E-05	6E-05	5E-05
500	0.0006	0.0003	0.0002	0.0001	0.0001	0.0002
1000	0.0015	0.0011	0.001	0.001	0.001	0.001
2000	0.0064	0.0047	0.0044	0.0043	0.0044	0.0044

As anticipated in the assignment, the main gain does happen between one and two threads, after which the performance reaches a bit of a plateau and afterwards begins to decrease again. There are many plausible explanations for this. One likely issue is with the cache: perhaps the OMP scheduler is giving the threads chunks that are too small and therefore the program is unable to fully take advantage of spatial locality. Another explanation could be related to NUMA architecture, perhaps adding additional threads is involving cores that are relatively far from the memory, and because of the reduction clause and implicit barrier the entire process must wait on the slowest thread. The most likely culprit, however, is that the outermost

while loop cannot be fully parallelized (because subsequent iterations rely on previous ones) and therefore is must still run the same number of iterations serially however many threads there are.