Part 1: Parallel Matrix-Matrix Multiplication

Parallelism on GEMM:

When parallelizing the code with OpenMP, the first thing I had to decide is which loop to parallelize. The optimized code was written in i,l,j format.

Parallelize the l loop

It is not feasible to parallelize the l loop due to dependencies in calculating each element in the matrix C. Each value of l corresponds to a combination of elements from matrices A and B that contribute to the same element in C[i][j]. Thus, multiple threads would attempt to update C[i][j] simultaneously, leading to race conditions which can result in incorrect results.

Parallelize the j loop

Square Matrix: Within this parallelization, since threads are working on the same row(i) of the resultant matrix, they may simultaneously access parts of the row that are stored in the same cache lines. This can lead to cache conflicts and false sharing. Additionally, multiple threads would need to access the same elements of matrix A, requiring significant synchronization and communication between threads for resource sharing. This added overhead further increases execution time.

Tall/Skinny: In this type of matrix, the n value is small, so the j loop has very few iterations. Thus, if I chose to parallelize the j loop, there would be an underutilization of threads as there is not enough work to be distributed.

Short/Fat: In this matrix, the j loop has many iterations (n) making it ideal for as it will fully utilize all threads. In this parallelization, each thread needs to access the same row of A. Luckily, each row of A is small enough to fit in the cache. Thus, contention is reduced because the cached rows will not experience frequent evictions due to the small size of the cached data. Most of the data from matrix A can remain in the L1/L2 cache which is private for each thread as open mp runs each thread on different cores.

Parallelize the i and j loop

Parallelizing both the i and j loops means that each thread works on its own unique element C[i][j]. While this approach is more fine-grained, it introduces additional contention and cache coherence issues. Multiple threads may access or modify elements in nearby memory locations (e.g., C[i][j] and C[i][j+1]). When one thread modifies an element, any other thread with that element's cache line in its cache will have that line invalidated. If these threads need to access elements within the same cache line again, they must reload it, which increases execution time.

Parallelize the i loop

Tall/Skinny Matrix: In this matrix, matrix A and the resultant matrix C have a large number of rows compared to columns. Thus, parallelizing the i loop will provide enough workload for each thread to be fully utilized. Each thread will be responsible for calculating a row of matrix C. Since data in matrices is stored in row-major format, this approach reduces cache contention and coherence issues, as well as the number of cache evictions and cache thrashing. By assigning each thread a distinct row, threads can work independently on separate cache lines, improving spatial locality and making cache utilization more efficient. In addition, each thread will read the same B[l][j] element for the calculation of the different rows of C. However, since B is small, it can fit into the cache which will avoid any cache contention.

Square Matrix: This is the best method as it avoids the cache contention described in the other implementations.

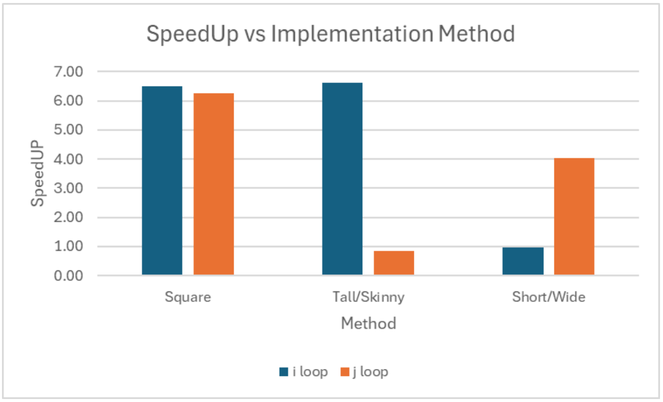


Figure 1 Speedup for Different Matrix Types

To test my reasoning above, I benchmarked the code with various loop parallelization options. I ran my code with 8 threads and left the scheduling to static scheduling with auto chunk size for now. However, I did not benchmark the i,j loop parallelization because implementing it would require changing my optimized i,l,j loop order to an i,j,l loop order in order to use the collapse command to parallelize the first two loops. The i,j,l loop order performs poorly in terms of cache spatial locality, making it not worth implementing. From the graph/results, it can be seen that my reasoning is correct. Note: Square Matrix (4096 by 4096), Tall/Skinny(4096 by 40), Short/Wide(40 by 4096)

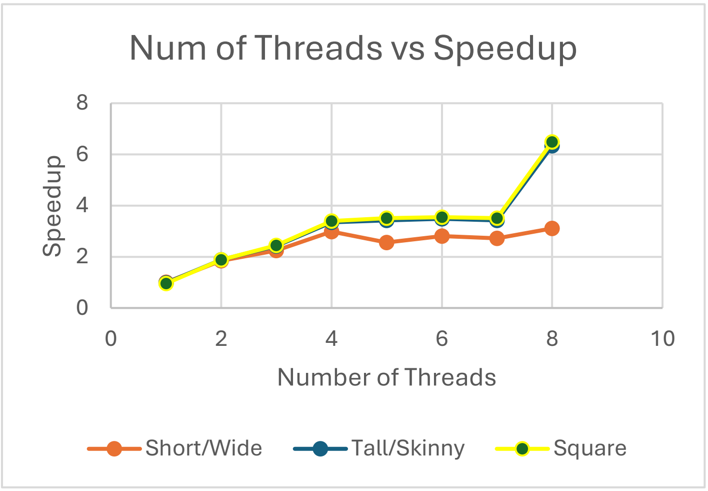
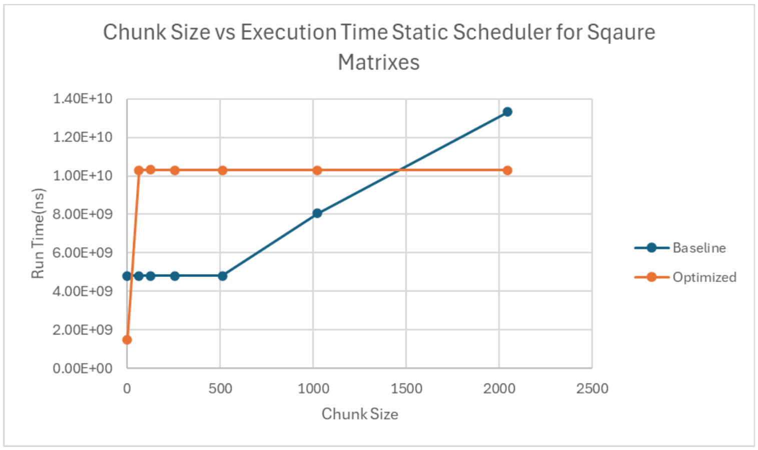


Figure 2 Num of Threads vs Speedup

The number of threads was the next parameter to be varied. The following results were calculated with static scheduler, auto chunk size. It can be seen that when the number of threads is 1 this is equivalent to just running the sequential code. We do not see a perfect positive slope because as the number of threads increase there become diminishing returns. This is due to the overhead of thread management (thread creation, scheduling, synchronizing etc.). In addition, more memory contention will occur. Although, there is a sharp rise in speed up from 7-8 threads in square + tall/skinny matrix, due to the memory accesses patterns resulted from the optimized tiled code.

The next parameter to test is the effect of the scheduler type and the chunk size. In the baseline gemm parallelized code, each task is decomposed so that it is responsible for a row of the resultant vector C. So the chunk size will dictate the number of rows that each thread is responsible for computing. Thus, with small chunk sizes, each thread is responsible for a small number of rows. The load will be balanced which is why we see a relatively constant run time. However, once we reach a point where the chunk size is greater than num tasks divided by the num of threads, there is not enough work for each thread. Thus, some threads may be left idle which results in the increasing execution time we see. This explains the shape of blue baseline curve below.



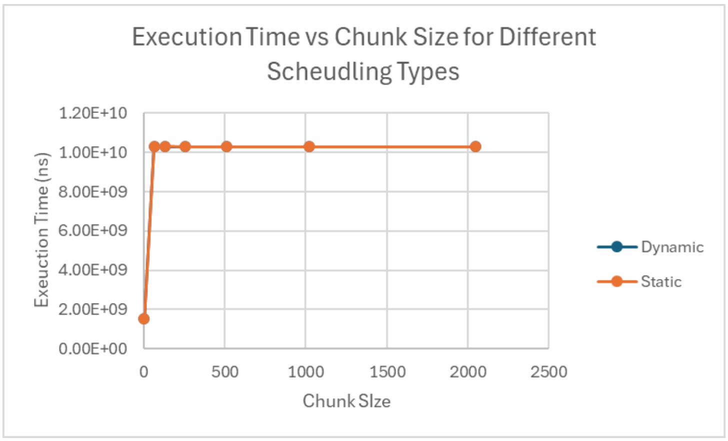
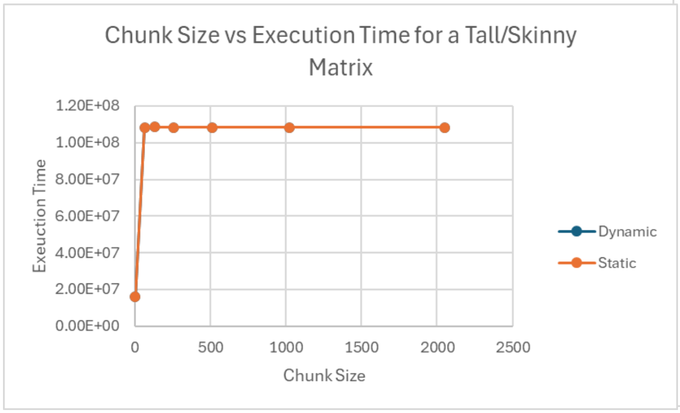


Figure 3 Chunk Size vs Execution Time on 4096 by 4096 matrix with 8 threads

For the optimized code (tiled + vectorized), the tiling is causing us to see a different pattern than baseline gemm. This is because the tiling causes the tasks to now be decomposed into each row of a tile of C. Thus, the workload for each task has now drastically decreased. The execution time is now constant for large chunk sizes as tiling results in each task being small enough to achieve good cache locality and that ensures the load is balanced even among large chunk sizes. There is an increase in execution time for small chunk sizes as smaller chunk sizes allow for more fine-grained load balancing.

From the graph above, the static and dynamic scheduling result in almost identical results for the square matrix. This is due to the nature of gemm. Since the workload is balanced for each task, using static vs dynamic makes no significant difference. There is a slight increase in run time for dynamic scheduling, this is due the scheduling overhead that is required to assign tasks to each thread at run time.

For tall/skinny matrices and short/wide matrices the same pattern is seen as the square matrix as each implementation is catered for the matrix type.



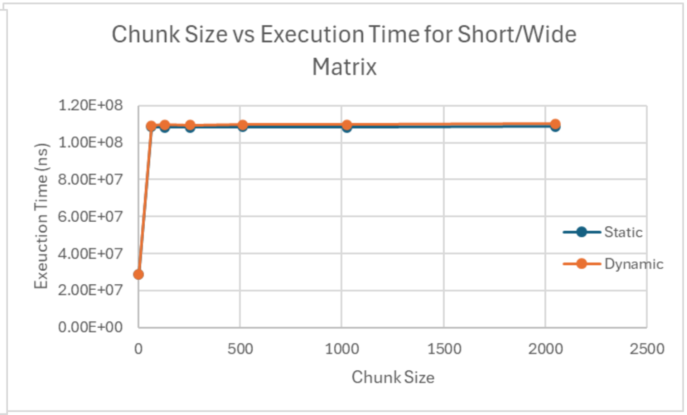
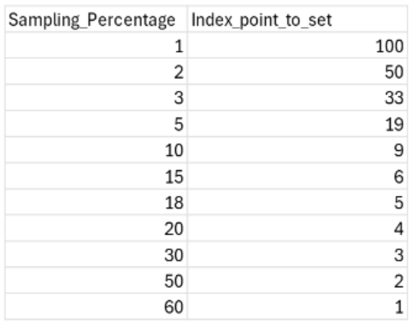


Figure 4 Chunk Size vs Execution Time for a Tall/Skinny and Short/Wide Matrix

Parallelism on SPMM:

Similar to GEMM, to perform parallelism on the optimized SPMM baseline. Parallel the i loop (outer loop/rows) and l loop (inner loop/columns on matrix B) is doable and can achieve certain speedup based on the input matrix size and scheduling strategy.



Sampling_Percentage	Index_point_to_set
1	100
2	50
3	33
5	19
10	9
15	6
18	5
20	4
30	3
50	2
60	1

Figure 5 Index points to set with the corresponding sampling percentage

<u>Dynamic VS Static Scheduling & Sparsity Effect:</u>

Recap the previous lab as we found this corresponding sparsity of the input matrix based on the sampling percentage. Perform the automatic static scheduling parallelism on each sampling percentage on i loops and get the result. And also, for each sampling percentage, I swap the chunk size from 1 to the 512, increment the power of 2 each time and record the best speedup for dynamic scheduling. The graph is obtained below.

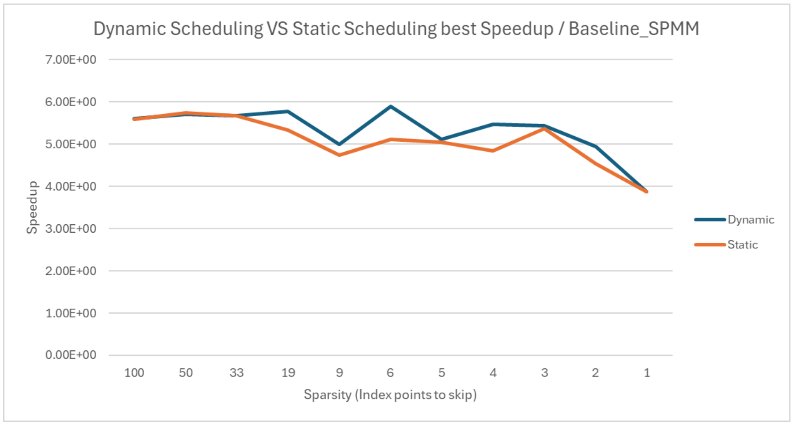
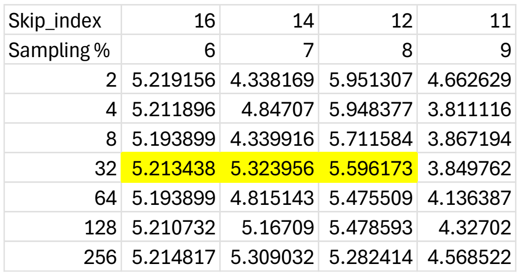


Figure 6 Dynamic Scheduling VS Static Scheduling best Speedup / Baseline_SPMM

From the graph, we can see that both dynamic and static scheduling parallelism all have a certain speedup compared to the baseline. In the range of sparsity from 19 to 1, the dynamic scheduling speed up performs relatively better than the static scheduling speedup. This is reasonable as due to the uncertainty of nonzero values in the sparse matrix, sometimes the pre-allocated rows can finish faster than the other rows, static scheduling will become idle once those rows finish and waiting for other threads to finish. Therefore, dynamic scheduling can be more ideal to handle this type of data structure as the worker pool model will always make show the threads will have certain tasks to execute. And our experiment will mainly focus on the sampling range from 6 to 9 as the speed up over the static scheduling is relatively larger and the sparsity level can represent a real-life sparse matrix.

<u>Square Matrix Parallelism</u>

For the square matrix parallelism, parallel the i loop (outer loos/rows) is chosen as this can best utilize each core memory space/cache without conflicting other data and also provide a relatively good spatial locality as each thread is accessed the matrix in the row wise order. And by performing experiments on the matrix size 4096*4096, the best chunk size when the sampling percentage between 6 to 9 is 32. The table on the right illustrates the simulation results on speedup improvements over baseline, as we can see the chunk size 32 weights the most across the sampling percentage from 6 to 8. It is also important to notice that due to the nature behaviour of the SPMM matrix, it is difficult to achieve an ideal load balancing on each thread, so it is hard to find one chunk size that can apply for different sparsity which is also why the chunk size of 32 is less performed when the sampling percentage is 9.



Skip_index	16	14	12	11
Sampling %	6	7	8	9
2	5.219156	4.338169	5.951307	4.662629
4	5.211896	4.84707	5.948377	3.811116
8	5.193899	4.339916	5.711584	3.867194
32	5.213438	5.323956	5.596173	3.849762
64	5.193899	4.815143	5.475509	4.136387
128	5.210732	5.16709	5.478593	4.32702
256	5.214817	5.309032	5.282414	4.568522

Figure 7 Chunk size analysis for 4096x4096 spmm within the sampling range 6-9

Tall Skinny Matrix Parallelism

Skip_index	16	14	12	11
Sampling %	6	7	8	9
2	4.035889	4.572199	4.157648	4.075065
4	5.788938	6.650955	5.653673	5.666777
8	5.837828	6.589723	5.743042	5.685153
32	5.823287	6.625381	5.756437	5.682935
64	5.866093	6.616604	5.749651	5.675568
128	5.916184	6.598371	6.357506	5.68529
256	5.917391	6.623369	6.354958	5.691103

Figure 8 Chunk size analysis for 4096x32 spmm within the sampling range 6-9

When the input matrix is tall and skinny (ratios of #rows to #cols are 100 and 0.01), parallel the i loop (outer loop/rows) is still the best strategy since the width of matrix became smaller, parallel the inner loops may introduce a lot of parallel overhead and lead to inefficient computations. Bue one thing I found is that for the sampling range 6 to 9, the best chunk size for the squared matrix may not be the best for the tall skinny matrix. By conducting experiments on matrix size 4096*32 spmm multiplication, the table on the right is obtained. It illustrates that different chunk sizes speedup over there sampling percentages, as we seen from the table, when the chunk size is 256, it provides the best speed up compared to the other chunk sizes. This is reasonable as the width of the matrix is small, a small chunk size may not best utilize the cache storage on each core so it may be ideal to increase the chunk size to let more data fit within the cache. In this specific experiment, I found that 256 could provide the best speedup across all the sampling percentages.

Short Wide Matrix Parallelism

From the gemm experiments, we found that for the short wide matrix multiplication, parallel the most inner loop(columns) could provide the best speedup. So, we first experimenting parallel the l loop in the spmm with static scheduling as it will help each thread divide the columns equally among each row. But after profiling, we found that this parallel strategy actually provides an average 0.33 speedup compared to baseline for the sampling percentage range 6 to 9. This shows that for spmm, even with short wide matrix multiplication, the inner loop parallelism may not be ideal as the data in each row is not contiguous in CSR data structure, so we decide to still parallel the i loop (most outer loop/rows).

Sampling Percentage	static speedup	dynamic speedup	Best Chunk size for dynamic
6	5.213036963	5.36	4
7	4.171637372	4.13	4
8	5.727642468	5.89	4
9	3.745003811	3.83	4

Figure 9 Static vs Dynamic Speedup over baseline

By comparing the static and dynamic scheduling, we found that the speedup over the baseline is nearly the same when the input matrix size is 32*4096. And the best chunk size for dynamic scheduling is also 4 for each percentage. This statistic shows that for short wide matrix spmm, if the sparsity of the matrix is distributed eqaully in each row, static scheduling should achieve the same performance as the dynamic scheduling. But when the sparsity is unknown, use static scheduling may not be ideal as some rows would finish early and we always want each thread not to go to sleep, so dynamic scheduling may still be the best for this case. And it is also important to notice that the ideal chunk size should be small enough to start so each thread can all get tasks to compute, it might be a good idea to set the chunk size equal to the number of rows/available threads to start.

Num of threads VS Speedup

To further explore the relationship of speedup with available threads, a single experiment is performed as shown on the right and it can verify that as the number of used threads increase, the speedup of the program also increases.

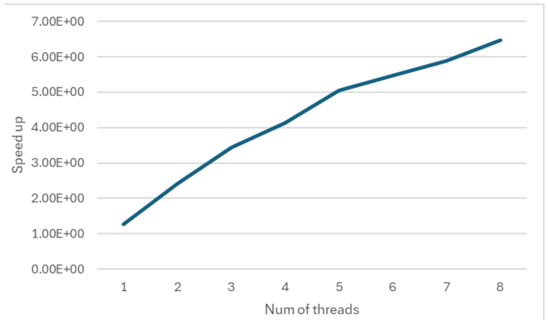


Figure 10 Num of threads VS Speedup

<u>Part 2: Integration Calculation</u>

For the integration function, where only certain OpenMP instructions are allowed, the pi integral value is computed as a sum of contributions from various x/step values. Each of these calculations is distributed among multiple threads, with the condition that no consecutive steps are assigned to the same thread. To achieve this, each thread processes every sp.NumThreads-th step starting from its thread Id. After each thread completes its calculations, the partial sum is stored in a shared vector, with the thread ID serving as the index. Once all threads have finished, a separate loop adds the values in the vector into the params->Sum variable. In terms of scheduling, I implemented manual static scheduling in the code. This ensures that iterations are explicitly assigned to threads based on the stride pattern, avoiding consecutive iterations being processed by the same thread. Using OpenMP's built-in scheduler would have resulted in incorrect results, as it would interfere with the custom stride logic, potentially mapping consecutive steps to the same thread.

For the integration function, where all OpenMP instructions were allowed, the loop iterations (from 0 to the total number of steps) were divided among the threads. Each thread was responsible for computing the partial sum of the integral for its assigned iterations. The reduction keyword in the #pragma directive was used so that each thread had a local copy of the sum variable. Once all threads completed their tasks, the local sums were reduced into a single global sum variable using the addition operation. This updated global sum was then used to calculate the value of pi.

In the code, each iteration performs the same amount of work (calculating the value of the function for the appropriate x and adding it to the sum variable). As a result, static scheduling is well-suited for this task. Static scheduling is beneficial because the workload is uniform, and it has lower overhead compared to dynamic scheduling. With static scheduling, iterations are assigned to threads before the loop starts, eliminating runtime task assignment overhead and improving performance. For simplicity, I used the default chunk size so that the iterations were divided evenly among the threads. This approach ensures a balanced workload and is efficient for this use case, where each iteration has the same computational cost.

The following results were collected from the numstep size of 10 ^ 8 and auto chunk size (if applicable) and with the number of threads set to 8.

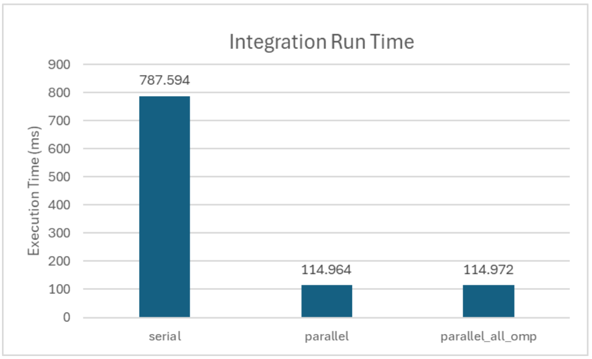
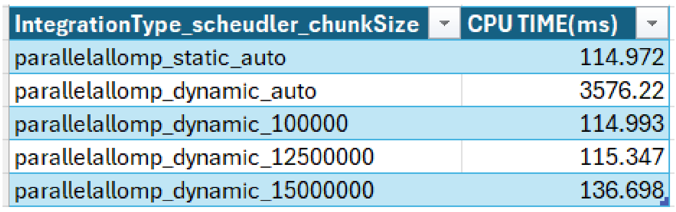


Figure 11 Run time of various PI implementations



IntegrationType_scheudler_chunkSize	CPU TIME(ms)
parallelallomp_static_auto	114.972
parallelallomp_dynamic_auto	3576.22
parallelallomp_dynamic_100000	114.993
parallelallomp_dynamic_12500000	115.347
parallelallomp_dynamic_15000000	136.698

Table 1 CPU time of various PI implementations

From the data collected above, the parallel implementations have a speedup of around 6.9x compared to the sequential implementation. Although in the chart both the parallel and the parallel with omp implementation have approximately the same run time, it is expected that as I increase the workload (more iterations) I can begin to see a greater difference in their execution times. In fact, I should see a smaller execution time for the parallel with omp as it does not have additional storage/loop overhead that the first implementation has. From the table, it is evident that for the parallel implementation with full OpenMP, static scheduling with the default (auto) chunk size provides the best performance. In contrast, dynamic scheduling with a chunk size of 1 results in significantly higher execution times, even exceeding the sequential implementation. This is because dynamic scheduling introduces considerable overhead as it dynamically assigns tasks to threads at runtime. To optimize dynamic scheduling, there needs to be a balance in the chunk size. If the chunk size is too small (as mentioned above), the scheduling overhead dominates and reduces performance. However, if the chunk size is too large, it becomes more difficult to fully exploit parallelism and results in poor load balancing which can increase the execution time.