ASSIGNMENT 2:

Matrix-matrix multiplication

Parallel and Distributed programming, 5 hp

Erik Bjerned, Oskar Holmes & Mattias Persson

2023-04-17

UPPSALA
UNIVERSITET

Contents

1 Introduction

The task presented is multiplication of 2 square and dense matrices. The multiplication should take place in a distributed memory parallel computer environment, and be implemented using MPI in C.

2 Parallelization

2.1 Distribution method

We elected to use Cannon's algorithm. The algorithm necessitates the use of a checkerboard distribution of matrices since the algorithm partitions submatrices to processor in a checkerboard pattern. The fact that the algorithm utilises checkerboard type distribution does also provide other advantages. It can have a significant impact on the computational of the square matrices in computers where PE:s are distributed in a similar pattern. This is because the submatrix elements can be partitioned to PE:s in a corresponding position as the main matrix, resulting in the partition in close proximity communicating over shorter distances. The obvious disadvantage comes from the fact that non square matrices can not be partitioned by the algorithm as simply as for the cases of squares. This makes it less versatile. However it is quite memory efficient since it passes around the data in steps, as opposed to keeping local copies of data that is not being currently used.

The algorithm itself is quite simple, which makes the implementation quite short. Using MPI's Cartesian communication network it is simple to create a edge-wrapping grid of processors. To distribute the data, a vector datatype is created such that the sub-matrices can be scattered with `MPI_Scatterv` in a single call. Since the loaded data is row-major, it is not possible to use `MPI_Scatter` since the access pattern to the elements of the sub-matrices is quite complicated. To finially collect the data from the processors the created datatype used used once again together with `MPI_Gatherv` to the "inverse" of the previous commands.

2.2 Memory estimate

As for all distributed problem, memory is quite important to keep track of to enable the computation as otherwise it is easy to exceed the amount available. In this implementation there are a few large memory allocations done

1. Input matricies - $2N^2$
2. Local matricies - $3(N/\sqrt{P})^2 \cdot P$
3. Result matricies - N^2

This gives an approximate sum of number of elements, since they store as `float` it will be about $24N^2$ bytes as the total usage. However the peak will be lower since not all is needed at the same time, thus can be deallocated earlier or allocated later.

3 Performance tests

3.1 Method

The tests performed on this program are quite simple, as there is not really that much to test. Strong scaling was tested on a 3600×3600 matrix. However due to the nature of Cannon's

algorithm, it is not possible to get a detailed results, the configurations of cores and matrix sizes are quite limited. This is especially the case for the weak scaling as it is not always possible to exactly match the workload increase as with the number of cores.

3.2 Test Results

3.2.1 Strong scaling

Table 1: Execution time for different amount of PEs, perform on a 3600×3600 matrix.

PEs	Execution time(s)
1	16.250
4	4.2887
9	2.2176
16	1.4850
25	1.2627

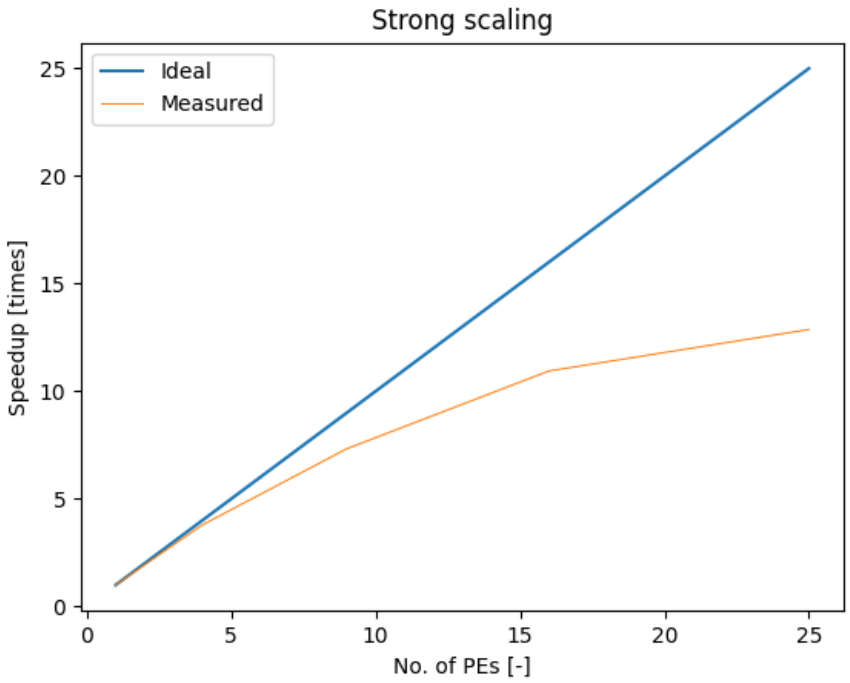

Figure 1: Strong scaling, speedup versus number of processors

3.2.2 Weak scaling

Table 2: Execution time for different amount of PEs

PEs	Matrix size	Execution time(s)
1	5716	64.2239
4	7488	36.2690
16	9072	20.7305

4 Discussion

Looking at the results of the tests of the strong scaling, it performs quite well at four cores, but after that decreases, an later on slows down significantly. It was expected that the results would not be ideal, as with a constant matrix size but increasing number of processors the amount of overhead would grow. This is because not only does the number of commutations between the processors have to increase, but multiplication of the sub-matrices are further divide and are not optimised for equally by the compiler. Although it was expected to be not ideal, it was not expected to be this bad, so the implementation has most likely some flaws.

As in regard to the weak scaling test the performance is not very good either, how bad it is is hard to say as it has quite low granularity. To give a better view of how this implementation scales, one should also improve the serial multiplication. Especially when handling such large sub-matrices as in the later cases of the weak scaling test the serial part can take considerable time. By using a cache blocking algorithm to solve it serially, the result would be more indicative of the parallel performance.

4.1 After revision

After revising the original code, significant differences are seen. The main reason why the previous version performed so bad was not because of any overhead or reasons related to the parallel and distributed implementation. Rather it was because of the serial algorithm used to calculate a PE's submatrix. As these matrices still can be quite large an efficient implementation is needed to fully utilise the parallelization. Before a naive implementation of matrix multiplication was used, and after the revision unnecessary memory accesses are reduced by taking advantage of loop invariant variables. Further, cache blocking is used to improve the algorithm, reducing the dependency of memory. Each submatrix is split into its own submatrices of size 60×60 to reduce cache misses.