Cole Fettkether
COEN 145

Program 1: Tiling Matrix Multiplication

Introduction

This paper will explore the process of matrix matrix multiplication and how splitting the matrix into smaller subsections can improve the efficiency of the multiplication process. Dividing the matrix leads to speedup due to cache locality, as the program does not have to keep retrieving data from stored memory in different location, but can access it rather quickly as it is in the cache. By parallelizing these subsections, we can see impressive speedup in the algorithm.

Description

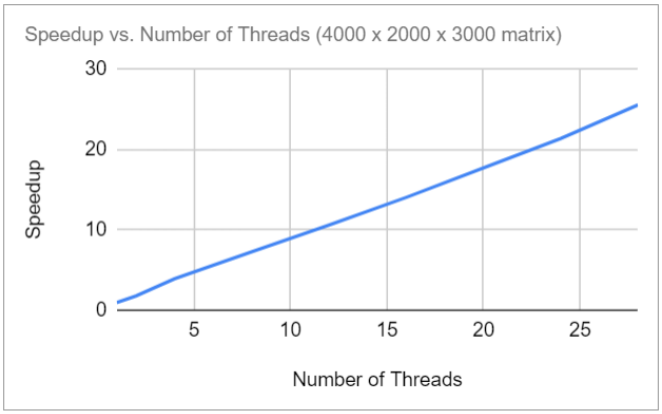
To parallelize the program, the following line is used:

```
#pragma omp parallel for shared(C, sum) private(i, j, k)
```

This loop uses the open mp interface to parallelize a for loop, where values for C, the output matrix, and sum, the values added up for one multiplication that are added to C, and the variables i, j, and k, the array indices, have their own copies created which allows multiple threads to run different parts of the loop.

Data and Analysis

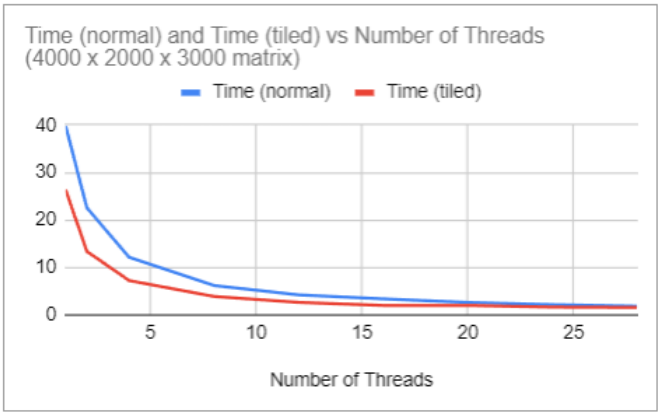
We can see from the following graph that adding multiple threads speeds up the program. This linear relationship makes sense as the work is divided evenly among the threads, thus, when you double the number of threads, you double the speedup.



Number of Threads	Time	Speedup
1	27.450927	1
2	14.943615	1.836966959
4	6.930526	3.96087209
8	3.755414	7.309693951
12	2.584728	10.62043163
16	1.954462	14.04526003
20	1.550768	17.70150467
24	1.286195	21.34274119
28	1.075866	25.51519148

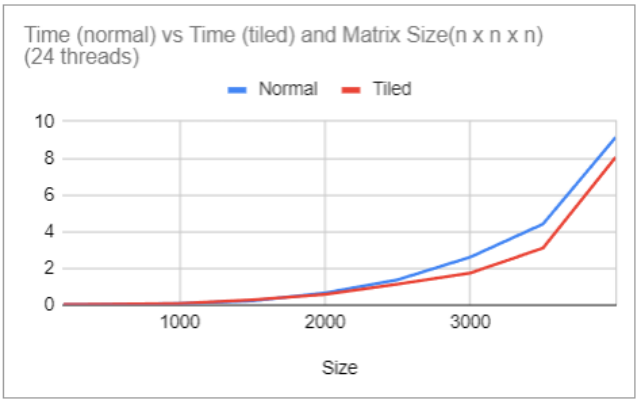
Now we can examine the effect of tiling on the matrix. From testing, it appeared that around 45 by 45 was the optimal tile size. This makes sense as the tiles being too large would result in the tiled matrix multiplication being similar to the original one, and not resulting in any speedup. Conversely, when the tile sizes are too small, it would not be as effective as there would be constant new memory calls. There are two things of interest when testing the tiled program vs normal program: how the number of threads affects the speedup, and how the matrix size affects the speedup.

Below is the graph demonstrating the time it takes for a tiled matrix multiplication vs non-tiled matrix multiplication. Clear from the data, tiling makes the matrix significantly faster (1.5x faster for one thread), but the effects have diminishing returns as the number of threads increases (1.18x faster for 28 threads).



Number of Threads	Time (normal)	Time (tiled)
1	39.858553	26.361597
2	22.546603	13.32637
4	12.063872	7.159545
8	6.119821	3.812687
12	4.139104	2.553546
16	3.30006	1.913413
20	2.539407	1.915704
24	2.073938	1.557876
28	1.753827	1.485941

Below is the data for different sized matrices, comparing the tiled run time vs normal runtime. Interestingly, the runtime is faster without tiling for lower matrix sizes, which could be due to the fact that the tile size is much closer to the matrix size.



Size	Normal	Tiled
200	0.001044	0.008367
500	0.009111	0.023719
1000	0.058949	0.063658
1500	0.205906	0.260673
2000	0.638302	0.558853
2500	1.357866	1.12011
3000	2.594919	1.724336
3500	4.401647	3.087375
4000	9.134544	8.04729

Conclusion

Overall, we see that tiling a matrix does lead to increased performance, due to cache locality. As matrix size increases, we see increased speedup with tile size. Especially for low numbers of threads, we see valuable speedup in the matrix multiplication. Typically, speedup is similar when we compare matrices with different dimensions multiplied by each other. This data and information is extremely relevant as matrix multiplication is the backbone of many algorithms including ML models, graph related problems, and computer graphics.