UPPSALA
UNIVERSITET

Assignment 2
Parallel and Distributed Programming

Written by:
Jakob Gölén
Simon Persson

April 24, 2022

1 Problem Description

Given two square matrices A and B of size N, a matrix-matrix multiplication can be calculated in serial and stored in a matrix C by using equation 1. Visualized in equation 2.

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} * B_{kj} \tag{1}$$

$$\begin{bmatrix} A_{0,0} & \dots & A_{0,N-1} \\ A_{1,0} & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ A_{N-1,0} & \dots & A_{N-1,N-1} \end{bmatrix} \times \begin{bmatrix} B_{0,0} & \dots & B_{0,N-1} \\ B_{1,0} & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ B_{N-1,0} & \dots & B_{N-1,N-1} \end{bmatrix} = \begin{bmatrix} C_{0,0} & \dots & C_{0,N-1} \\ C_{1,0} & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ C_{N-1,0} & \dots & C_{N-1,N-1} \end{bmatrix} \tag{2}$$

The goal of this assignment is to implement a matrix-matrix multiplication algorithm in C using OpenMPI that can perform the multiplication in parallel, and then evaluate the performance. The program was supposed to take an input file where the A and B matrices are stored and output a file where the resulting C matrix is stored.

2 Implementation

2.1 Algorithm

In order to implement a parallel matrix-matrix multiplication algorithm, Cannon's algorithm was chosen for the calculations. Cannon's algorithm is well suited for 2D-grids which the matrices are, and if a two dimensional cartesian configuration was chosen for the processing elements, each processor only has to communicate with its nearest neighbors during the calculations, unlike for example Fox algorithm where blocks of the matrix has to be sent to every process in the same rows as the block. Therefore it is assumed that $\sqrt{p}$ is an integer, with p being the number of processing elements, so the processes can be laid out in a two dimensional periodic grid. It is also assumed that $\frac{N}{\sqrt{p}}$ is an integer, so the processes can split the matrix evenly into blocks.

The main idea of Cannon's algorithm is to split up the A and B matrices into blocks of equally sized submatrices. Unlike Fox algorithm, an initial shift has to be performed. Each processing element check which row and column it is on in the two dimensional grid and sends the A submatrix as many steps to the left as its row number starting with row 0 and sends the B submatrix as many steps upwards as its column number starting with column 0. Concretely this means that for example the processing element on row 2 column 2 will send its A submatrix to the processor two steps left and recieve an A submatrix from the processor two steps to the right. It will send its B submatrix to the processor two steps upwards in the grid and recieve a B submatrix from the processor two steps downward in the grid. See figure 2 for more detail.

With the initial shift done, each processing elements calculates the matrix-matrix multiplication in its local submatrix using equation 1 and adds it to a local result matrix. Then, The local A matrices shifts one step to the left and the local B matrices shifts one step upwards, then every processing element calculates the matrix-matrix multiplication in its local submatrix and the local matrix-matrix multiplication is performed again.

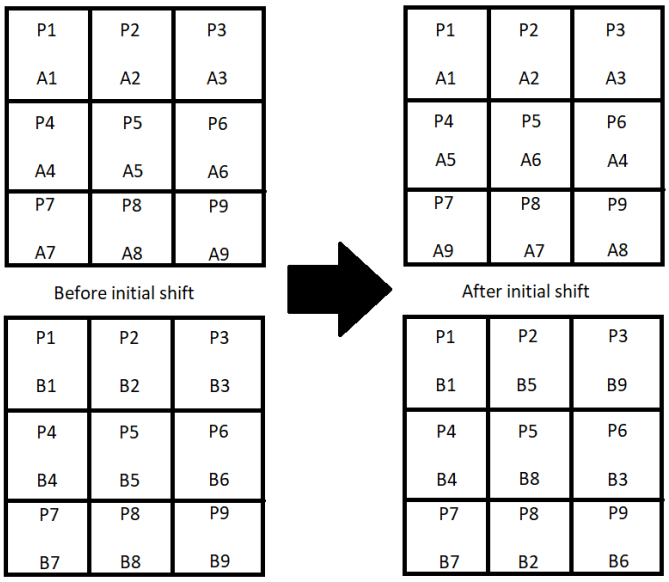

Figure 1: How the submatrices A1 - A9 and B1 - B9 shifts between the processing elements P1 - P9 as the initial shift is performed.

This is done $\sqrt{P}$ times, eg. once if there are one processing element, twice if there are four processing elements and so on.

2.2 Code implementation

Firstly, the input and output names are saved and MPI is initialized. A cartesian topology is created using MPI_Cart_create and every processing element's rank, row number and column number is saved. The neighbors to the north, south, east and west were found by using MPI_Cart_shift. Next, the processing element which has rank zero reads the input file and allocates vectors to store the A and B matrix. The matrix size is sent to every processing element and next, a vector datatype is created using MPI_Type_Vector in order to split the local vectors into blocks. In order to use the vector type, all matrices were stored as vectors.

Next, the A and B matrices were split into equally sized blocks and these submatrices were scattered to the processing elements using MPI_Scatterv and the vector data type. Then memory for the full A and B matrices were freed since they were no longer needed.

The initial shift was performed by first finding the shift length with MPI_Cart_shift with the shift length corresponding to the processing elements row number for the local A matrix and and column number for the local B matrix. Then the shift was performed with MPI_Sendrecv_replace.

Every processing element created a local result vector C and set the elements to zero. A timer started and then, Cannon's algorithm was implemented as described in section 2.1. The shifts were again performed with MPI_Sendrecv_replace to and from the neighbors of the processing elements.

When the matrix-matrix multiplication was done, the timer was stopped. The local result vector was gathered to the process with rank 0 using MPI_Gatherv and the vector datatype. The timings from every processor was also gathered with MPI_Gather and stored in a vector by the processor with rank 0. The highest time was calculated and

printed to the console and the output file was written.

3 Performance Experiments

When the program was written and the output results had been confirmed to be correct, the next step was to measure the parallel performance. All measurements were done on Uppmax on the Snowy clusters, using the timings printed by the program. Timing measurements were done on 1, 4, 9, 16 and 25 cores since they are square numbers. Matrix sizes ranging from 3600x3600 to 10525x10525 were used. Speedup was calculated by dividing the time on the current number of cores by the time gotten on one core.

For the ideal speedup, both Amdahl's law described in equation 3 and Gustafson's law described in equation 4 was considered. p is the number of processing elements and f_p is the part of the code that is parallelized.

$$S \leq \frac{1}{(1 - f_p) + \frac{f_p}{p}} \tag{3}$$

$$S \leq p + (p - 1) * (1 - f_p) \tag{4}$$

Since only the parallel part of the program is timed as described in section 2.2, f_p is one. This means that both equation 3 and 4 can be simplified to $S \leq p$. The ideal speedup is therefore just the number of processing elements used.

Since a matrix-matrix multiplication has computational complexity $O(N^3)$, if the matrix size doubles, the computational complexity is eight times higher. Using a base matrix size of 3600 running on one core, in order to have the same number of calculations when running on 4 cores, the matrix size has to increase by $\sqrt[3]{4} \approx 1.587$ which corresponds to a matrix size of about 5714,6. The nearest number divisible by four is 5716. To run of 9, 16 and 25 cores, matrices of size 7488, 9072 and 10525 were used, using the same principle to have the same amount of work per core.

The memory usage was also of interest. The matrices stores floating point numbers at double precision which are 8 bytes each, so the memory to store one matrix of size N is $8 * N^2$. In the program, only four matrices are stored at the same time. The input matrices A and B are stored in the process with rank 0 and all processes store the local A and B submatrices, which combined are the same size as the full A and B input matrices, thus leading to the equivalent of four matrices being stored. The memory for the full A and B matrices is freed before the memory for the local result C matrices are allocated, and when the calculations are done, the local A, B and C matrices is stored in every processing element and the full C matrix is stored in the processing element with rank 0, which again is the equivalent of four matrices being stored. Thus, no more than the equivalent of four matrices are stored at one time. The memory for the timings are also added, which is the size of a double times the number of processors, this leads to the memory usage as being described in equation 5. By using the equation, it can be seen that if the program calculates the matrix-matrix multiplication on a matrix of size 3600 with 16 cores, the memory usage is 414 720 128 bytes ($\approx$ 414.7 MB)

$$Memory(Bytes) = 8 * 4 * N^2 + 8 * p \tag{5}$$

4 Results

cores	1	2	4	8	16
size 3600	34.7	9.9	6.6	4.8	2.24
size 7200	306.2	76.6	52.7	38.8	21.4
size 9072	546.2	131.1	104.8	77.4	Nan

Table 1: Times for different sized matrices dependent on amount of cores

cores	1	4	9	16	25
size 3600	1	3.51	5.23	7.26	15.48
size 7200	1	3.00	5.81	7.89	14.33
size 9072	1	4.17	5.21	7.05	Nan

Table 2: Speedup for different sized matrices dependent on amount of cores

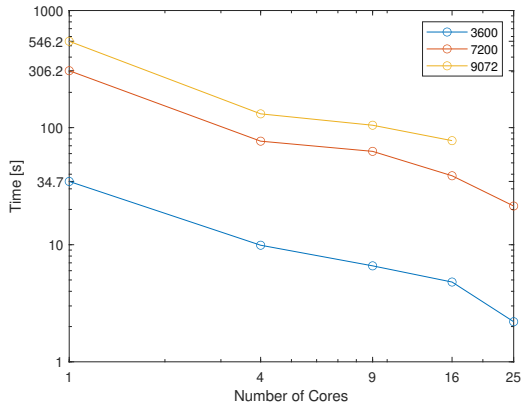

Figure 2: Time for different sizes of matrices dependent on cores used

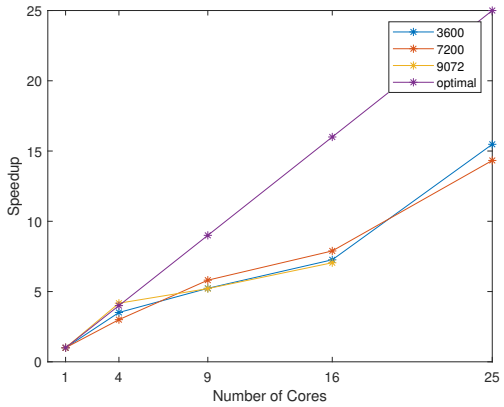

Figure 3: speedup for different sizes of matrices dependent on cores used

In figure 2 together with table 1 one can see how the time to solve the matrix-matrix multiplication is affected by parallelizing the work over different amount of cores. Figure 3 and table 2 instead show how much faster the problem finishes dependent on amount of cores used compared to the speed of only one core. The purple line in figure 3 indicates what the optimal speedup would be as described in section 3.

cores	1	4	9	16	25
matrix size	3600	5716	7488	9072	10525
Time [s]	34.7	37.7	59.9	77.5	68.3

Table 3: Time for weak scalability

Weak scalability is measured in figure 4 and table 3. This means that each core processes the same amount of work. Any core of the 25 parallel cores does the same amount of work as any core of the four parallel cores. Ideally, this should be constant, i.e. if the same amount of work is done per core, the time should be the same.

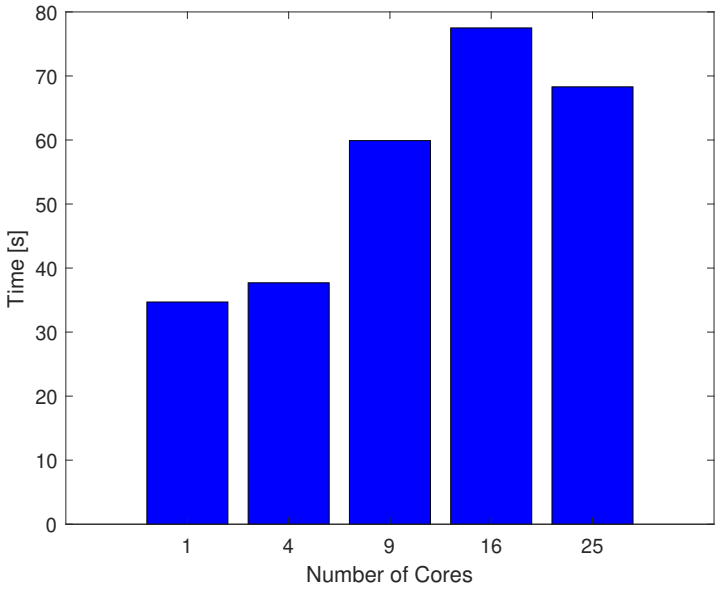

Figure 4: Time for weak scalability

5 Discussion

For the strong scalability, the speedup (table 2) from parallel cores follow the optimal path up to four cores where-after there is still speedup but at a slower rate. With four cores the problem should be solved four times as quickly, which is reflected in practice as seen in figure 3, but for 16 cores there should be a 16-fold speedup while in practice there is only a seven-fold speedup.

This is also reflected in the weak scalability figure (figure 4), where the slowest core among the four parallel cores works at the same speed as the single core and while nine or more cores work in parallel the cores seem to work slower even if each core has the same amount of work as for one and four parallel cores. The decreased speedup for more than four cores is likely a consequence from an increase in data-transfers between cores, where there is always a waiting time between sending and receiving the local matrix vectors between the cores and MPI_Sendrecv_replace is called more often if more cores are used as described in section 2.1. A way to work around this problem is to send the next local matrix vectors as temporary vectors while the current ones are being used for calculations. This way there is no need to wait for the sending and receiving to be completed between calculations, the new calculations can be started the moment the old ones are done, meanwhile the data for the next calculations are sent. This can be done with non blocking communication by using Isend and Irecieve from OpenMPI. The drawback with this is that more memory is used, since essentially two copies of the local A matrix and two copies of the local B matrix is stored, which is why the method using MPI_Sendrecv_replace was used.