Dept. of Computer Science, University of California, Davis
ECS 158 Megha Jain, Andy Wu October, 2018

Project #2

1 Implementation Overview

1.1 Phase 1

In this phase, we parallelized our serial code for matrix multiplication using OpenMP. The serial code we chose corresponded to the best performing permutation in P1, in our case i-k-j, due its leveraging of temporal locality. We used the directive - parallel for schedule(<params>). We tested our code with different parameters to the schedule directive (static, dynamic, guided). Further we also tested our code for different number of threads which was specified at run time. Following table(s) show the data we collected :

	ikj	pthread	OpenMP
50	0.00021267	0.000598	0.00313
200	0.00987166	0.0056086	0.003356
500	0.060559	0.021824	0.020065
1000	0.562276	0.1342	0.1282643
2000	4.68472467	1.1234546	1.0369173

Compared to the pthread version, the OpenMP certainly performs better, especially for higher N (> 200).This is because the overhead in the pthread implementation of creating and passing structs as arguments is now not required. As seen later, openMP also allows us to dynamically change scheduling parameters and the number of threads which in the case of pthread would require rewriting parts of the code for every set of different parameters. Thus, overall openMP is easier to use, is more flexible, dynmaic and customizable and performs better due to less overhead.

Num Threads	2000
1	4.746319
2	2.591482
4	1.345081
8	1.045834
16	1.495975
32	1.709762

As seen, even though the peak performance happens at 8 threads, the greatest performance leap occurs when going from 1 to 2 threads, This is because the time advantage gained by adding more threads after 2 is compensated/nullified by the overhead of managing, switching, and synchronizing the added threads. This is especially true since largest N allowed is 2000. If the higher matrix orders (10k+) were tested, then most likely greater number of threads would increase performance. The performance also goes down after $>$ 8 threads because the CSIF machines can execute at most 8 threads at once and adding more threads than that does not speed up the calculation but does add extra overhead of managing and switching the added threads.

Schedule	2000
Static	1.046877
Static(4)	1.04448
Dynamic	1.042853
Dynamic (4)	1.039684
Guided (4)	1.03472
Guided (8)	1.044264

We tested different scheduling parameters for N=2000. As seen from the table, the difference in the time measurements isn't very significant. That being said, dynamic seems to perform better than static as in case of dynamic, the next iteration is given to the first available thread. This prevents the entire computation by a thread from being throttled by some particularly slow iterations. Guided with chunk size of 4 performs the best. The higher chunk size of 8 for guided leads to poorer performance, as towards the end the chunk-sizes get smaller and uneven.

1.2 Phase 2

In this phase, we wrote serial code to compute the static heat distribution of a room. We followed the three steps as outlined in the assignment. First, to initialize the walls, we iterated once over the resolution of the room and for each iteration, we initialized the corresponding point on all four walls. This reduced the amount of work by initializing all four walls in one loop instead of using four loops and initializing each wall individually.

To initialize the interior, we first calculated the mean of all the edge values. We did this using two for loops resulting in all exterior points being traversed once only to avoid over counting. Then we initialized the interior points to this mean value in a nested loop.

The third part involved calculating the new temperatures in a loop until the difference between two iterations was below a threshold for all points. This part was tricky because we needed to store the old values that were being used to calculate new values, and also the new values. In order to do this, we used a three-dimensional array (allocated as a 1D array) of dimension $R \times R \times 2$, where R is the resolution. By switching between using the first and second $R \times R$ arrays, we could store all the values in one structure and avoid copying the data back and forth. In order to calculate the new values, we iterated over the old array and calculated the average of the surrounding points. In order to determine when to stop, we used a max variable. Since difference between iterations must be less than epsilon for *all* points, we kept track of the max difference. If the max is less than epsilon, then the difference at all points is below epsilon, so the iteration can stop. Otherwise, there is *at least* one point whose difference is greater than epsilon, so the loop must continue.

1.3 Phase 3

In this phase, we parallelized our serial code for the calculation of heat distribution. Specifically, we parallelised 4 for loops with common directive of parallel for and some other directives specifying scheduling policy. For two of the loops, we also used a reduction clause. The first was for calculating the max difference value between previous and current iteration. Since the calculation of h value for each $h_{i,j}$ was done in a nested loop which was parallelised using openMP, the reduction clause collapsed the results of each thread so that the final max value would represent the max value for all the threads. Similarly, we use a second reduction clause of sum whilst calculating the mean of all the edge values. This also combined the local/individual results from each thread into one final value.

We used static scheduling parameters for cases where temporal locality could possibly be used to our advantage (i.e in the edge initialization and mean calculation loops). For the main hmap loop, we used the dynamic scheduling option since that proved to be the fastest.

We tested our code for both serial and omp versions, for 4 various inputs values for N, fire temp, wall temp, and epsilon as seen in the following tables (serial table followed by omp):

	Initialise	Hmap
100,100, 0, 0.1	6.20E-05	0.0102213
500,100,20,0.01	0.000902	0.5797
1000, 75, 30, 0.01	0.004796	1.442595
2000, 82,16, 0.015	0.011504	5.67665

	Initialise	Hmap
100,100, 0, 0.1	2.37E-03	0.005585
500,100,20,0.01	0.002523	0.1734093
1000, 75, 30, 0.01	0.001225	0.935885
2000, 82,16, 0.015	0.00534	3.60775

The number of iterations required to converge (and thus time) is directly proportional to the difference between fire temp and wall temp and is inversely proportional to the value of epsilon.

For the initialization part, we see that that the parallelized code actually performs slower for $N < 1000$. In these cases, it would actually be better to leave the initialization loops to be performed by one thread only. The performance gain from parallelizing these loops only occurs when N is considerably large.

For the hmap calculation part, as seen, the omp version certainly has a significant performance advantage over the serial version, most notably for the last test case where N=2000.

	Initialise	Hmap
2 Threads	0.008626	3.725454
4 Threads	0.005944	3.58648
8 Threads	0.005957	3.609938

Further, we also tested our code for the last test case but with varying number of threads - 2, 4, and 8. As expected, the greatest leap in performance occurs on addition of two threads. After this, it narrowly improves for 4 and decays for 8 (similar to results from

phase 1). This is because, as the the number of threads increase, the overhead of managing, switching and synchronizing them overtakes the performance gain generated by using these extra threads.

Overall, we concluded that openMP was definitely easier to use and synchronize with respected to shared memory than pthread. It was also more dynamic and customizable. And lastly, as expected, when the input size was large it was significantly faster than pthread as it had less overhead from not having to create, manage and pass argument structs.