Parallel and Distributed Programming
Uppsala University – Spring 2020
Report for Assignment 2: Dense matrix multiplication

Li Ju, Shuyi Qin

7th June 2020

I Introduction

1 Problem Definition

In mathematics, matrix multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The result matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix.

If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix and element at row i and column j in $\mathbf{A}$ and $\mathbf{B}$ are noted as a_{ij} and b_{ij}, the matrix product $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix such that $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$ for $i = 1, \cdots, m$ and $j = 1, \cdots, p$. [1]

In this assignment, two square dense matrices are given, $\mathbf{A} = \{a_{ij}\}$ and $\mathbf{B} = \{b_{ij}\}$, $A, B \in \mathbf{R}^{n \times n}$, $i, j = 1, 2, \cdots, n$. The program is able to performing the matrix multiplication in a distributed memory parallel computer environment using C and MPI.

2 Error Control

In this program, all the numbers in matrices are represented with floating-point format in C and results are stored with 6 decimal places.

3 Computer details

For assignment 2, all tests are conducted on UPPMAX system Rackham. The detailed information of Rackham cluster is listed below:

a) Rackham comprises 9720 cores in the form of 486 nodes with two 10-core Intel Xeon V4 CPU's each. There are "fat" nodes,4 with 1TB memory and 32 with 256 GB memory, with the rest having 128 GB.

b) Rackham's storage system (named Crex) uses the Lustre file system and provides 6.6PB of storage.

c) The interconnect is Infiniband FDR which supports a theoretical bandwidth of 56Gb/s and a latency of 0.7 microseconds.

d) Each compute node consists of two (2) Intel Xeon E5 2630 v4 at 2.20 GHz/core (10 cores, 20 threads, 25 MB LLC, and a bandwidth of 68.3 GB/s) [2]. Each compute node has either 8x16384MB (128GB) or 16x16384MB (256GB) or 16x65536MB (1TB) of ECC 2400MHz DIMM DRAM memory. For local storage each compute node has a 2TB disk at 7200RPM.

e) Each system node also has two CPUs identical to a compute node. System nodes has 16x16384 (256GB) of ECC 2400MHz DIMM DRAM. For local storage each system node contains 4x600GB 10 000 RPM SAS disks for a total of 2.4 TB.

II Solution

1 Algorithm design

To compute the multiplication parallel, Fox's algorithm is applied, which is shown as Algorithm: Fox multiplication.

Algorithm Fox Multiplication

Require: Partitioning matrix A and B into blocks: submatrices at row i and column j are noted as A_{ij} or B_{ij}.
Broadcast A_{ii} to all cores which are at the same row of A_{ii}
for step=1; step < number of dimensions; step++ **do**
Multiply received submatrix of A with local submatrix of B and add the result to local result matrix
Send local submatrix of B to upper core
Broadcast $A_{i,(i+step) \mod numberofdimensionofcore-grid}$ to all cores which are at the same row
end for
Collect all local result matrices and store it

In the process, serval kinds of communication are used, including Send (Recv), Isend (Irecv), Bcast, and Barrier.

1. Considering the limited size of MPI system buffer, in order to avoid deadlock due to some large messages, nonblocking send function Isend is used, together with MPI_Wait. But the program cannot be executed to the next step if the communication haven't finished, so there is no need to overlap the communication with computations. The other processes only focus on receiving data. Therefore, Recv is used here to simplified the data communication.

2. In matrix multiplication, the linear process rank number cannot reflect the actual communication mode between the processes. At the same time, considering the load balancing and communication cost, a two-dimensional topology is established. The process is mapped to a two-dimensional grid using MPI_Cart, The process topology is managed by groups using MPI_Comm_split. The use of process topology can facilitate and simplify the code. On the other hand, it can help the processes map to the processors better and organize the flow of communication, so as to obtain better communication performance.

3. For communication with neighbors to exchange adjacent values, MPI_Bcast is used to pass matrix A to all cores in the same row. Isend is used to send matrix B, which is to configure all transmissions first and finally wait for them all finish by another MPI_Wait function. In this case, two advantages arise comparing with Ssend, 1). It is not required to wait for neighbor data transmission finish before one process send its own data, which will make the whole communication serial and costly, correspondingly; 2). Each process are in equal position in this scenario and it is easily to get deadlocked. To avoid deadlock, one must start send first and others wait before sending, which requires more work for coding.

4. MPI_Barrier is also used to make sure each process have finished its own work before exchange data. This is more efficient and easy comparing with other signals in MPI.

2 Performance experiments

For performance evaluation, time analysis is introduced. In our performance test, the counted time included the time cost on adjacent message passing, calculation for sub-tasks and barrier waiting for each process (data reading and writing, task assigning and collecting not included). The maximum time cost of all processes is printed and used to represent the time cost of the parallelized program. All the tests are with optimization flag O3 of GCC

In case of strong scalability, the number of processors is increased while the problem size remains constant. This results in a reduced workload per processor. In our experiments, multiple cases are tested, the problem size is fixed by input matrix whose dimension is 3600. Each test case is done 5 times and the minimum time cost is used to present the time cost of the test case. Numbers of processors are chosen as 1, 4, 9, and 16.

In case of weak scalability, both the number of processors and the problem size are increased. This results in a constant workload per processor. In our experiments, we fix the workload for each process as a matrix has around 3600 dimension. Same as strong scalability tests, each test case is also done 5 times and the minimum time cost is used to present the time cost of the test

case. The detailed test cases are listed in performance analysis part.

III Performance analysis

1 Strong scalability analysis

The results of strong scalability evaluation are listed as Table(1):

dimension	Num of Processors	Time cost/s	Speedup
3600	1	54.119592	1
3600	4	14.431511	3.7501
3600	9	6.410852	8.4419
3600	16	3.838544	14.0990

Table 1: Strong scalability results

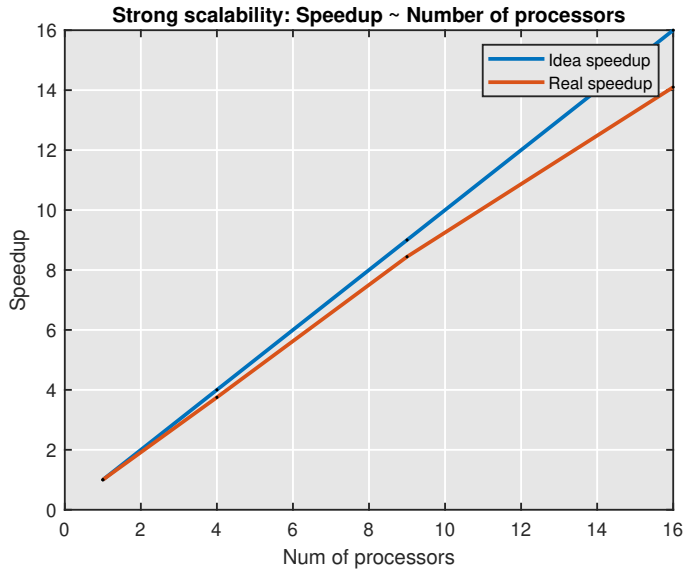

Figure 1: Strong scalability: speedup versus number of processors

The figure of speedup versus number of processors is plotted as Figure (1) shows. When the amount of task is fixed, as the number of processors increases, the speedup of the program increase correspondingly.

The time cost evaluated here contains three parts: message passing, calculation and barrier waiting. Strong scaling pointed out that for a fixed problem, the speedup is limited by the fraction of the serial part of the software that is not amenable to parallelization.[2]
In order to check which part is the most costly one, another time evaluation for three parts are done and the results are shown as Table(2). As the number of processors increases, the time

cost on calculation reduces significantly.

When there are 16 processors, the program does not reach the limit of speedup obviously.

Num of Processors	Message passing/s	Calculation/s	Barrier waiting/s
1	-	54.119592	-
4	0.020447	14.070332	0.156709
9	0.018162	6.151764	0.240926
16	0.025508	3.728589	0.084447

Table 2: Strong scalability: time evaluation of different parts

However, compare with the serial implementation, the parallelized program spends time on message passing and barrier waiting. So when the number of processors keep increasing, the efficient of parallelization would decrease.

2 Weak scalability analysis

The computational complexity of dense matrix-matrix multiplication of matrices of size n is $O(n^3)$. Thus, when the matrix size doubled, the arithmetic work becomes eight times more. So we choose the metric whose dimension is 7488 and calculate it on 9 cores.
The results of weak scalability evaluation are listed as Table(3):

Core num	dimension	dimension per core	Time cost/s	Expected cost/s	Efficiency
1	3600	4.6656 e10	54.119592	-	1
9	7488	4.6650 e10	58.900222	54.119592	0.9188

Table 3: Weak scalability: results

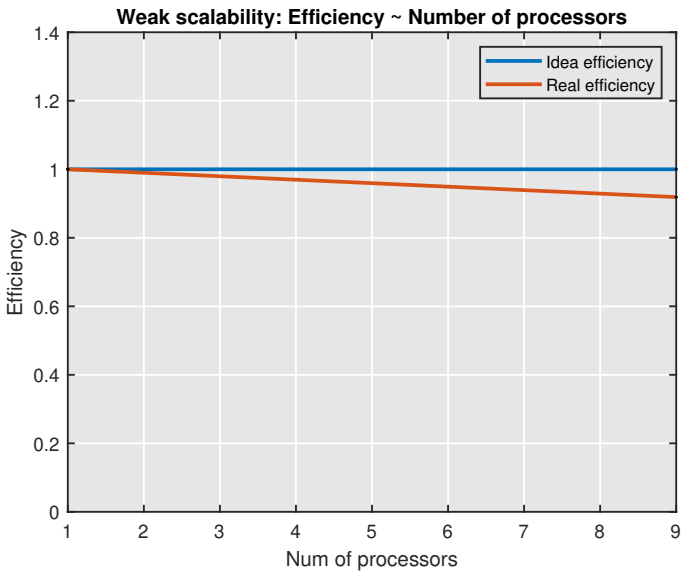

Figure 2: Weak scalability: efficiency versus number of cores

The data is illustrated graphically by Figure(2). Ideally, when the workload of each core is fixed, the efficiency should always be 1 when the number of core increases. However here in this case, the efficiency decreases slightly. The reason of this can be explained by what we found in strong scalability part: when the number of processors increase, the time cost on inter-core communication and barrier waiting increases, which make the efficiency decrease.

IV Conclusion

In this assignment, the parallelized matrix multiplication is implemented with MPI. Performance analysis are conducted, and based on the performance results, more detailed tests and discussions are made. The advantage of the parallel algorithm reflects when the matrix has a large dimension.

References

[1] Wikipedia, Matrix multiplication, `https://en.wikipedia.org/wiki/Matrix_multiplication`.

[2] Xin Li,Scalability: strong and weak scaling, `https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/`.