UPPSALA
UNIVERSITET

Assignment 2

Parallel and distributed programming 5 credits 1TD070 62007

Group members:

Ibrohim Hamoud ibrohimmn1994@gmail.com

Uppsala

March 19, 2022

1 Introduction and Problem description

This work deals with parallel matrix multiplication using a checkerboard distribution of the matrices. In this application, the processors can be viewed as a grid, where small sub-matrices from the whole matrix can be assigned to different processors. Thus, in the case of four processors, the elements of 4×4 are assigned to a grid of 4×4 processors, hence the name checkerboard mapping.

1.1 Foxs' algorithm

Fox's algorithm is an algorithm that distributes the matrix using a checkerboard scheme like the above. The algorithm takes n steps for the multiplication of matrices of n dimensions. In the first step, each process multiplies element-wise the diagonal entry of A in its process row by its element of B. Next step, each process multiplies the element immediately to the right of the diagonal of A by the element of B directly beneath its own element of B. The stages that follow are a repeat of these steps.

Assuming a multiplication of two 3×3 matrices A and B as follows

$$\begin{vmatrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \\ A_7 & A_8 & A_9 \end{vmatrix} \times \begin{vmatrix} B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 \end{vmatrix}$$

In the first step, The leading elements(the diagonal elements) in each row are distributed on all row entries, that are A_1 for the first row, A_2 for the second row and A_3 for the third row. Thus, The matrix B and the new A matrix are multiplied element-wise.

$$\begin{vmatrix} A_1 & A_1 & A_1 \\ A_5 & A_5 & A_5 \\ A_9 & A_9 & A_9 \end{vmatrix} \times \begin{vmatrix} B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 \end{vmatrix} = \begin{vmatrix} B_1.A_1 & B_2.A_1 & B_3.A_1 \\ B_4.A_5 & B_5.A_5 & B_6.A_5 \\ B_7.A_9 & B_8.A_9 & B_9.A_9 \end{vmatrix}$$

For the next step, The elements in B-matrix are moved upward column-wise, and new leading elements in A-matrix are identified and distributed, that are A_2, A_6 and A_7. Thus, element-wise matrix-matrix multiplication is then applied with the results being added to the results of the previous step.

$$\begin{vmatrix} A_2 & A_2 & A_2 \\ A_6 & A_6 & A_6 \\ A_7 & A_7 & A_7 \end{vmatrix} \times \begin{vmatrix} B_4 & B_5 & B_6 \\ B_7 & B_8 & B_9 \\ B_1 & B_2 & B_3 \end{vmatrix} = \begin{vmatrix} B_4.A_2 + B_1.A_1 & B_5.A_2 + B_2.A_1 & B_6.A_2 + B_3.A_1 \\ B_7.A_6 + B_4.A_5 & B_8.A_6 + B_5.A_5 & B_9.A_6 + B_6.A_5 \\ B_1.A_7 + B_7.A_9 & B_2.A_7 + B_8.A_9 & B_3.A_7 + B_9.A_9 \end{vmatrix}$$

The same step is repeated for a final time, thus obtaining a full matrix-matrix multiplication.

$$\begin{vmatrix} A_3 & A_3 & A_3 \\ A_4 & A_4 & A_4 \\ A_8 & A_8 & A_8 \end{vmatrix} \times \begin{vmatrix} B_7 & B_8 & B_9 \\ B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 \end{vmatrix} = \begin{vmatrix} B_7.A_3 + B_4.A_2 + B_1.A_1 & B_8.A_3 + B_5.A_2 + B_2.A_1 & B_9.A_3 + B_6.A_2 + B_3.A_1 \\ B_1.A_4 + B_7.A_6 + B_4.A_5 & B_2.A_4 + B_8.A_6 + B_5.A_5 & B_3.A_4 + B_9.A_6 + B_6.A_5 \\ B_4.A_8 + B_1.A_7 + B_7.A_9 & B_5.A_8 + B_2.A_7 + B_8.A_9 & B_6.A_8 + B_3.A_7 + B_9.A_9 \end{vmatrix}$$

The description for the algorithm applied using openmpi is given for in Chart (1) below for 2×2 matrices [1][2][3].

Chart (1)

Assuming tow matrices A and B with 4 processors a,b,c and d.

(1) Create a communicator with Cartesian topology(CartComm)

```
| a b |
| c d |
```

-Create another two communicator, one for rows(RowComm) and one for columns(ColComm)

---

(2) Data distribution:

-For 4 for processors, the matrices A and B should be divide into 4 parts and distributed among the processors

```
|A1 A2|  |B1 B2|
|A3 A4|  |B2 B4|
```

Such that A1 and B1 reside in a, A2 and B2 reside in b and so on.

- Each processors will have three local matrices A-local, B-local and A-extra. Such that, A and B parts will be stored in B-local and A-extra, while the A-local is used for pivoting in Fox's algorithm.

- Apply MPI-Scatter in ColCom to distribute the row parts of A and B, such that processor a gets |A1 A2| and |B1 B2|, and the processor c gets |A3 A4| and |B3 B4|.

- Apply MPI-Scatter in RowComm to divide what is stored in the processors a and c on the processors that reside in their respective row. Thus, |A1 A2| that is stored in processor a will be divided such that processor a gets only A1 and processor b gets only A2. Similarly for the rest of the data.

---

(3) Start the Foxs-algorithm's iteration

1) Identify the leading processors a and c.

2) Apply MPI-Scatter to distribute the A part stored in A_extra in the leading processors on all A_local in the other processors that are in the same row. For example, A_extra = A1 in the processor a. Thus, after applying MPI_Scatter, A_local = A1 in the processor a and b.

3) Multiply A_local and B_local in each processor and sore the result in C_local using the BLAS function cblas_dgemm.

4) Pivot and B matrix using MPI_Sendrecv_replace between the processors in ColCom.

```
5) repeat the procedure by identifying the next leading processors in each row.
 ------------------------------------------------------------------------------
```

2 Results and discussion

According to chart (1) for the application of Fox's algorithm, the communication between the processors is performed using the MPI-scatter and MPI-Bcast function instead of Send/Recv functions. Thus, there is no need to establish a connection between the processors explicitly, where these MPI operations are taken care of implicitly. The readability and of program is improved considerably with a MPI-Scatter instead of multiple Send/Recv. Additionally, the program do not need any explicit synchronization with MPI-Barrier as would be the case with Send/Recv.On the other hand, the implementation is facilitated drastically looking at the large number of processors that might be deployed in the parallelization.

The choice of the communicator topology to be Cartesian stems from the nature of the problem and the structure of the data. Thus, For a matrix-related application, Cartesian topology gives advantage of mapping the processors in similar fashion to matrix-structure.

The function MPI-Sendrecv-replace is used for pivoting the elements of B-matrix. This function combines both buffered send and recv and executes them concurrently with the same buffer. This makes it easier to avoid deadlocks. This operation is useful for executing a shift operation across a chain of processors. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly in order to prevent cyclic dependencies that may lead to deadlock[4].

The results of the time measurements expressed in table 1, show an expected stagnation in the speedup for above 36 processors. This agrees with Amdahl's Law as the plot reaches a plateau.
Plot 2 shows a linear increase in time execution with the size of the problem. Although the number of processors was scaled up proportionally.this is expected behavior as the parallel part scales linearly with the number of resources. The last column in table 2 gives a rough estimation of the problem scaling.

Memory column in table 1 is given in bytes per processor. A matrix of 3600×3600 that is divided into n parts among n processors. Thus, each processor hold $3 \times 4 \times 3600 \times 3600/n$ of doubles which corresponds three matrices that are local A, A-extra B and C-matrices.

Table 1: Time measurements for different number of processors for matrix-matrix multiplication of size 3600×3600.

Number of processors	Time(BLAS)	Time(loops)	Memory
1	68.880521	49.457481	155520000
4	26.158118	26.529109	38880000.0
9	11.719567	11.494531	17280000.0
16	10.382563	12.160445	9720000.0
64	2.214064	2.033333	2430000.0
144	3.878983	5.345102	1080000.0

Table 2: Measurements of the execution time for different problem size, with the number of processors scaled proportionally.

Matrices' dimension	Number of processors	Time	matrix-dimension/number of processors
1800	1	6.042439	$\times 1$
3600	4	26.529109	$\times 4$
7200	16	81.982405	$\times 16$
18000	100	240.494036	$\times 100$

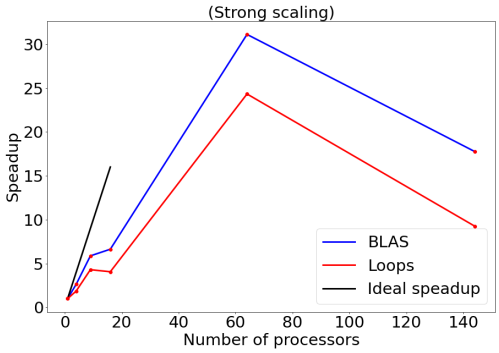

Figure 1: Plot of the speedup as a function of the number of the processors in blue, and the ideal speedup for the line in orange.

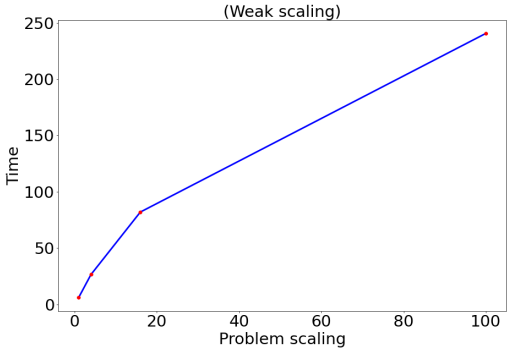

Figure 2: Plot of the execution time as a function of the problem scale.

References

[1] "Foxs' algorithm lecture slides."

[2] "Foxs' algorithm lecture slides."

[3] P. Pacheco, *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[4] "Open mpi-documentation."