Uppsala University

Parallel and Distributed Programming

1TD070

Assignment 2 Report (Group 12)

Authors:
Emil Edvardsson
Hamza Aziz Khan

April, 2023

Contents

1 Problem description

The task of this assignment is to implement a code that can perform matrix-matrix multiplication for two N by N matrices (A and B) and write the resulting matrix (C) to a text file. The code is implemented using C and MPI. Finally we analyse the performance of our implementation through weak and strong scalability analysis.

2 Method

Our strategy for computing matrix-matrix multiplication in parallel is to partition the rows of matrix A and the columns of matrix B to all processes, while requiring the dimensions N of the matrices to be divisible with the number of processes.

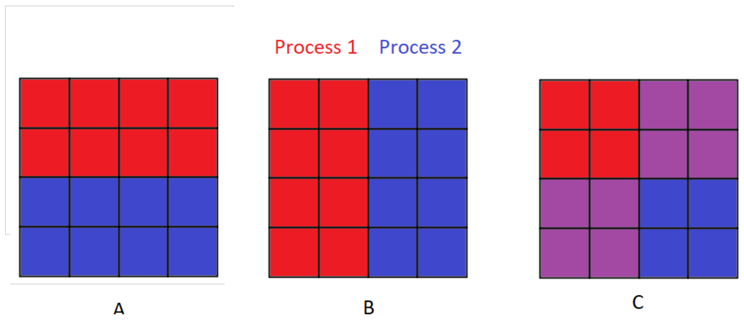

Figure 1: Partition of rows/columns to all processes. Red area in A and B is local to process 1 and blue area in A and B is local to process 2. In matrix C, red marks the elements calculated solely by process 1 and blue for process 2. The purple area marks the elements that needs communication between process to be calculated.

The way partitioning of the columns and rows as well as the computation of the result is presented in figure 1 above. Here we see that parts of the resulting matrix can be calculated by each process individually. In our implementation we use non-blocking send communications, where all processes sends their part to all other processes before starting to calculate their part of the resulting matrix. At every iteration a blocking receive is implemented before that specific part is calculated, which should be readily available since all local data has already been sent to the other processes. Every process has their own $N * number\_of\_processes$ dynamically and locally allocated arrays, used to store the result of their part of the operation. After all processes have calculated their part, the result is stored in the original C matrix (represented as a 1d array) by the root process using MPI_Gather. A pseudo-representation of the algorithm used is shown below.

Algorithm 1 Matrix Matrix multiplier in parallel

```
1: Read full input on root process
2: Scatter input to all processes
3: All processes **ISend** their local data to all other processes
4: **for** number of processes **do**
5:     Each process calculates part of C matrix using local and **IRecv** data for the specific B column needed
6: **end for**
7: root process gathers and writes result to output file
```

MPI_Datatype was used in order to be able to scatter the data in a single scatter statement. The rows was converted to a contiguous block of memory while the columns had to be resized.

3 Numerical experiments description

figure 2 is the strong scaling speedup which plots number of processes against speedup. the values used for this are in table 1. figure 3 shows the strong scaling efficiency. the values for this can be found in table 1. both figure 2 and 3 were done for input3600.txt
figure 4 and 5 show weak scalability and use tables 2 and 3 respectively.

4 Numerical experiments results

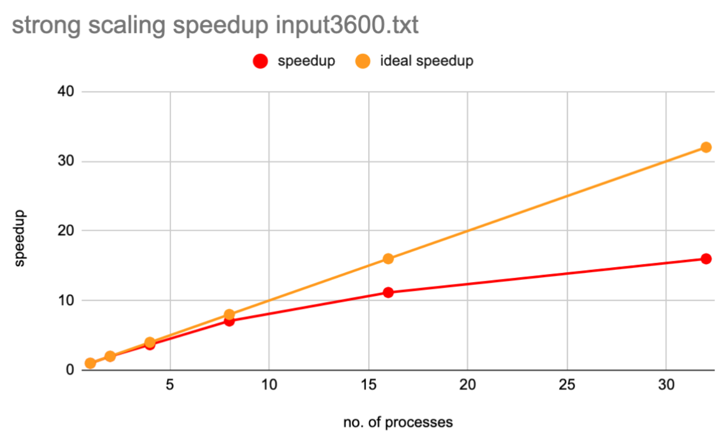

Figure 2: Strong Scaling Speedup. Speed reduces considerably after 8 processes

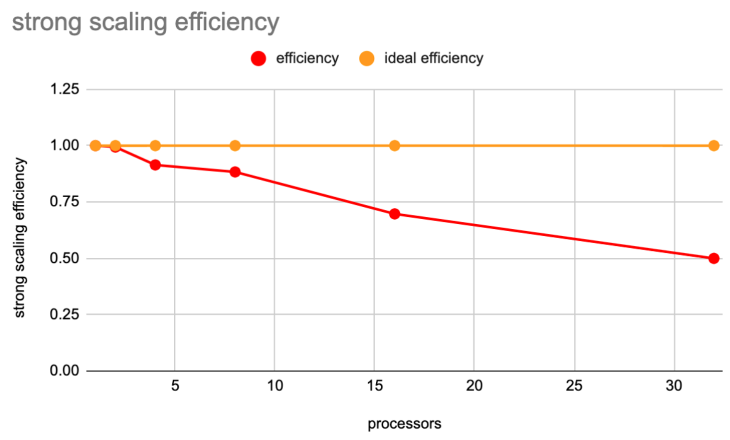

Figure 3: strong scaling efficiency. efficiency drops after 8 processes

Table 1: Strong Scaling

processors	time	speedup	efficiency
1	58.065	1	1
2	29.217	1.987388	0.9936944
4	15.876	3.6574229	0.91435572
8	8.2176	7.06591859	0.88323982
16	5.2056	11.1542217	0.69713885
32	3.6315	15.9890771	0.49965866

From figure 2, as the number of processes increases the speedup increases but at a decreasing rate each iteration. This is due to communication overhead which increases as the number of processes increase. The ideal number of processes to choose is one that reduces the serial time considerably, but also does not waste too much time on communication (inefficient). From Figure 3, after 8 processes, the efficiency seems to drop and figure 2 shows that speedup does not increase significantly.

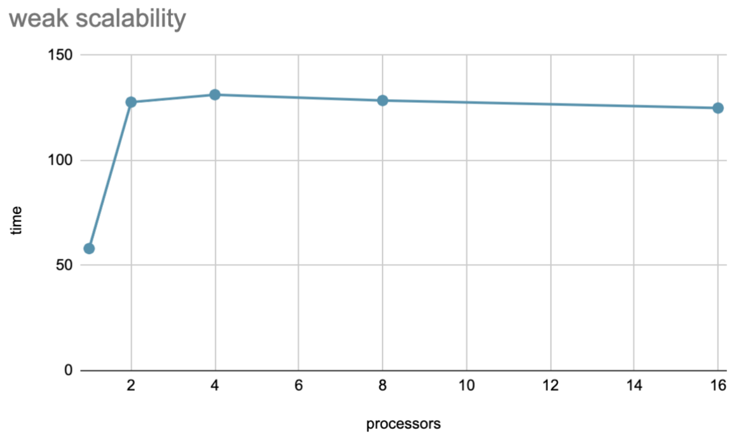

Figure 4: Weak scaling analysis with the matrix size increasing slightly

Table 2: weak scaling

processors	time	matrix size
1	58.065	3600x3600
2	127.741642	5716x5716
4	131.27556	7488x7488
8	128.51112	9072x9072
16	124.919179	10525x10525

Initially in figure 4 and table 2, the number of processes was doubled and the matrix size was increased slightly. This led to some processes having less work than others as the number of processes increases at a faster rate than the matrix size. Ideally we would want each process to have roughly the same amount of work producing a linear graph as the matrix size increases. This was done in table 3 and figure 5.

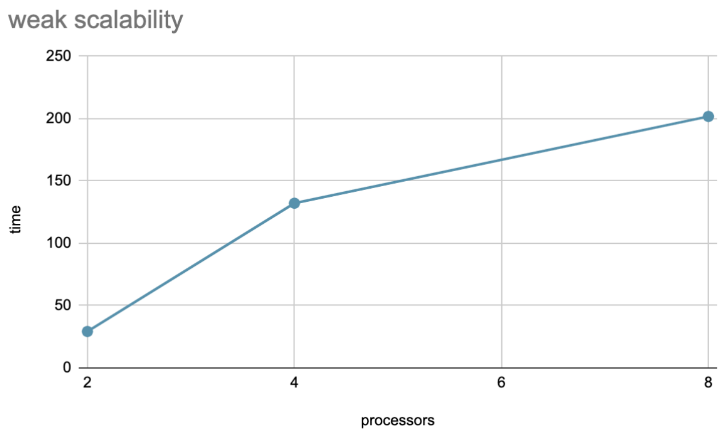

Figure 5: Weak scaling analysis with doubling matrix size per iteration

Table 3: weak scaling

processors	time	matrix size
2	29.217	3600x3600
4	132.129072	7488x7488
8	201.755031	10525x10525

Ideally, as the matrix size and processes doubles the execution time should remain the same or only increase slightly because the work per processor remains the same. However, we have a steep linear curve showing that execution time seems to increase significantly. This is because as the matrix size doubles the arithmetic work becomes eight times more and so each process has more work to do.