Final Project
INF236: Parallel Programming
Parallel Matrix Multiplication

Kateřina Čížková, Luca Klingenberg

May 10, 2021

1 Introduction

Our goal was to implement parallel matrix multiplication. The standard algorithm has complexity $\mathcal{O}(n^3)$ and is straightforward to parallelize. We used its implementation as a reference. We compared it with Strassen's algorithm, which has complexity $\mathcal{O}(n^{\log 7})$ but is harder to implement.

2 Algorithms

In this section all the implemented algorithms are explained in further detail.

2.1 Matrix Multiplication

The standard $\mathcal{O}(n^3)$ algorithm is simple. Each element cc_{ij} of the resulting matrix is calculated as $c_{ij} = \sum^k a_{ik} \cdot b_{kj}$. It is possible to implement the calculation with only three nested for-loops, but matrices are stored row-wise, so it is better to access it in consecutive order. Because of that, we used the following implementation. Complexity is still $\mathcal{O}(n^3)$, but it runs faster.

Algorithm 1 Matrix Multiplication

Input: $\mathbf{A}, \mathbf{B}$
Output: $\mathbf{C}$ (the resulting matrix)

```
1: function MATMUL(A, B)
2:     for i = 0, ..., n − 1 do
3:         for j = 0, ..., n − 1 do
4:             c[i][j] = 0
5:         for k = 0, ..., n − 1 do
6:             for j = 0, ..., n − 1 do
7:                 c[i][j] += a[i][k] · b[k][j]
8:     return C
```

2.2 Parallel Matrix Multiplication

We just parallelized the outermost loop of the simple matrix multiplication algorithm described in previous section.

2.3 Strassen Algorithm

The matrix multiplication can be formulated in terms of block matrices:

$$A \cdot B = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{01}B_{11} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = C$$

According to this formula, it needs 8 multiplications of half-size matrices. The idea of Strassen's algorithm is to use more additions and subtractions, but only 7 multiplications half-size matrices. The algorithm is used recursively for these multiplications.

Algorithm 2 Strassen Matrix Multiplication

Input: $\mathbf{A}, \mathbf{B}$
Output: $\mathbf{C}$ (the resulting matrix)

1: **function** STRASSEN($\mathbf{A}, \mathbf{B}, n$)
2: **if** n == cutoff **then**
3: **return** MATMUL($\mathbf{A}, \mathbf{B}$)
4: $\mathbf{P}_1 = \text{STRASSEN}(\mathbf{A}_{00} + \mathbf{A}_{11}, \mathbf{B}_{00} + \mathbf{B}_{11}, \frac{n}{2})$
5: $\mathbf{P}_2 = \text{STRASSEN}(\mathbf{A}_{10} + \mathbf{A}_{11}, \mathbf{B}_{00}, \frac{n}{2})$
6: $\mathbf{P}_3 = \text{STRASSEN}(\mathbf{A}_{00}, \mathbf{B}_{01} - \mathbf{B}_{11}, \frac{n}{2})$
7: $\mathbf{P}_4 = \text{STRASSEN}(\mathbf{A}_{11}, \mathbf{B}_{10} - \mathbf{B}_{00}, \frac{n}{2})$
8: $\mathbf{P}_5 = \text{STRASSEN}(\mathbf{A}_{00} + \mathbf{A}_{01}, \mathbf{B}_{11}, \frac{n}{2})$
9: $\mathbf{P}_6 = \text{STRASSEN}(\mathbf{A}_{10} - \mathbf{A}_{00}, \mathbf{B}_{00} + \mathbf{B}_{01}, \frac{n}{2})$
10: $\mathbf{P}_7 = \text{STRASSEN}(\mathbf{A}_{01} - \mathbf{A}_{11}, \mathbf{B}_{10} + \mathbf{B}_{11}, \frac{n}{2})$
11: $\mathbf{C}_{00} = \mathbf{P}_1 + \mathbf{P}_4 - \mathbf{P}_5 + \mathbf{P}_7$
12: $\mathbf{C}_{01} = \mathbf{P}_3 + \mathbf{P}_5$
13: $\mathbf{C}_{10} = \mathbf{P}_2 + \mathbf{P}_4$
14: $\mathbf{C}_{11} = \mathbf{P}_1 - \mathbf{P}_2 + \mathbf{P}_3 + \mathbf{P}_6$
15: **return** $\mathbf{C}$

We used Winograd's version of Strassen's algorithm described in [**boyer2009memory**]. It needs only 15 additions and subtractions instead of 18 because some results of these additions and subtractions are reused. We also used scheduling table 1, therefore our implementation needs only two temporary matrices in each recursive call to store intermediate results.

Strassen's algorithm is recursive and it is working with quarters of input matrices. Because of that, it is useful to use z-ordering for storing matrices in our implementation. Its advantage is that all submatrices are stored in consecutive parts of memory. But we switch from Strassen's algorithm to the simple matrix multiplication when the matrices are too small so Strassen's algorithm overhead is too big. For the simple matrix multiplication is better to have the matrices ordered row-wise. We combined both and used the z-ordering down to the cutting point of recursion. We left the small submatrices which are multiplied with the simple multiplication algorithm in row-wise order.

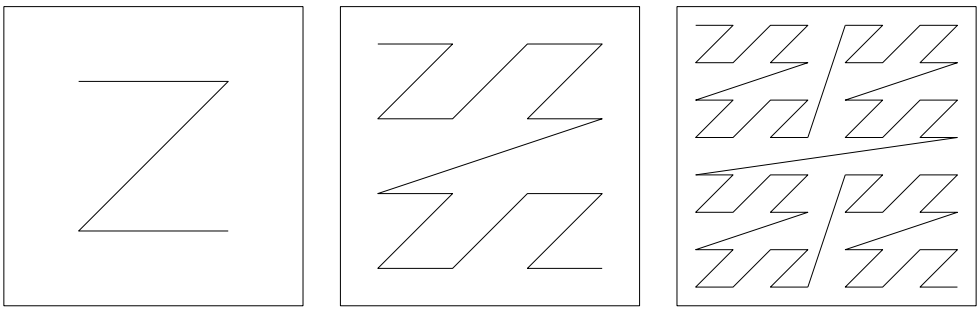


Figure 1: Z-ordering

2.4 Parallel Strassen Algorithm

It is straightforward to parallelize additions and subtractions in Strassen's algorithm. The problem is that for small matrices it is not worth it to initialize a parallel region. Because of that, we can not just insert `#pragma omp parallel for` before the addition and subtraction for-loop.

Then, we were thinking about parallelizing recursive calls – calculate P_1 to P_7 in parallel. But since we are using only two temporary matrices in each recursive call, it is necessary to finish a recursive call before starting the

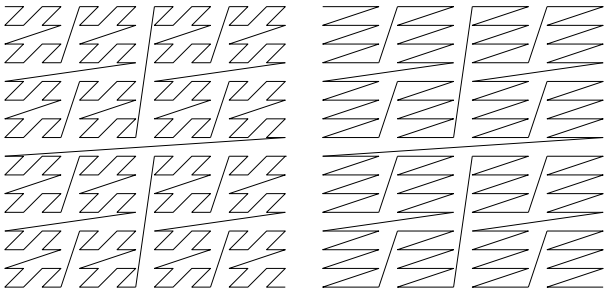


Figure 2: Z-ordering

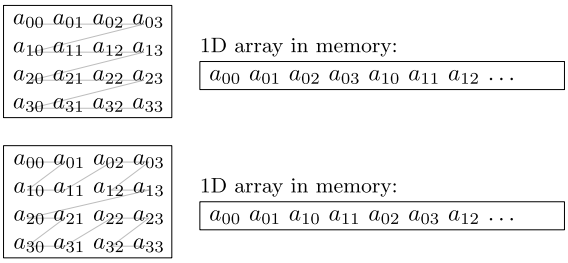
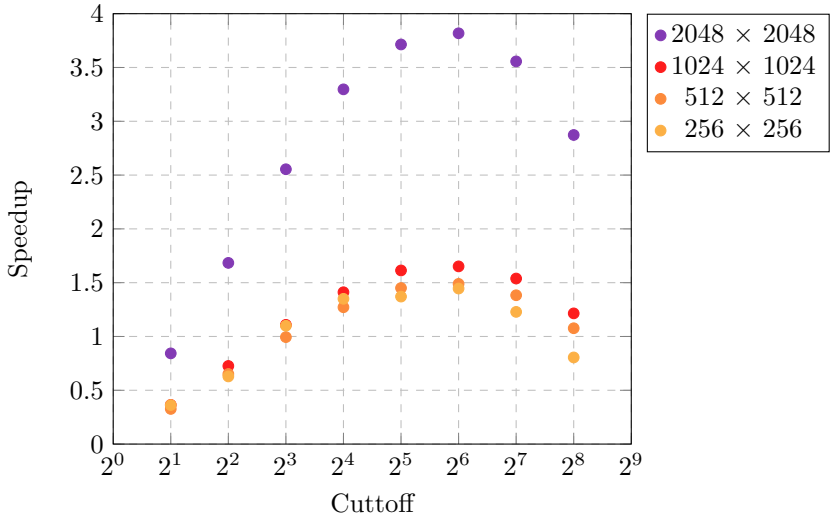


Figure 3: Z-ordering

next one. This may be solved using more space for intermediate results, but we did not try that.

3 Experiments

Sequential Strassen Algorithm with different values for the cutoff level



4 Conclusion