Uppsala University

Parallel & Distributed Programming

1TD070

Assignment 2:

Dense Matrix Multiplication

Students:
Omar Malik
Junjie Chu
Panagiotis Stefanos Aslanis

Teachers:
Maya Neytcheva
Jarmo Rantakokko

June 10, 2021

1 Introduction

In this assignment we are tasked with the multiplication of two dense matrices A, B in a parallel and distributed programming environment(i.e., $A * B = C$); to achieve that we utilize MPI to slice the matrices in a way that each PROC (i.e, processor) is assigned an equal amount of values/elements to multiply the MPI communicator to gather the results and write C to an out file.

2 Specifications

The program was run on the Rackham cluster at Uppsala University's UPPMAX supercomputing center. Up to four nodes were used, where each node contains two 10-core Intel® Xeon® V4, 64-bit CPUs, each of which runs at 2.2 GHz. The machines use the CentOS Linux 7 operating system. Regarding the C programming language, the C99 standard was used with the GCC compiler release 4.8.5, and the parallelization was done using the OpenMPI distributed-memory library.

3 Description of the implementation

3.1 Algorithm

We would like to choose a row-wise algorithm. Compared with the serial algorithm using only one process, the row-wise algorithm allow us to use multiple processes which could speed up the computation.
In our algorithm, the input matrices are A and B and the result matrix is C. Their size is $N * N$. The number of processes is p. And the chunk is N/p.
Firstly, we consider to split the input matrix A and the output matrix C. Thus, each process will have a localA matrix(chunk rows and N columns) as well as a localC matrix(chunk rows and N columns). The input matrix B will be stored in all processes(i.e. broadcast). In each process, the localC matrix could be calculated locally. This algorithm is very fast but it is obvious that we waste a lot of memory to store matrix B. This is a challenge to the hardware.
Then, we consider to split the input matrix B as well. The matrix B will be divided by column. Each process will have a localB matrix(chunk columns and N rows). The localA matrix and localC matrix are the same as previous algorithm. After one computation of localA matrix and localB matrix, we get a result matrix(chunk rows and chunk columns). We name one computation as one circle. The result matrix is part of the localC matrix. The localC matrix includes such p result matrices. So we need to compute for p times. That is to say we have p circles in total. After one computation completes, the localB matrix need to be exchange between the processes. The rules are set as: the rank i process will send data to the rank $i-1$ process and receive data from rank $i+1$ process.
Compared with the first parallel algorithm, we save the memory which is used to store B. But we waste some time on communicating with each other. We need a trade-off between time and space.

3.2 Implementation of MPI code

The matrices are stored in the input file by rows first. So we choose to read the matrix A and C by row first. We need to split the matrix B by column. To realize that, we have 3 methods in total. In the first method, we will stored the matrix B by column first. In the 2nd and 3rd method, we will try to use MPI_Vector to implement that.

In the first version, we use fscanf to read the input matrix B by row, then store them by column, which cause that we fail to visit the matrix B's memory address continuously. This is not very efficient.
In the 2nd and 3rd version, we use MPI_Vector to reach our goal. The difference between them is that, we use 1 vector type in the 2nd version and 2 vector types in the 3rd version.
The relevant part of the code of 2nd version is as follows.

```
if (rank == 0){
  //create new global datatype for root 0
  //the coltemp represents 1 col in global B and the stride is N.
  MPI_Type_vector(N, 1, N, MPI_DOUBLE, &gcoltemp);//This is 1 column.
    MPI_Type_commit(&gcoltemp);
    MPI_Type_create_resized(gcoltemp, 0, 1*sizeof(double), &gcol);//This is to
  specify the position of the next column
    MPI_Type_commit(&gcol);
}
MPI_Scatter(B ,chunk,gcol,localB_r,chunk*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

The MPI_Type_vector has 3 important variables. In the vector, it has N blocks in total. Each block has only 1 element. And the stride of the 2 blocks is N. This is because the original matrix is a row-wise matrix, so the distance between two adjacent elements in a column is N(the number of elements in a row). One vector represents one column. Since we would like to send more than 1 column, so we have to specify the start of each vector. The start of each vector is not at the end the previous vector. The vector starts at the next address of the first block of the previous vector. Thus we use MPI_Type_create_resized to specify the size of the vector.
The relevant part of the code of 3rd version is as follows.

```
if (rank == 0){
  //create new global datatype for root 0
  //the coltemp represents 1 col in global B and the stride is N.
  MPI_Type_vector(N, 1, N, MPI_DOUBLE, &gcoltemp);//This is 1 column.
    MPI_Type_commit(&gcoltemp);
    MPI_Type_create_resized(gcoltemp, 0, 1*sizeof(double), &gcol);//This is to
  specify the position of the next column
    MPI_Type_commit(&gcol);
}
//create new local datatype for all processes
//the rowtemp represents 1 row in local B and the stride is chunk.
MPI_Type_vector(N, 1, chunk, MPI_DOUBLE, &lrowtemp);//This is 1 row.
  MPI_Type_commit(&lrowtemp);
  MPI_Type_create_resized(lrowtemp, 0, 1*sizeof(double), &lrow);//This is to
  specify the position of the next column
  MPI_Type_commit(&lrow);
  MPI_Scatter(B ,chunk,gcol,localB_r,chunk,lrow,0,MPI_COMM_WORLD);
```

In the third version, we use an extra vector type to receive the data from MPI_Scatter. Very similar to that in 2nd version, most of the variables are the same. But since the local B in each process is row-wise and has *chunk* columns and N rows(*chunk* elements in a row), the third varible is changed to *chunk*.
The difference between version 2 and 3 is that the local B matrix in version 2 is column-wise but in version 3 is row-wise.
If we take the 3rd version, this is a very classical row-wise matrix multiplication in each process and we need extra optimization of the order of the loops.
But if we take the 2nd version, the local A and local C are row-wise but local B is column-wise. This is a very smart algorithm because the i, j, k order is very efficient in the way and there is no need to do extra optimization.

```
//Multiplication
for (int i=0;i<chunk;i++){
  for (int j=0;j<chunk;j++){
    double tempc = 0;
    for (int k=0;k<N;k++){
      tempc=tempc + localA[i*N+k] * localB_r[j*N+k];
    }
    localC[i*N+(indexblock*chunk+j)]=tempc;
  }
}
```

When k increases by 1, the address of local A and local B both move one step which means we could visit the matrices continuously.
After comparing the version 2 and version 3, we decided to use version 2 as our final version. This version is smarter and faster. And it is very stable, works well on the University clusters, local linux laptop and VM. But version 3 fails in some virtual machines in our experiments.
The destination and the source of the data are computed by:

```
// Current rank process send data to rank+1 process; receive data from rank-1
  process
int SendTo = (rank + 1) % size;
int RecvFrom = (rank - 1 + size) % size;
```

The implementation of one circle is as follows:

```
for (int circle = 0; circle < size; circle++){
  //One rank's localC has several blocks, determine which block we are computing
  int indexblock = (rank-circle+size)%size;
  //Multiplication
  for (int i=0;i<chunk;i++){
    for (int j=0;j<chunk;j++){
      double tempc = 0;
      for (int k=0;k<N;k++){
        tempc=tempc + localA[i*N+k] * localB_r[j*N+k];
      }
      localC[i*N+(indexblock*chunk+j)]=tempc;
    }
  }
  //Copy, send and receive localB
  for (int i=0;i<N*chunk;i++){
    localB_s[i] = localB_r[i];
  }
  MPI_Sendrecv(localB_s, chunk*N, MPI_DOUBLE, SendTo, circle, localB_r, chunk*N,
  MPI_DOUBLE, RecvFrom, circle, MPI_COMM_WORLD, &status);

}
```

In the part of code, we use *localB_s* to store the data we want to send and use *localB_r* to store the data the process receives so that the data used in communication will not be polluted.
The result matrices are collected via MPI_Gather.

```
//After all calculation complete, gather
MPI_Barrier(MPI_COMM_WORLD);
MPI_Gather(localC,N*chunk,MPI_DOUBLE,C,N*chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

At last, the root rank will write the output matrix into the output file.

4 Numerical Experiments

The program was verified by comparing one of the saved output matrices with a corresponding reference matrix that was computed using MATLAB. This was done by computing the L_2 norm of the element-wise difference between the output and reference matrices, which was on the order of 10^{-5}. The scalability of the program's parallel implementation was then verified in two ways:

- By measuring its runtime against the number of processors p — also known as the communicators' size — for a fixed matrix size of $N \times N$: this is known as the **strong scalability** of the program. For this scalability, the number of communicators were chosen so that they perfectly divide the chosen number of rows N. In this case, N was chosen to be 7200, and so the number of communicators that divide this value would be the factors of 7200, from 1 up to and including 24 communicators.
- By also measuring its runtime against a varying communicator size p and matrix size $N \times N$, keeping the number of operations-per-communicator — also known as the *workload* — constant for all $\{p, N\}$ pairs: this is known as the **weak scalability** of the program. To this end, the total number of operations to multiply two dense matrices would be around N^3 (the time complexity is $\mathcal{O}(N^3)$), and so the number of operations per communicator would be $N^3/p = c$, where we would like to keep c constant. For two different $\{p, N\}$ pairs, labelled $\{p_0, N_0\}$ and $\{p_k, N_k\}$, we then have $N_0^3/p_0 = N_k^3/p_k$, so $p_k = (N_k/N_0)^3 p_0$. We chose to start with $p_0 = 1$ and $N_0 = 3600$. Additionally, the following values of N_k were used: 5716, 7488, and 9072, which correspond to $p_k = 4$, 9, and 16. The number of operations per communicator would then be approximately $N_0^3/p_0 = 4.66 \times 10^{10}$ operations.

5 Results & Discussion

Table 1: *Strong scalability.* Total runtime $T(N, p)$ vs. the communicators' size p. Here $N = 7200$. Note that the chosen communicators' sizes are all factors of the chosen N, so that the rows would be perfectly divided amongst the communicators. Here the speedup was calculated as $T(N, 1)/T(N, p)$.

p	Runtime (s)	Speedup
1	411.392384	1.000000
2	208.953059	1.968827
3	155.526403	2.645161
4	104.244406	3.946422
5	99.954465	4.115798
6	75.384604	5.457247
8	59.250695	6.943250
9	57.986206	7.094659
10	49.636556	8.288093
12	43.185257	9.526223
15	39.480216	10.420216
18	32.078411	12.824587
20	29.914310	13.752361
24	25.109648	16.383837

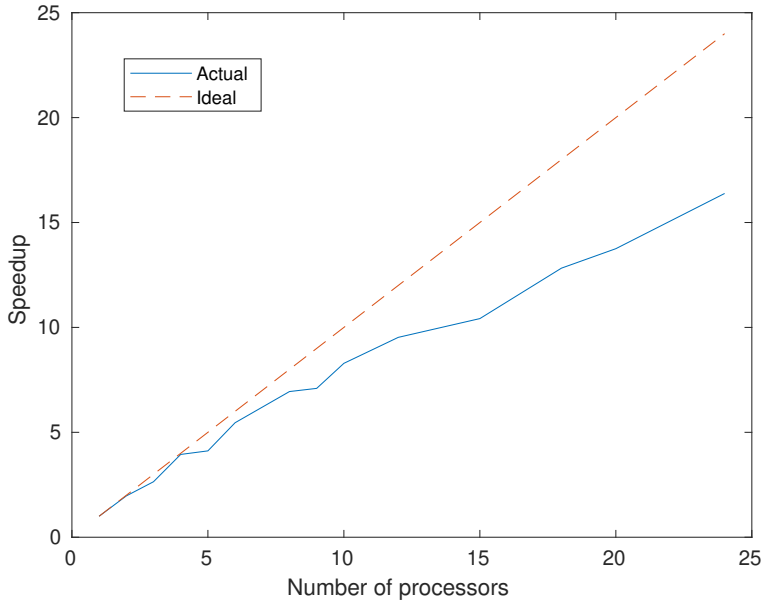

Figure 1: *Strong scalability.* Speedup with respect to communicators' size p, calculated as $T(N,1)/T(N,p)$, where T is the implementation's runtime. The dashed line shows the ideal speedup. Here $N = 7200$.

Table 1 along with Figure 1 show the runtime and speedup of our parallel dense matrix multiplication as a function of the communicators' size; these show the strong scalability of the program. The actual speedup deviates significantly from the ideal speedup for greater than 10 communicators.

Additionally, Table 2 and Figures 2-3 show the runtimes and efficiency of our implementation for a constant workload; these show the weak scalability of the program.

Table 2: *Weak scalability.* Total runtime $T(N,p)$ vs. the communicators' size p and matrix size $N \times N$. Here p and N were chosen so that each communicator receives approximately the same workload for different values of N. In this case, the workload is approximately 4.66×10^{10} operations per communicator. The efficiency here was calculated as $T(N,1)/T(N,p)$.

p	N	Runtime (s)	Efficiency
1	3600	51.172037	1.000000
4	5716	54.191330	0.944285
9	7488	65.018304	0.787040
16	9072	73.745387	0.693902

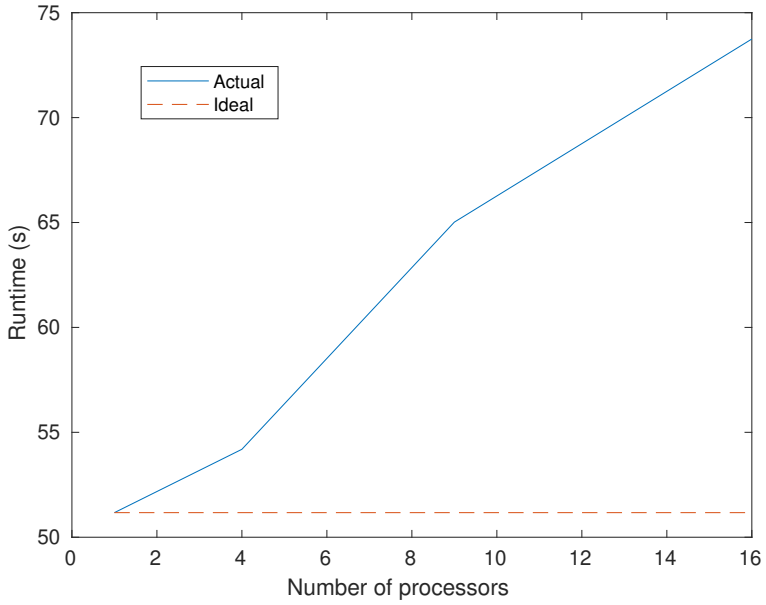

Figure 2: *Weak scalability.* Total runtime vs. the communicators' size p and matrix size $N \times N$.

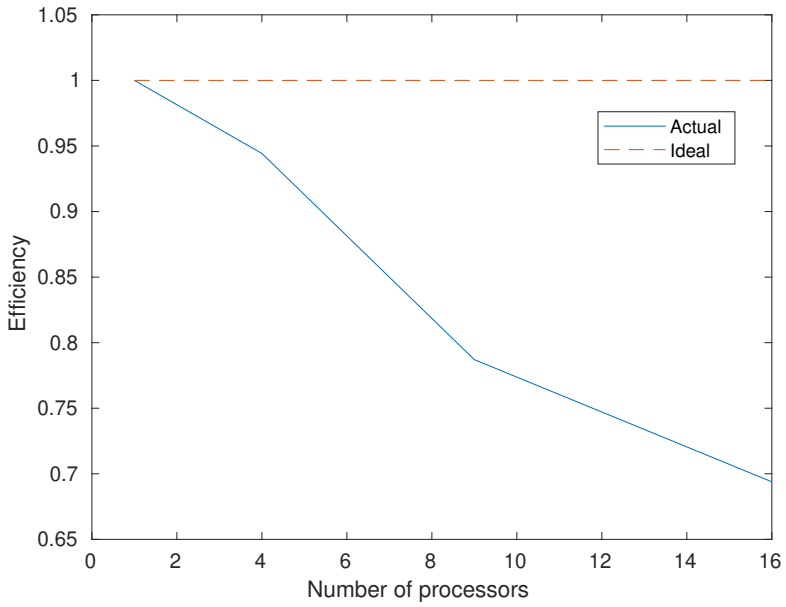

Figure 3: *Weak scalability.* Efficiency of the parallel implementation, which was calculated as $T(N,1)/T(N,p)$.

It can be seen that the efficiency decreases along with the communicators' size down to a limit of roughly 0.69: this implies that each communicator performs roughly 69% the amount of work in parallel compared to that of a single processor in serial, despite the fact that each communicator had the same workload. In other words, our parallel implementation scales relatively well, keeping each processor busy to an appreciable extent.

The deviation of the actual speedup and efficiency from their ideal counterparts is due to the expensive MPI calls that are necessary to transfer chunks of data across a series of communicators, which is necessary to complete the matrix multiplication. This is also exacerbated by the fact that we use a blocking send/receive call to exchange said data. Consider the following example with 20 communicators, indexed 0 to 19. Suppose that communicator 0 must send its data to communicator 19. In the best case scenario, the topology of the communicator network is such that these two communicators are directly adjacent to each other: in this case the data can be sent directly. In the worst case scenario, however, these two communicators are the farthest apart: in this case the data must be sent from communicator 0 to the intermediate communicators before finally arriving at communicator 19. Since the communicators will not start and end their local matrix computations at the same time, two adjacent communicators will likely have to wait for the other to finish their local computations before they exchange their data. This waiting period would increase the time taken for communicator 0's data to be sent to communicator 19. Additionally, as the communicators' size increases, the data exchange must occur over longer distances to complete the matrix multiplication. This larger distance, combined with the waiting period to exchange data between two adjacent communicators, would considerably increase the communication overhead.