Program #1: Matrix-multiplication improved

3.2 In p1, we found that ikj order was the fastest to compute because of how ikj accessed our cache lines. Therefore, we copied our code for our ikj function, as well as our printMats() function and our timeittook() function. Our old code also had error checking. Now, we changed the arguments so that it only takes one argument for the matrix order. 3.3 We included omp.h in our mmm_omp.c file and made the necessary changes in our Makefile to run mmm_omp.c. We added -fopenmp to use the openMP library. We added #pragma omp parallel and #pragma omp for before the for loop calculating the matrix multiplication. 3.4

Pthread

50	200	500	1000	2000
0.251289	.60225	.26800	.16754	1.66725
0.30052	.54047	.29488	.18369	1.58212
0.20059	.52864	.25212	.17868	1.57365

OpenMP

50	200	500	1000	2000
0.5194	.66645	.2563	.1574	1.2247
0.5419	.5864	.2643	.1556	1.223
0.4195	.4855	.2462	.1597	1.2239

Running the OpenMP version for matrix orders of 50,200,500,1000,2000, we found that the OpenMP runs almost double when N is low (i.e. N = 50). However, as N increases, the advantages of OpenMP is more obvious. We start to see that OpenMP is faster at N=500, with an average of .272 seconds for pthreads and .2556 seconds for OpenMP. You can definitely see a difference at N=1000 and N=2000, with OpenMP being an average of .3837 seconds faster at N=2000. This is probably because when N=50, the matrix is small enough that OpenMP is not needed and Pthread optimizes the double for loops. Each thread executes each of their respective section of code independently. Tasks are divided among the threads so each thread can run their parts simultaneously. Thus, task parallelism and data parallelism are achieved with OpenMP. Advantages is that we will not need to create the threads like we have to do in pthreads. Thus, the code looks cleaner because we just call #pragma. The layout and decomposition is handled by OpenMP. Also, since it runs faster on larger N, openMP is more scalable in our case. However, OpenMP can be at a disadvantage because we do not have as much control as we do in pthread on which data each thread is assigned to. There is a limit to what we can do to OpenMP when pthread is more flexible.

Setting Environment variable

Thread =1	Thread =2	Thread =4	Thread =8	Thread =16	Thread =32
5.203	2.928	1.410	1.239	1.525	1.712
5.210	2.871	1.399	1.226	1.584	1.720
5.193	2.957	1.395	1.235	1.483	1.726

Changing the number of OMP threads by setting the environment variable, the results for the performances are as expected for the most part... except when we use too many threads. It turns out that using 8 threads on a matrix size of 2000 is the most optimal using OMP threads. When we use 16 and 32 threads, the performance time starts to increase a little bit, as using too many threads can actually hurt the performance time. This could be due to This could be due to false sharing, where there are many, many threads accessing the data that is adjacent in memory. Amdahl's Law explains why the speedup gets less and less, until it gets worse. Amdahl's Law explains that speedup is drastic with the addition of more cores when there are less cores, and then the greater the number of processors, the less and less it will speed up. We did some research and we found that thread creation takes up a lot of time. So, the time to finish the matrix multiplication for the 8 threads is probably less than it takes to create 32 threads and then going through the matrix multiplication, even though it might be faster to compute the result. Ultimately, not as many thread as possible is the fastest, and you will have to test to find the "sweet spot".

Dynamic, n/8	Dynamic, n/32	Static, n/8	Static, n/32	auto
1.2387	1.3343	1.2218	1.2652	1.5438
1.2209	1.2728	1.2213	1.2533	1.3264
1.2209	1.3264	1.2369	1.2552	1.3358
Avg: 1.2268	Avg: 1.3112	Avg: 1.2267	Avg: 1.2579	Avg: 1.402

Schedule: Schedule is helpful for controlling how threads iterate through the loops. Using static scheduling and dynamic scheduling, we have to be careful about the scheduling parameters. In dynamic scheduling, the work will occur at runtime which involves more overhead. Dynamic is useful when each computation is different depending on which subset of the matrix the thread is working on. On the other hand, static is more useful if each subset is around the same computation time, as static scheduling divides a loop into a specific number of subsets. In our case, static and dynamic scheduling do not differ much. Private: We found that privatizing data does not really help our case in our matrix multiplication. It would not be appropriate to initialize matA, matB, matC, or n with private(). We tried using private(r), where r stores the temporary sum that is used in the for loops while doing the computations, but this does not lead to better performance as it will be the same if we initialized r before we start the timer. Firstprivate() is not appropriate for our case either because there is no values that we need to initialize right before our for loop -- matA, matB, and n are computed and given, respectively, beforehand. Lastprivate() would not be appropriate either, as it should be used if we need a variable to get the last value of the loop. This applies to changing the default as well. Privatizing these variables is not needed in our computation. Barrier: We tried barrier, and we found that this is not appropriate with what we are trying to do, as it does not

depend on the computation before. MatA and matB have the values we need and we want all the threads to compute simultaneously and then join() the results. Our program does not require threads to wait for another before proceeding further. Critical: We found that the critical directive is not needed, because there isn't really a race condition if we divide the loop with a for directive. Since one thread only operates on the calculation at once, defeating what we want it to do, which is compute the matrix simultaneously. Section: We tried using sections but we found that this is not useful in our matrix multiplication. The threads should work at their respective sections, and no section can just be executed by any thread available, because they are assigned to their respective subsets of the matrix to start with, and they need to be done so in order to get compute the resulting matrix correctly. Single should not be used because, just like critical, we want all the threads to be working simultaneously and have them join. There is no section of the code that requires the execution of only one thread. The logic for single goes for the same for master. Task: We spent the most time working with the task directive. In task, work units are pushed to a queue and popped when there are available threads to be executed. We tried to put the task directive before the calculations are assigned to matC, and putting the single directive right before it is assigned to matC, and we did not find a increase in performance here. We did the same for when we get the corresponding element in matA before calculating matB, but again, to no avail. We tried putting task right before inner for loops but this averaged around 14 seconds, compared to 1.2267 seconds which is the fastest we found. We implemented the task directive right before the inner for loop and it seems to have increased the performance a little, but it may be small enough to be trivial.

Program #2: Heat distribution

4.1 In Program 2, we first copied over what we had for error checking for the last program as a template, changed each error checking conditions accordingly. For example, if the map order is below 0 or above 2000, if the temperature is below or above 0 or 100 respectively, or if the epsilon is out of bounds. We first initialized array h and array g, representing the old heat and the new heat points respectively. H and g are going to be representing h and g matrices, and each point will be accessed by i*row+j, similar to how we did it for the first program. Thus, we malloc() 2000*2000+2000 so it will be enough space if the map order is 2000.

4.2 and 4.3

Now, we will initialize array h, which is going to have the initial temperatures. We use double for loops to access columns and rows of the matrix. The entire first row is going to have the temperature of the heat, so if it is looking at the first row (i.e. i == 0), then we assign the value that user has inputted as the value of the heat map. We do the same for the last row which has the edge temperatures as well as the first and last column that have the edge temperatures (i.e. j == 0 and j == n-1). We also accounted for the corner cases where the two corners on the north wall should be the values of the heat wall. Now, for the rest of the elements it will be interior values. Before the for loops, we calculated the temperature that the interior points should be assigned to, by summing each temperature of the perimeter of the matrix which has all the initialized temperatures and then

dividing it by the size of the perimeter. We do this by multiplying the temperature of the heat wall by n and then the remaining walls by the perimeter-n. We calculated the perimeter by (n-1)*4. (n-1 because we won't double count the corners). Now that we have initialized the temperatures of h, we will now utilize g in order to compare the differences between each element of the newly computed values with elements of h. We created double for loops again to access the elements of h and g. At each iteration, I will be adding each element that is left of the currently element (h[(i-1) * matrix_order + j]), right (h[(i+1) * matrix_order + j]), above (h[i * matrix_order + (j-1)]), and below (h[i * matrix_order + (j+1)]). Then, we divide it by 4 in order to find the mean of the surrounding temperatures. We assign this value to g, and now g has the newly computed average temperature from h. Now, we created two double for loops to compare each element of h and g. In order to compare the difference, we used the fabs() function to find the absolute value and subtracted g at that element (by using g[i*matrix_order+j) from h at that element (by using h[i*matrix_order+j). We kept a max variable and compared to find the MAXIMUM difference between the iterations. If this difference is greater than max, then this will be new max. Lastly, we assigned the element in g to element in h so now h will be have the newly computed values, ready for the next iteration. In order to find an iteration that has the maximum difference between two points under the given epsilon, we put a while loop over our two double for loops. So, while max, which is the variable that is getting the maximum difference between two iterations, is still bigger than the epsilon, then we will keep computing for a matrix that isn't. Later, we added the condition "max == 0" as well so it can get into the while loop the first iteration. Then, it will start comparing. Since we only print out the results at each power of two until the last iteration, we kept a variable called powerOfTwo and called printf() only when i was at that iteration. Lastly, we added the clock_gettime() to time our initialization as well as the computation, and added necessarily print statements to follow Joel's format.

4.4

From the serial code we added some omp clauses. Using omp is not as straightforward as program 1. I tried multiple combinations. I added a collapse clause for one the for loops and ran 500 100 0 0.01. For some reason it does speed up the computation power. For experimentation, I parallelized every for loop and that did not produce the correct hmap.

OMP

epsilon:	**0.1**	**0.01**	**0.001**	**0.0001**
Hmap before:	0.730	0.148	0.3197	0.2817
Hmap after:	0.845	0.837	7.76	26.24

Serial

epsilon:	**0.1**	**0.01**	**0.001**	**0.0001**
Hmap before:	0.122	0.109	0.197	0.555
Hmap after:	0.8731	0.8624	9.31	27.24

I ran a couple of rounds with the matrix order being 500. I added a reduction clause when computing the the new heat map out of the old one and the timing only improved by a second. I think it is a bit harder to parallelize this code because of how we implemented what is inside the for loop. Our last loop that tells us if our max variable is greater than the current difference between two successive points has an if statement and parallelizing that produced a different hmap. Since the iteration is not producing the same result as the other iterations, parralelzing it would not be effective.