Parallel and Distributed Programming – Assignment 2
Group 15

Claude Carlsson
Edvin Germer
Ture Hassler

April 20, 2023

UPPSALA
UNIVERSITET

Contents

1 Introduction

Matrix multiplication is a very common mathematical operation that is at the core of linear algebra, and therefor very relevant in many fields of science. The time complexity for the standard matrix multiplication algorithm is $\mathcal{O}(n^3)$, so it is highly desirable to optimize this operation so that it runs as fast as possible for large matrices. The mathematical operation of matrix multiplication can be optimized both serially and using parallelization. But relevant for this assignment, is to optimize the operation using parallelization with MPI processes.

Given that we have matrices $A = \{a_{i,j}\}$ and $B = \{b_{i,j}\} \quad i, j = 1, 2, \ldots, n$.
$A, B \in \mathbb{R}^{n \times n}$. Matrix multiplication can then be mathematically described as:
$C = AB$ or elementwise as seen in equation 1.

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}, \tag{1}$$

2 Parallelization

2.1 Matrix partition and algorithm

Our proposed solution is a mix of row-wise and column-wise 1D partitioning, where matrix A is partitioned row-wise and matrix B column-wise.

Given matrices A and B of size nxn, and p parallel MPI processes, the matrix elements are distributed to the processes in "blocks" of size mxn or nxm where m is the "width" of each block and is calculated as $\frac{n}{p}$. In practice, this means that at any one time of the matrix multiplication each rank has access to a block of rows from matrix A and a block of columns from matrix B, which can be multiplied together to give a square block matrix of size mxm which correspond to a sub-block in the resulting C matrix.

After the row and column blocks have been multiplied together, the processes have to exchange elements so that the remaining blocks in the C matrix can be calculated. It can be chosen arbitrarily if the process should exchange the A elements (block of rows) or the B elements (block of columns), in this case, the A elements are kept the same on all processes throughout the matrix multiplication and the B elements are sent to a neighbouring process after each sub-block of C has been calculated. The calculation of a sub-block in the C matrix is referred to in this report as a "stage" of the calculations. To perform the complete matrix multiplication, the program iterates over as many stages as there are processes. The process is illustrated in figure 1 for a $3x3$ matrix with 3 parallel MPI processes.

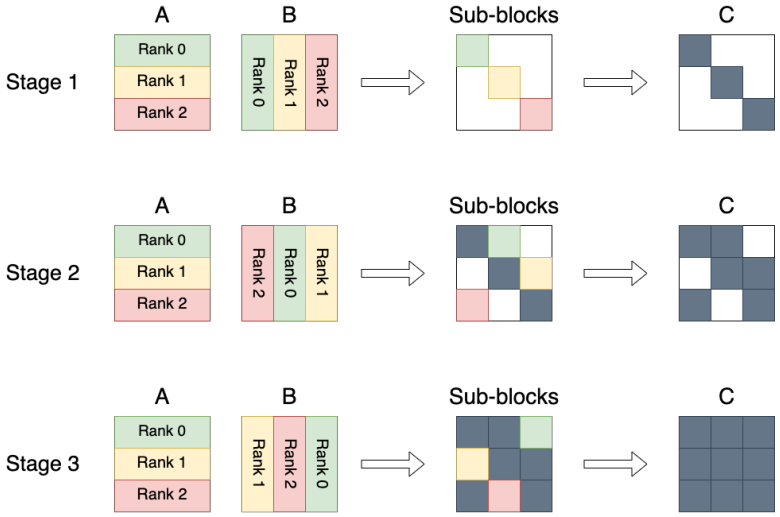

Figure 1: Simple illustration of parallel matrix multiplication

As we can see in figure 1 the first stage calculates the diagonal sub-blocks of the C matrix, one sub-block per process. The next stages then iteratively step one sub-block to the right in the C matrix, until all sub-blocks have been calculated and the matrix multiplication is complete.

To summarize, the large full matrices A, B and C are stored on the first rank, which handle the I/O. All processes then store local blocks of the A and B matrices which is used to calculate the corresponding C block. To efficiently make use of the memory, the matrices are stored as 1D arrays. This results in an efficient solution at the cost of readability and complicated indexing of the arrays. In figure 2 we can see an example of how a $4x4$ matrix is distributed across two parallel MPI processes and how to resulting arrays look.

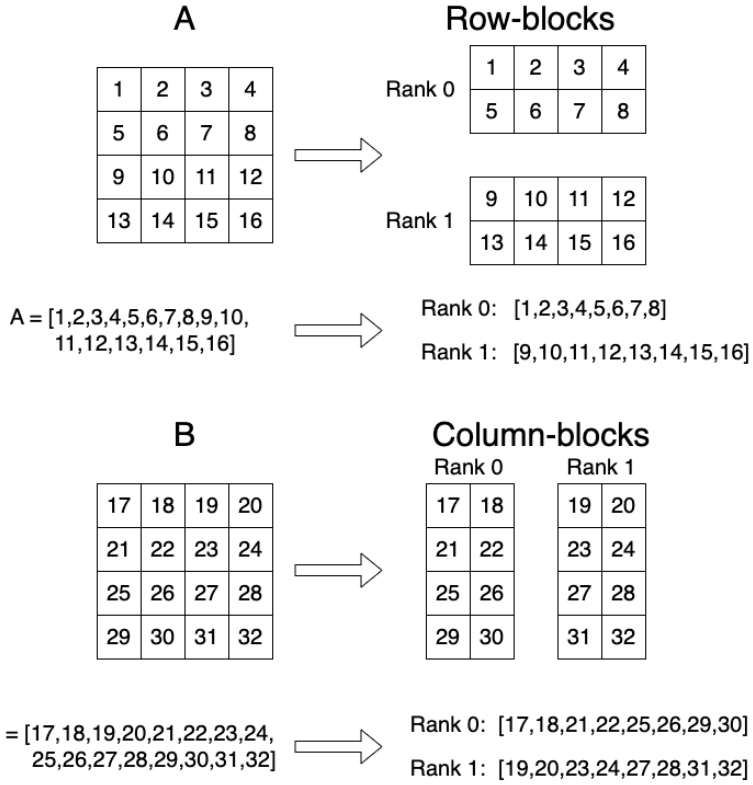

Figure 2: Illustration of how the distributed matrices are stored

The resulting distributed matrices have to be indexed differently for the A and B matrices. For the row-blocks you select element k on row i with $A\_block[i * n + k]$. For the column-block you select element k on row i with $B\_block[i * m + k]$. Where n is the size of the nxn matrix, p is the number of parallel processes and m is the block-width calculated by $\frac{n}{p}$.

2.2 Implementation

We implemented the algorithm in C using the MPI library. The first process reads the data and then broadcasts the size n to all other processes using **MPI_Bcast**. The first process then partitions and distributes the sub-matrices of A and B to the different processes using **MPI_Scatter** and **MPI_Send** while the other processes uses **MPI_Recv**. This ensures that each process gets their own distributed piece of the matrices, reducing the memory pressure and allowing the algorithm to work for very large matrices where it would be unfeasible to keep the entire matrix in the memory.

Each process then perform the matrix multiplication on the sub-matrices and saves it locally. After this the B sub-matrix is sent to the next neighbouring process, corresponding to shifting the column one step to the right, see figure 1. This is done using **MPI_Sendrecv**.

After the matrix multiplication is done each process has computed their corresponding row of sub-matrices in the C matrix and the results are sent back to process 0 using **MPI_Gather**.

3 Performance experiments and results

For the performance experiments we performed measurements of both the strong and the weak scaling, and in addition to this we also measured the memory usage.

3.1 System specifications

All timed measurments were performed on the UPPMAX high performance computing centre.

System: Uppmax Cluster "snowy"
Processor: Intel Xeon E5-2660
Operating System: CentOS Linux 7 (Core)
Compiler version: gcc 12.2.0
MPI version: openmpi 4.1.4

3.2 Performance experiments

Strong scaling is how the time changes when keeping the problem size fixed while varying the number of processing elements. We measured this using a fixed 3600x3600 matrix size and varying the number of processors from 1 to 16.

Weak scaling is how the time changes when the work per processing element is kept constant, which means increasing both the number of processors and the size of the matrix. However, we need to take into consideration that the algorithm is $\mathcal{O}(n^3)$, this means that when we double the size of the matrices we need to increase our processing units with $2^3 = 8$ times. We are going to test the following matrix sizes for weak scaling: 3600, 5716, 7488, and 9072, with 1, 4, 9, and 16 processing units respectively.

3.3 Results

3.3.1 Strong scaling

Table 1: The experiment values from strong scaling when using matrices of size 3600x3600

Processes	Measured time	Ideal time
1	623,94	623,94
2	294,53	311,97
4	147,04	155,98
8	59,25	77,99
16	25,61	39,00

Table 2: The speedup values from strong scaling when using matrices of size 3600x3600

Processes	Measured	Ideal
1	1,0	1,0
2	2,1	2,0
4	4,2	4,0
8	10,5	8,0
16	24,4	16,0

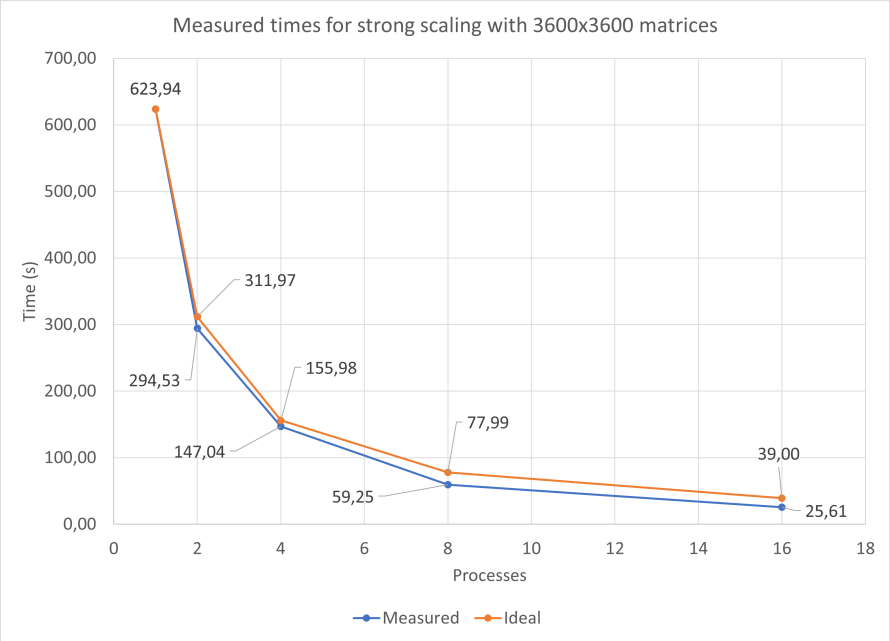

Figure 3: Plot with measured and ideal times for the strong scaling when using matrices of size 3600x3600

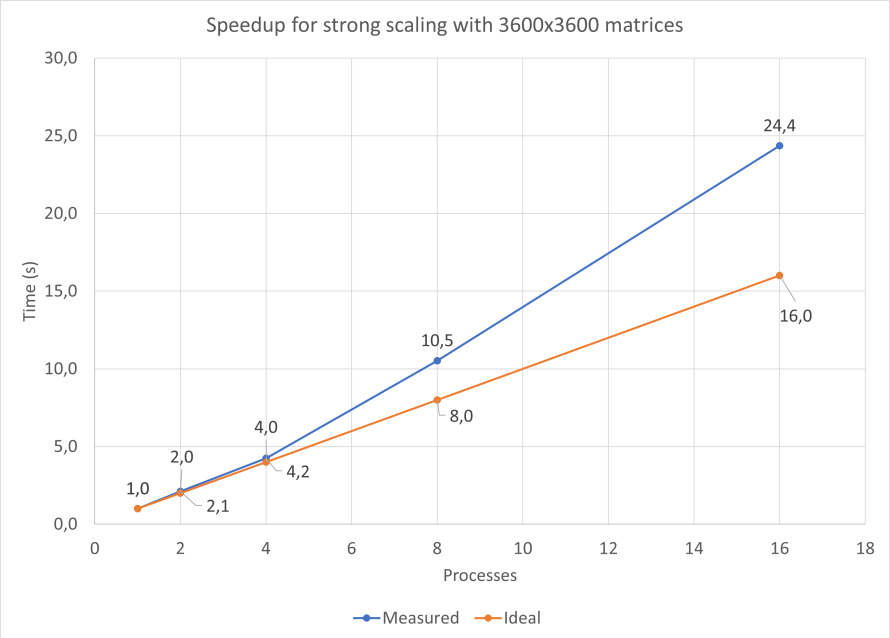

Figure 4: Plot with measured and ideal speedup for the strong scaling when using matrices of size 3600x3600

3.3.2 Weak scaling

Table 3: The experiment values from weak scaling

Processes and matrix size	Measured time	Ideal time
1 (3600x3600)	623,94	623,94
4 (5716x5716)	599,07	623,94
9 (7488x7488)	546,37	623,94
16 (9072x9072)	584,26	623,94

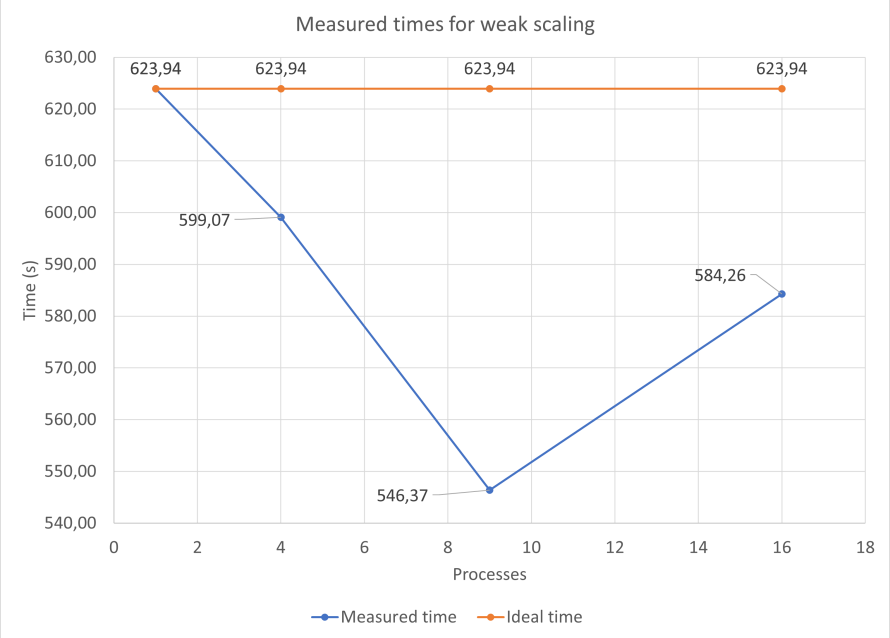

Figure 5: Plot with measured and ideal times for the weak scaling

3.4 Memory usage

After we ran the matrix 9072x9072 with 16 processes we examined how much memory the job used on UPPMAX. By running:

```
1 sacct -j <job_id> --format=JobID,JobName,MaxRSS
```

we can see what our peak memory usage was during the execution of the job. Our job used approximately 131,637KB.

4 Discussion and analysis

Based on our scaling experiments, we can see that we achieved better than ideal scaling for both strong and weak scaling, which can be seen in figure 3 and 5 respectively. This is also known as super-linear speedup, which can be seen in figure 4 for strong scaling. For our tests, we have used large matrices, this results in that most of the computation occurs locally on each of the processes, which reduces the communication overhead as a percentage of execution time.

Since we are partitioning the matrices, we can use the cache of each of the processing units. This may lead to fewer cache misses, but also be able to fit the whole blocks into the cache, which otherwise would not perhaps be possible for a single processing unit. This will probably contribute to the super-linear speedup that we have observed.

However, the matrices that we have used have an even distribution of data in them. This scenario is ideal for our algorithm since it will reduce the idle time of each processing unit. If instead, the matrices would have a skewed data distribution, it could potentially lead to an increase in the idle time for some of the processors.

4.1 Future improvements

In our algorithm, we have seen excellent scaling performance, however, the partitioning of our matrices is done in 1D. A potential improvement would be to use 2D partitioning by using checkerboard. This might further evenly distribute the workload among the processing units, especially if the matrix has an uneven data distribution.