Parallelization of Dense Matrix Multiplication

Victoria Sedig
Parallel and Distributed Programming
Uppsala University

27 May 2021

1 Introduction

1.1 Background

In this project a parallelized matrix-matrix multiplication program was developed using open MPI. The method used to do the parallelisation was the Cannon algorithm.

The Cannon algorithm divides the matrices into blocks partitions. The number of blocks are the number of processes that are going to be used. The calculations will be done parallely and when all calculations are done they will be sent to one processor to gather the results into a solution matrix.

2 Method

2.1 Parallelization

The matrix values are read into an array A and an array B in processor 0. These blocks are then partitioned with checkerboard partitioning and sent out to the rest of the processors. The processes are grouped into rows and column communicators using MPI_Split. This makes the sends easy to work with using $MPI\_Sendrecv\_replace$. First the initial moves of the canon algorithm are done. Then the full cannon movements are done. After all the multiplications are done the result from each processor is sent to process zero who will write the final result to an output file.

2.2 Advantages and Disadvantages

An advantage with using checkerboard partitioning is that both Cannons algorithm and Fox algorithm are well documented and uses checkerboard partitioning. A disadvantage is that the topology optimal for Cannons algorithm is a torus topology but Uppmax has a tree-leaves topology.

2.3 Verification of Correctness

To test the correctness of the code a reference program for sequential matrix-matrix multiplication was made. The output for the programs were compared

(when the same input file was used) with the diff command. This was done for a variation of matrix sizes and number of processors.

3 Experiments

3.1 Tests

All experiments were run on Uppmax Snowy with a prepared bash file. The calculations and plots were made in matlab. There were two experiments, one for strong scalability and one for weak scalability.

3.2 Memory usage Estimation

The code has to allocate space for matrix A (n^2), matrix B (n^2) and matrix C (n^2). After all the processors have sent in their part of the matrix C the results are not gathered in the correct order in matrix C. Therefor the results from C are put in to another matrix C_final (n^2) with the elements stored in correct order. This means there are 4 matrix allocations of size (n^2). This gives the memory usage estimation:

$$\epsilon = 8 * 4(n^2) \tag{1}$$

4 Results

4.1 Strong Scalability

In Table 1 and Figure 1 the result for the speedup is seen. The number of processors is put to values were the matrix size n is evenly divided by the square root of the number of processors.

Table 1: The speedup for different number of processors. The size of the matrix was constantly $n = 3600$.

#Cores	1	4	9	16	25	36
Time	44.9413	12.3071	10.7240	6.9668	3.2338	1.2762
Speedup	1.0000	3.6517	4.1907	6.4508	13.8973	35.2138

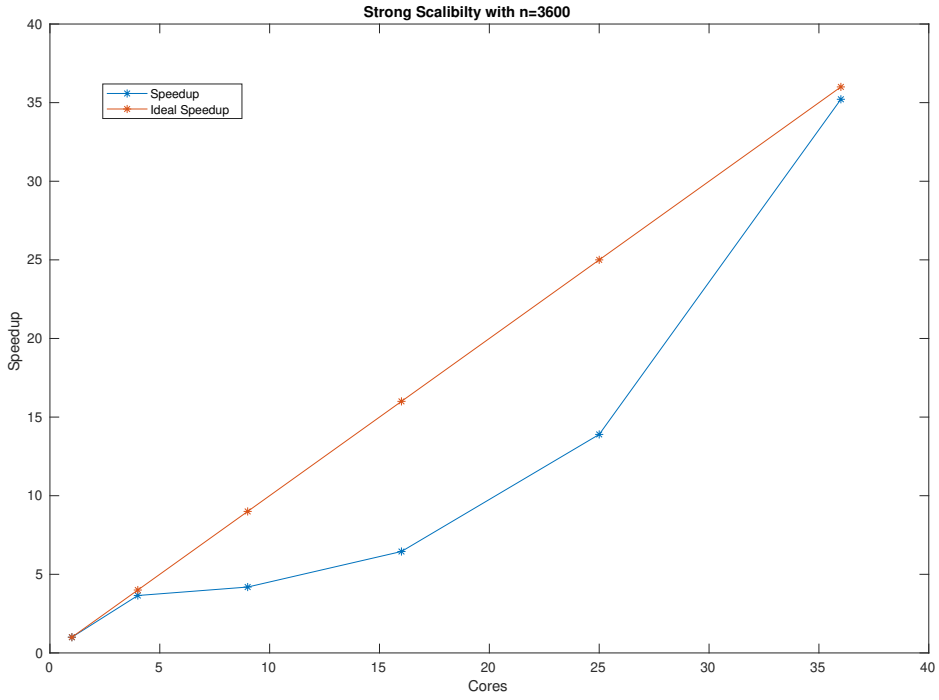

Figure 1: The speedup and the ideal speedup plotted against the number of cores used. The matrix size was kept constant at $n = 3600$.

4.2 Weak Scalability

Table 2: The weak scalability with increasing numbers of processors and matrix size n. Since the complexity for matrix multiplication is $O(n^3)$ the sizes for the matrix was chosen with the formula $n \approx n_1 p^{1/3}$ where p is the number of processors and n_1 is the matrix size used for one processor. .

#Cores	1	4	16	25
Matrix Size	3600	5716	9072	10525
Time	44.9413	49.1453	114.2198	114.7669

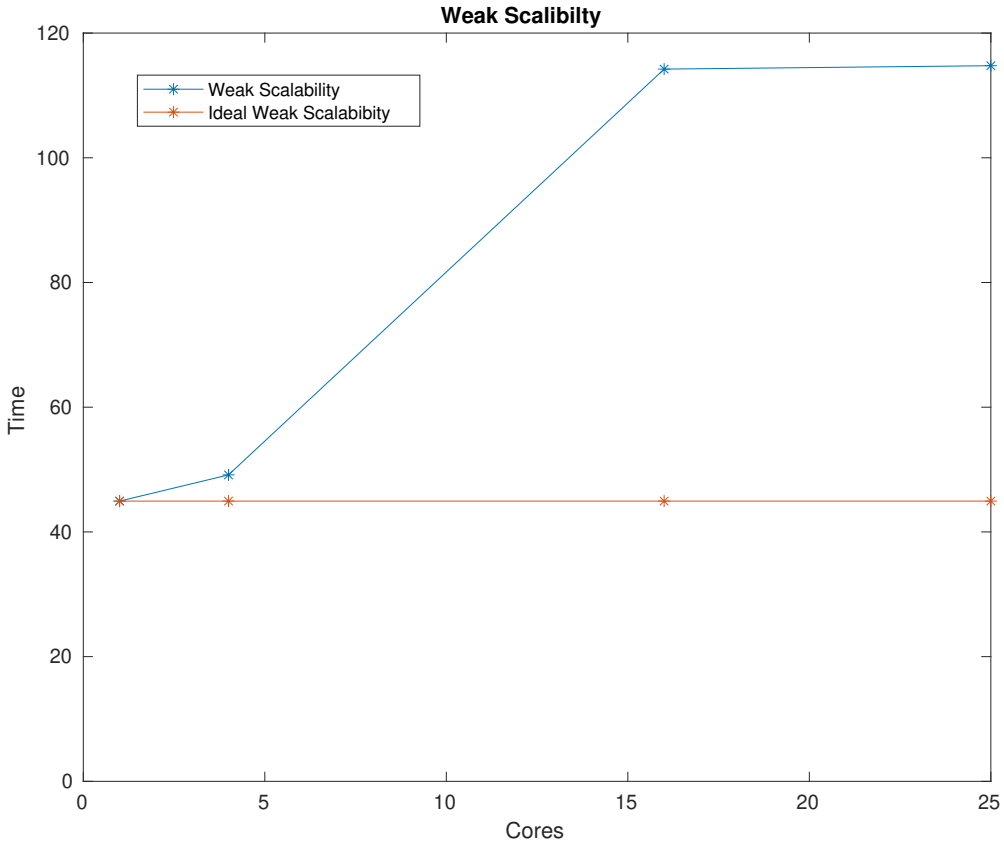

Figure 2: The execution time for the weak scalability plotted against the number of cores. The matrix size was increased with the number of cores to keep the work constant over the processors.

5 Discussion

In Table 1 and Figure 1 the speedup and the ideal speedup is shown. In Figure 1 the speedup follows the ideal speedup up to 4 cores. Between 4 and 25 cores the speedup does not follow the ideal speedup but at 36 cores it is almost ideal again. The reason to why the speedup increases for 36 cores may have to do with cache memory. For 36 cores each processor will have smaller matrices to allocate than for the cases when the number if cores are fewer. Maybe for 36 cores and $n = 3600$ the whole array for each processor fits in the cache.

In Table 2 and Figure 2 the weak scalability is shown. It is increasing up to 16 cores but after that it seems to stabilize as the result for 16 and 25 cores are similar. The reason for the increase may be because of the communication between processors. The more processors the more communication time will be needed as the groups have to do $sqrt(p)$ number of $MPI\_sendrecv\_replace$ for each matrix. For 16 and 25 processes it does not make a big difference as it would only go from 4 to 5 loops.