Shahid Beheshti University

Faculty of Mathematical Science

Department of Computer Science

Concurrent Programming – Spring 2022 – Assignment 3 – Strassen Matrix Multiplication implementation with OpenMP

By:

Arman Davoodi

Introduction

In this assignment, Strassen Matrix multiplication Algorithm is written in C++ language in both parallel and sequential format; and the performance of both implementations are compared with each other to get the amount of speedup gained by parallelizing this algorithm using 8 threads. The OpenMP library is used to parallelize the algorithm.

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).

Strassen Matrix Multiplication

Another approach is to use the divide and conquer paradigm to present an algorithm capable of computing dot products of two matrices with a time complexity lower than O(n^3).

To simplify the problem, we assume that both matrices are n by n square matrices with n being a power of 2.

With the above assumption we can rewrite our matrices like *Equation 2* in a way that for any i and j from 1 to 2, A_{ij}, B_{ij}, and C_{ij} will be of size n/2 by n/2.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad A.B = C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}, \qquad C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$$

$$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}, \qquad C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$$

Equation 2

If matrices A and B are 2 by 2 matrices, matrix C can be computed easily, However if the size of our matrices is 4 or higher we need to use this algorithm.

The first step after dividing the matrices is to construct ten matrices shown in *Equation 3*, then by using these matrices, 7 other matrices are constructed as shown in *Equation 4*. The last step shown in

Equation 5 uses matrices constructed in previous steps to compute matrices C_{11}, C_{12}, C_{21}, and C_{22}.

$$S_1 = B_{12} - B_{22}, \quad S_2 = A_{11} + A_{12}, \quad S_3 = A_{21} + A_{22}, \quad S_4 = B_{21} - B_{11}$$

$$S_5 = A_{11} + A_{22}, \quad S_6 = B_{11} + B_{22}, \quad S_7 = A_{12} - A_{22}, \quad S_8 = B_{21} + B_{22}$$

$$S_9 = A_{11} - A_{21}, \quad S_{10} = B_{11} + B_{12}$$

Equation 3

Since matrix addition and subtraction can be done in $O(n^2)$, the construction of matrices S_1 to S_{10} have a time complexity of $O(n^2)$.

$$P_1 = A_{11}.S_1 = A_{11}.B_{12} - A_{11}.B_{22}, \quad P_2 = S_2.B_{22} = A_{11}.B_{22} + A_{12}.B_{22}$$

$$P_3 = S_3.B_{11} = A_{21}.B_{11} + A_{22}.B_{11}, \quad P_4 = A_{22}.S_4 = A_{22}.B_{21} - A_{22}.B_{11}$$

$$P_5 = S_5.S_6 = A_{11}.B_{11} + A_{11}.B_{22} + A_{22}.B_{11} + A_{22}.B_{22}$$

$$P_6 = S_7.S_8 = A_{12}.B_{21} + A_{12}.B_{22} - A_{22}.B_{21} - A_{22}.B_{22}$$

$$P_7 = S_9.S_{10} = A_{11}.B_{11} + A_{11}.B_{12} - A_{21}.B_{11} - A_{21}.B_{12}$$

Equation 4

Matrices P_1 to P_7 can be computed by 7 recursive calls of this algorithm.

$$C_{11} = P_5 + P_4 - P_2 + P_6, \quad C_{12} = P_1 + P_2, \quad C_{21} = P_3 + P_4, \quad C_{22} = P_5 + P_1 - P_3 - P_7$$

Equation 5

Construction of matrices C_{11}, C_{12}, C_{21}, and C_{22} by using *Equation 5* has a time complexity of $O(n^2)$ due to the usage of matrix addition and subtraction methods.

Therefore the time complexity of Strassen matrix multiplication as shown in *Equation 6* is $O(n^{\lg 7})$ which is approximately $O(n^{2.81})$ and is lower than $O(n^3)$. This can be computed by using the master method.

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T\left(\frac{n}{2}\right) + \theta(n^2) & n > 1 \end{cases}$$

Equation 6

To compute dot products of non-square matrices or matrices with dimension sizes that are not a power of 2, padding method can be used in which some rows and columns of 0s are appended at the end of input matrices.

Sequential Implementation

To improve the performance of the algorithm, the implementation uses the naïve method to compute the dot product of matrices of size 64 by 64 or lower. This is due to the fact that the Strassen algorithm has a lot of overhead and therefore it is not efficient to use for small matrices. Moreover to reduce the amount of computations, instead of creating new matrices for A_{ij}, B_{ij}, and C_{ij}, starting indexes and matrix sizes are passed to the function. Also to avoid initializing 7 P_i matrices, these matrices are computed in S_j matrices.

Parallel Implementation

In parallel implementation, the naïve dot product algorithm, and matrix addition and subtraction methods are parallelized using the OpenMP parallel-for as their outer loop. Also a 'parallel omp sections' is used to compute S_i matrices. And the computation of each S_i matrix is done in a separate omp section. The same method of parallelization is used for computing P_i and C_{ij} matrices as well.

Comparison

To compare time efficiency for these implementations, both algorithms were run on 7 of different sizes. The result of each one 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	64×64	Sequential	0.000916	1.51
		Parallel	0.000606	
2	128×128	Sequential	0.006583	1.99
		Parallel	0.003314	
3	256×256	Sequential	0.047338	2.05
		Parallel	0.023147	
4	512×512	Sequential	0.337624	2.52
		Parallel	0.133969	
5	1024×1024	Sequential	2.375158	2.95
		Parallel	0.803882	
6	2048×2048	Sequential	16.872692	3.26
		Parallel	5.183055	
7	4096×4096	Sequential	118.203156	3.26
		Parallel	36.284472	

Conclusion

In the end, we can conclude that this parallel implementation of the Strassen algorithm can give us more than 3.2 times speedup on large enough matrices on the specified system with 8 threads.