Shahid Beheshti University

Faculty of Mathematical Science

Department of Computer Science

Concurrent Programming – Spring 2022 – Assignment 4 – Simple Matrix Multiplication implementation with OpenMP

By:

Arman Davoodi

Introduction

In this assignment, Matrix multiplication Algorithm is written in C++ language in both parallel and sequential format. The purpose of this assignment is to compare the computation time of the parallel matrix multiplication algorithm using different scheduling methods provided by OpenMP.

The results given in this report are based on several runs on a system with Intel Core I7-8550U CPU and Windows 10 operating system.

Simple Matrix Multiplication

The brute force method to compute the dot product of two matrices is to implement the formula illustrated in *Equation 1* for all elements of C. *Algorithm 1* demonstrates the pseudo-code for this implementation.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \,.\, B_{kj}$$

Equation 1

Algorithm 1: Naïve Matrix Multiplication Algorithm(A[N][K], B[K][M]):

Define Matrix C of size NM.

```
1_  BEGIN
2_      for i from 0 to N do
3_          for j from 0 to M do
4_              C[i][j] = 0
5_              for k from 0 to K do
6_                  C[i][j] += A[i][k] * B[k][j]
```

7_ **endfor**

8_ **endfor**

9_ **endfor**

10_ **return** C

11_ **END**

It is clear that in *Algorithm 1* the instruction at line 6 is repeated N*M*K times. Therefore using this algorithm on two n by n matrices has a time complexity of $O(n^3)$.

Comparing Sequential and Parallel Implementations Using Default Scheduling

To parallelize this algorithm, omp parallel for directive is used on the outer loop of the code.

To compare time efficiency for these two implementations, both of them were run on 7 cases of different sizes. The result of each test is illustrated on *Table 1*. Duration is the average time of all of those runs in seconds. Also, the parallel algorithm is run on 8 threads.

Table 1

row	size		Duration(s)	Speedup
1	32×32 32×32	Sequential	0.000020	0.048
		Parallel	0.000413	
2	64×64 64×64	Sequential	0.001209	2.159
		Parallel	0.000560	
3	128×128 128×128	Sequential	0.007312	3.063
		Parallel	0.002387	
4	256×256 256×256	Sequential	0.057853	3.370
		Parallel	0.017169	
5	512×512 512×512	Sequential	0.568763	3.707
		Parallel	0.153420	
6	1024×1024 1024×1024	Sequential	5.165422	3.353
		Parallel	1.540610	
7	2048×2048 2048×2048	Sequential	77.582010	4.988
		Parallel	15.552244	

Chunk Size Tuning

To find the best chunk size for each scheduling algorithm, all these algorithms were run for two matrices of size 512 * 512 but with different chunk sizes. The runtimes of each test is shown in seconds in *Table 2*.

Table 2

	Chunk Size				
Scheduler	1	5	10	20	50
Static	0.159887	0.158931	0.170409	0.193674	0.251522
Dynamic	0.150089	0.153813	0.168305	0.186750	0.234359
Guided	0.151009	0.152819	0.159376	0.172715	0.230930

We can conclude from *Table 2* that in this problem chunk size of 1 is the best chunk size.

Comparing Different Scheduling Algorithms

In this section we compare the runtimes of different scheduling algorithms to see which scheduling works best for our problem. From the results illustrated in *Table 3* we can understand that in this problem the overhead of static scheduling makes it more costly in total whereas dynamic and guided scheduling decrease the runtime a bit. However in this problem, these amounts are so small that we can say that all of these algorithms take approximately an equal amount of time to run.

Table 3

	Matrix Size			
Scheduler	128×128	256×256	512×512	1024×1024
Sequential	0.007312	0.057853	0.568763	5.165422
Default	0.002387	0.017169	0.153420	1.540610
Static	0.002356	0.017702	0.158366	1.553314
Dynamic	0.002241	0.017081	0.150304	1.535990
Guided	0.002221	0.017060	0.149570	1.538591

Conclusion

In the end, we can see that by using a simple parallelization technique for matrix multiplication, we can get a speedup up to 5 using 8 threads.

Also, we can see that for this problem, using dynamic and guided scheduling can make our runtime better but not by much.