Assignment 2 - Dense matrix-matrix multiplication

Emy Engström
Jacob Kovaleff Malmenstedt
Viveka Olsson

April 2023

UPPSALA
UNIVERSITET

---

1 Problem description

The problem setting is to create a program that performs matrix-matrix multiplication with Message Passing Interface, MPI. The multiplication is defined as $C = AB$ where $A = a_{i,j}$ and $B = b_{i,j}$ which gives the product $C_{i,j} = \Sigma_{k=1}^{n} a_{i,k} b_{k,j}$ (for $A, B \in R^{nxn}$, $i, j = 1, 2, ..., n$). An important part of this assignment is to choose an appropriate partitioning strategy for partitioning the matrices A, B and C. For example could Column-wise, Row-wise or Checkerboard partitioning be used to design the program.

The program should take an input file name and an output file name as the two only arguments. The input file must contain the A and the B matrix, and will therefore consist of $2n^2 + 1$ numbers. The program should be possible to use with different matrix sizes but the assumption can be made that n can be divided by the number of processes. Timing must be implemented in the program.

2 Implementation

Before implementing the matrix multiplication was a partitioning strategy chosen. Matrix A was partitioned row-wise and matrix B was partitioned column-wise. This strategy is relatively easy to implement when using MPI commands, since distributing rows and columns isn't too complicated. One down side was that keeping track of all the indexes when calculation the matrix multiplication was difficult and caused many errors. Another drawback could be that column- and row-wise partitioning use more memory in comparison to checkerboard partitioning.

The main function starts by taking two input arguments - the name of the input file and the name of the output file. A suitable command for running the program could for example be $mpirun - np4matmulinput.txtoutput.txt$ to run the program with 4 processes. MPI is initialized immediately and thereafter is the input file read by the process with rank 0. Space for matrix A and matrix B is allocated with a command that looks like this: $malloc(n * n * sizeof(float))$. The matrices A and B are scanned with $fscan$ and all values are put into a matrix each. The value n, which is the size of one side of the matrix, is broadcasted to all other processes with $MPI\_Bcast(\&n, 1, MPI\_INT, 0,$ $MPI\_COMM_WORLD)$. After n has been broadcasted space for A_chunk and B_chunk is allocated with $malloc(n * num_rows * sizeof(float))$ and C_chunk is allocated with $malloc(num\_rows * num\_rows * sizeof(float))$. The chunk matrices is the matrices where each of the processes receive and calculate their part of the whole matrix. The next step is for the program to scatter the matrix B between all processes with $MPI\_Scatter(B, n * num\_rows, MPI\_FLOAT, B\_chunk, n * num\_rows,$ $MPI\_FLOAT, 0, MPI\_COMM\_WORLD)$.

The following part of the code is for each of the processes to calculate their part of the matrix C. This is executed by iterating over $size$ which is the number of processes used. First the A_chunk is calculated by two further for loops with $A\_chunk[j * n + i] = A[(i * n + j) + l * num_rows]$. Thereafter is the A_chunk broadcasted by $MPI\_Bcast(A\_chunk, n * num\_rows, MPI\_FLOAT, 0, MPI\_COMM\_WORLD)$. The next part of the loop is for C_chunk to be calculated. This is done by multiplying A_chunk with each row in B_chunk. At the last part of the loop the chunks are gathered in C by the process with rank 0. This is done by letting all the processes send with

MPI_send to the process with rank 0, and then letting that process receive the data using *MPI_Recv*.

At the end of main is the output file written, all allocated spaces freed and MPI finalized. The output file is written with the same function *write_output* as used in stencil.c in Assignment 1 but with a few changes to have the data presented in a row-wise major. The function *write_output* is defined before the main code. The whole code, except for reading the input and writing the output is timed in the same way as in Assignment 1.

3 Description of numerical experiments

For the weak scalability experiments a matrix of size 1800x1800 with one process was used, as well as a matrix of size 3600x3600 with 4 processes and a matrix of size 7488 x 7488 with 16 processes, since each process then has around the same work load. For the strong scalability tests the matrices of size 3600x3600 and 5716x5716 was used and tested on 1, 4 and 16 number of processes. All timings have been excecuted on UPPMAX.

4 Results

4.1 Weak scaling

The resulting time measurements for the weak scalability performance experiments can be seen in table 1.

Table 1: Running times for weak scalability

N.o processors	Size of matrix	Running time[s]
1	1800	6.378053
4	3600	22.42885
16	7488	68.128391

4.2 Strong scaling

The resulting time measurements for the strong scalability performance experiments can be seen in tables 2 and 3. The strong scalability has been tested twice, one time for a 3600x3600 matrix and and the other for matrix 5716x5716.

Table 2: Running times for strong scalability with 3600x3600 matrix

N.o processors	Running time[s]
1	51.402817
4	22.42885
9	14.008527
16	11.653639

Table 3: Running times for strong scalability with 5716x5716 matrix

N.o processors	Running time[s]
1	208.716208
4	74.376177
16	34.996749

Strong scaling and speedup is different names for the same thing. The speedup for the program has been made visible in graphs 1 and 2 with different matrix sizes.

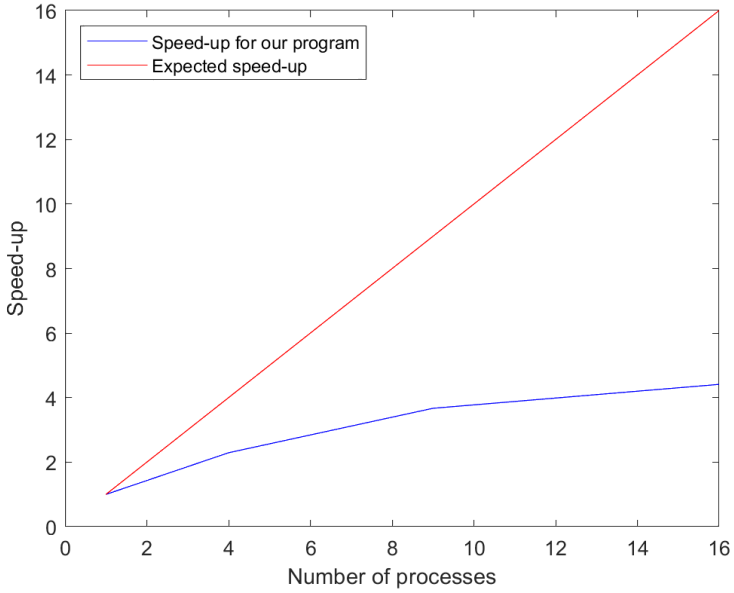

Figure 1: Speedup with 3600 matrix

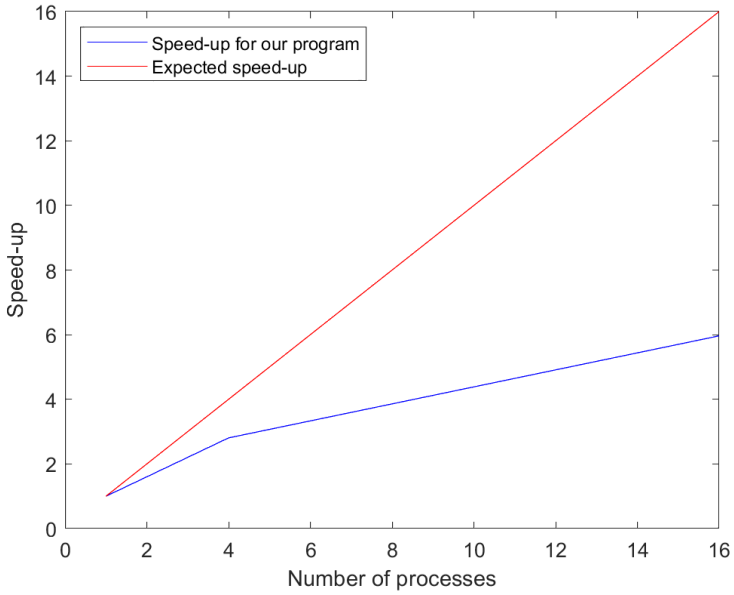

Figure 2: Speedup with 5716 matrix

5 Discussion

The result of this program is mainly expected. The parallelization makes the execution of the program faster, which was the purpose of the assignment, and it is obvious in the speedup graphs that the more processes the faster the program. The speedup and strong scalability is good but not optimal which is very normal. The weak scalability, however, is worse than expected. An ideal weak scaling would be to have as equal time as possible for all runs, but this is not the case for this program. The program would probably have to be implemented in a more effective way to get better scaling and raise the performance.