Compulsory Assignment

Dense matrix-matrix multiplication

Course '*Parallel and Distributed Programming*'
Division of Scientific Computing, Department of Information Technology
Uppsala University

1 Problem setting

Let two square dense matrices be given, $A = \{a_{i,j}\}$ and $B = \{b_{i,j}\}$, $A, B \in \mathbf{R^{n \times n}}$, $i, j = 1, 2, \cdots, n$. Let $C = AB$. As is well known, $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}, \quad i, j = 1, 2, \cdots, n.$
The task is to:

1. design a matrix-matrix multiplication procedure for performing the multiplication in a distributed memory parallel computer environment,
2. implement the algorithm using C and MPI, and
3. evaluate the performance of your implementation.

You are encouraged to work in teams. However, all topics in the assignment should be covered by each student.

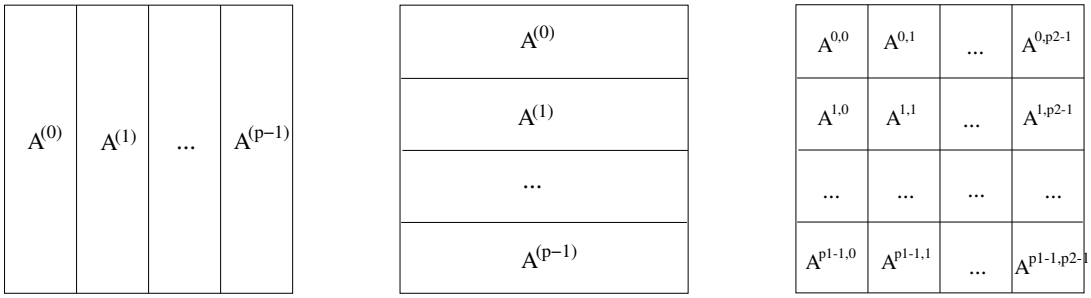
2 Partitioning strategies

Since your program is targeting a distributed memory architecture, you need to partition the matrices A, B and C. Choose the partitioning strategy that you find most suitable. Some possible partitioning strategies are shown in Figure 1 and presented below. It is up to you whether all matrices are partitioned the same way or not. Note, however, that you should motivate the particular data distribution you choose and how it could influence the performance and memory requirements. Note also that both matrices must be distributed. It is not acceptable to replicate one of the matrices on all the processing elements.

Colum-wise (1D-type) partitioning

The matrices are partitioned in blocks, containing $m = n/p$ columns (Figure 1(a)). (We assume that p is a divisor of n, thus, $n = mp$.) If all matrices are partitioned in the same way, the blocks $A^{(s)}, B^{(s)}, C^{(s)}$, each of size $n \times m$, are local for processor P_s, $s = 0, 1, \cdots, p-1$. The data distribution imposes additional structure in each block-column $A_i^{(s)}, B_i^{(s)}, C_i^{(s)}$, $i = 0, 1, \cdots, p-1$, each of size $m \times m$. A block $C_i^{(s)}$ is computed as follows

$$C_i^{(s)} = \sum_{j=0}^{p-1} A_i^{(j)} B_j^{(s)}, \; i = 0, 1, \cdots, p-1.$$



(a) Colum-wise partitioning (b) Row-wise partitioning (c) Checkerboard part.

Figure 1:

The blocks $B_j^{(s)}$ are local to process P_s. From the blocks $A_i^{(j)}$ only $A_i^{(s)}$ is local, the others have to be sent to P_s.

Row-wise (1D-type) partitioning

It is very similar to the column-wise partitioning (Figure 1(b)).

Checkerboard (2D-type) partitioning

This partitioning strategy is depicted in Figure 1(c). The processing elements are assumed to form a 2D Cartesian grid where $p = p1 \times p2$. It is also assumed that n is divisible by both $p1$ and $p2$, namely, $r1 = n/p1$ and $r2 = n/p2$, where the size of the blocks is $r1 \times r2$ and $r2 \times r1$ respectively.
You are free to use existing algorithms, not only the ones considered during the lectures, provided that you give the reference to the source you have used.

3 Implementation

Your program is supposed to take two arguments. The first argument is the name of a file containing the matrices A and B (the input file). The second argument is the name of the file to which the program will write the result, that is matrix C (the output file).
The input file will contain $2n^2 + 1$ numbers separated by white spaces. The first number is n (the matrix size). This is supposed to be an integer. The subsequent n^2 numbers are the elements of A, stored in row-major order. The last n^2 elements are the elements of B, also stored in row-major order. The output file shall contain the elements of C stored in ASCII format, in row-major order, and nothing else. Just as in the input file, the elements shall be separated by white spaces. All matrix elements are floating-point numbers and should be stored with six (6) decimal places.
Your implementation must work for matrices of different sizes, but you may make the assumption that n is divisible by the number of processes p if you use row-wise or column-wise partitioning. Likewise, you may assume that $\sqrt{p}$ is an integer, by which n is divisible, if you use checkerboard partitioning.

Measure the time for multiplication, including computations and communications, but not I/O, in seconds. The number of seconds (and nothing else, not even 'seconds' or an abbreviation thereof) shall be written to `stdout` when the multiplication is done.
Note that your code is supposed to be reasonably well structured and documented.

4 Numerical experiments

Your numerical experiments shall be run on UPPMAX (Snowy). Use the batch system for every execution of your program! You are free to use as many processors (cores) as you decide, but maximum two nodes. Also, be careful with allocating the right amount of memory, depending on the computing resources you require. Note that the maximum available memory per core on Snowy is 16 GB (see `https://www.uppmax.uu.se/resources/systems/the-snowy-cluster/`). Further, pay attention whether you are using the resource you ask for up to its full capacity.
Perform experiments to demonstrate the strong and weak scalability of your implementation. For the strong scalability experiments, it is enough to use one matrix size. When planning the scalability tests, note, that the computational complexity of dense matrix-matrix multiplication of matrices of size n is $O(n^3)$. Thus, when you double the matrix size, the arithmetic work becomes eight times more. Estimate (measure) the memory usage for your algorithm (in bytes).
If you do not want do write your own input files, you may use the files stored in the directory `/proj/uppmax2023-2-13/nobackup/matmul_indata` on UPPMAX. Larger files are intended for performance experiments, while while smaller files can be used for functionality testing. Files named `inputN.txt` contain input matrices (A and B) of size $N \times N$ on the format described in Section 3. Files named `outputN_ref.txt` contain the expected output when using `inputN.txt` as input and are intended to be used as a reference solution on development testing.
Problem sizes to test with your own matrices: The sizes of the matrices should be chosen large enough to utilize the resources you use. For the scalability tests it is suggested you use matrices of size 3600×3600 and (reasonably) larger.
Note that you have a limited disc quota at UPPMAX! Therefore, it is recommended that you **do not copy** the larger input files to your home directory, but read them from the project directory (where they are stored now). Moreover, it is a good idea to remove large output files as soon as possible. You may also skip the printing to files during the numerical experiments. However, note that the code that you hand in must follow the specifications in Section 3.

5 Report

You are supposed to write a report on your results. The report can be written in Swedish or English, and should contain (at least) the following parts:

1. a brief description of the original problem;
2. a description of your implementation - describe and motivate the partitioning strategy you have chosen, its advantages (and possible disadvantages);
3. a description of your numerical experiments;
4. the results from the numerical experiments:
   - execution times should be presented in tables,
   - the same holds for speedup values,
   - also, plot speedup together with the ideal speedup in the speedup plot;
   - the plots can be drawn using MATLAB, or any other tool that you prefer;
   - a discussion of the results - do they follow your expectations? Why/why not?

Note: The weak scalability results (the timing) should be presented in a table. Speedup in this case is not relevant.

6 File to be submitted

The file that you upload in Studium must be a compressed tar file named `A2.tar.gz`. It shall contain a directory named `A2`, with the following files inside:

- A PDF file named `A2_Report.pdf` containing you report.
- Your code, following the specifications listed in Section 3. The code shall compile on UPPMAX without warnings.
- A Makefile that does the following:
  1. Builds a binary named `matmul` from your code when the command `make` is invoked. This must work on the Linux system as well as on UPPMAX!
  2. Removes all binary files and object files (if any) when the command `make clean` is invoked.

Since your code will be tested automatically, it is very important that you follow the instructions regarding the implementation and the structure of the tar file! Failure to do so will result in immediate rejection of the assignment.
Note: performance tests on tiny matrices and many processes indicate that you have missed important parts of the course and the assignment will be rejected even if the code works correctly.

7 Self testing

You will be provided some scripts to self-test the correctness of your code. The details are included in a separate instruction file, named `Self_test_A2.pdf`.

Important note: When you upload a first revised version of your work, change the name of the file as `A2-R1.tar.gz` and if a second revision is allowed - `A2-R2.tar.gz`.
The file `A2.tar.gz` must be submitted as stated in Studium. As Studium allows for submissions after the deadline, first versions of assignments, submitted with a delay will be considered with a delay. Assignments that do not meet the requirements are returned. For submitting a final version of the assignment check the deadlines in Studium. Assignments submitted after this date will not be considered and approved.

Precaution - reminder

Use the UPPMAX resources with care! We are given 2000 core/hours for the whole course.

Success!

Maya, Marina, Tobias

Any comments on the assignment will be highly appreciated and will be considered for further improvements. Thank you!