COMP5426

Assignment 2 – Pairwise Inner Products of Vectors

A. Problem definition and Requirements

In the pairwise inner products of vector, first I need to initialize N vectors, N is the input defined by the user. User also needs to define the length of every vector, this value is called M. Therefore, the first step is to generate a random 2D matrix of size N by M and print this initial N vectors. Take N equals to 5, M equals to 2 as an example, all the number in these N vectors should be the float type, the output is shown in Fig 1.

```
The initial N*M vector is:
0.194 0.085
0.764 0.350
0.993 0.431
0.540 0.136
0.784 0.896
```

Fig 1 Initial N by M vectors

The second requirement is that in this assignment, I can't use master/worker programming model, as the processes are organized as a one-dimensional linear array. At the same time, I must use MPI non-blocking send and receive functions to overlap the communication and computation to reduce the waste of resources and time caused by communication and waiting of processes.

The third task is to distribute the workload for every process to achieve best load balancing. In my program the number of processes is restricted to odd numbers only and the input N must be divisible by number of processes.

Last, I need to consider two situations which are sequential computation and parallel computation. In sequential computation we only have one process to perform the production, and in parallel computation I need to distribute data to each processes and perform the production, after the calculation, all processes' results will be collected by the first process to get the final result, and then perform a sequential computation to self-check whether the result is correct.

B. Parallel algorithm design

This part is the most important task in this assignment, I will fully introduce and explain how I distribute workload to each process to achieve the load balancing.

As I mentioned before, the process number must be an odd number and the input N must be divisible by the number of process. Therefore, the first step is to distribute N / P vectors to every process. Taking N equals to 6, the number of process equals to 3 as an example, every process will get two vectors. Then processes using the vectors they have to perform

production, so now every process performs three productions. The process 0 calculates (0,0), (0,1), (1,1), the process 1 calculates (2,2), (2,3), (3,3), the process 2 calculates (4,4), (4,5), (5,5). Then process 1 needs to send the vectors it owned to process 0, process 2 sends the vectors to process 1, and process 0 sends the vectors to process 2. Now every process has 4 vectors, then they can perform productions, at this time every process can perform 4 productions. The process 0 will calculate (0,2), (0,3), (1,2), (1,3), other processes will also finish their calculations. Now, every process performs 7 calculations, it achieves the goal of load balancing for every process. Fig 2 shows the results calculated by each process, and for each colour it represents a process.

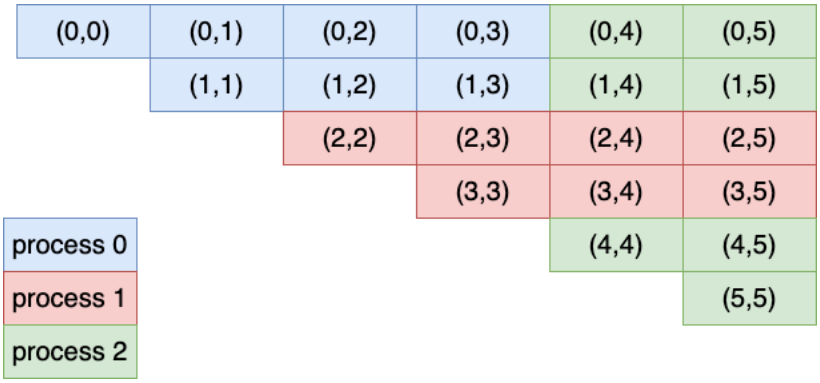

Fig 2 Results calculated by each process

Based on this algorithm, every process will perform the same number of calculations, if the number of processes and the number of vectors defined by users are following the requirements.

There are also two important things to be noted in this part. The first is that in this part, I need to use the MPI non-blocking send and receive functions here to achieve better performance. Second is to know how many iterations we need to calculate the results. In the example mentioned before, only two iterations, one is the self-calculation the second is every process gets data from the next process and then calculate the second results. After all we can have the result.

But if the number of process is 5, the iterations is not the same as two. For example, we have 5 processes and 5 vectors. Each process will have one vector at the beginning, so in the first iterations, they can only finish one calculation to get one result. Then every process will get the data form its next process to calculate another result. As we know for 5 vectors the final result will have (5+1) * 5 / 2 which is 15 vector pairs. Now every process only has two calculations, so only 10 vector pairs at total now. We need another iteration to calculate another five results, as process 1 has the data sent by process 2, it can send this data to process 0, so process 0 can perform the third calculation. Then it is very clear that process 0 will use its own data, the data of process 1, and the data of process 2 to calculate the result. Process 1 will use its own data, the data of process 2, and the data of process 3 to calculate the result. It is the same for other processes. We can know that if the number of processes is 5, 3 iterations are needed to calculate the results, if the number of process is 7, 4 iterations are needed to calculate the results. We can conclude that the equation between the number of iterations and the number of processes is (N+1) / 2.

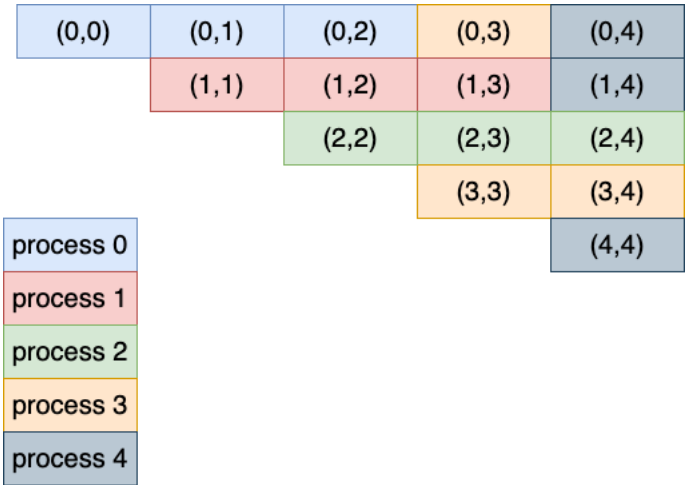

Fig 3 Results for 5 processes and 5 vectors

In conclusion, the parallel algorithm is distributing the number of vectors to each process equally, and then based on the number of iterations to send, receive, and calculate data between processes. After all iterations have been finished, other processes send vector pairs calculated by them to the processes 0, then print the final result. In this algorithm, every process will perform the same number of calculations to achieve the best load balancing.

C. Implementation and Testing

C.1 Implementation

The function that performing production. The “innerProduct” function is to product two vectors, the “product” function is to finish the calculation of the whole N by M matrix’s production.

```
float innerProduct(float* matrixOne, float* matrixTwo, int m) {
    float result = 0;
    //printf("matrixOne[0] is: %-6.2f , matrixTwo[0] is:  %-6.2f. \n", matrixOne[0], matrixTwo[0]);
    for (int i = 0; i < m; i++) {
        result += matrixOne[i] * matrixTwo[i];
    }

    return result;
}

void product(float** matrix, int rows, int m, float* results) {
    int i, j;
    int counter = 0;
    for (i = 0; i < rows; i++) {
        for (j = i; j < rows; j++) {
            results[counter] = innerProduct(matrix[i], matrix[j], m);
            counter++;
        }
    }
}
```

Fig 4 Product function

This is the core function of this program, I implement MPI non-blocking send and receive functions. In the first iteration, every process just needs to use its own data to perform calculation. If the number of process is larger than 1, the iteration number is larger than 1 too, in this case every process needs to use data sent by other processes to do the calculation, so I write a small swap function at the end of this function to swap the data in receive vector and send vector.

```
for (int iter = 0; iter < (numprocs+1)/2; iter++) {
    //every process sends its own sendVector data and store the received data in receiveVector.
    MPI_Isend( buf: sendVector[0],  count: vectorPerProcs * m, MPI_FLOAT,  dest: (myid - 1 + numprocs) % numprocs,  tag: 1, MPI_COMM_WORLD, &sendRequest);
    MPI_Irecv( buf: receiveVector[0],  count: vectorPerProcs * m, MPI_FLOAT,  source: (myid + 1) % numprocs,  tag: 1, MPI_COMM_WORLD, &receiveRequest);
    if (iter == 0) {
        //in first iteration only need to compute with itself
        for (int i = 0; i < vectorPerProcs; i++) {
            for(int j = i; j < vectorPerProcs; j++) {
                // vector product between i-th in local vec, j-th in send vec
                float triangle_ij = innerProduct(processVector[i], sendVector[j], m);
                // compute Cij
                /*  for every loop plus vectorPerProcs * ((numprocs + 1)/2) first
                 *  as it is a triangle need to minus the number of # which is a small triangle too.
                 *  (0,0) (0,1) (0,2) (0,3) (0,4)
                 *    #   (1,1) (1,2) (1,3) (1,4)
                 *    #     #   (2,2) (2,3) (2,4)
                 *    #     #     #   (3,3) (3,4)
                 *    #     #     #     #   (4,4)
                 */
                int counter  = vectorPerProcs * ((numprocs + 1)/2) * i - (i - 1)*i/2 + j -i;
                localResult[counter] = triangle_ij;
            }
        }
    } else {
        for (int i = 0; i < vectorPerProcs; i++) {
            for (int j = 0; j < vectorPerProcs; j++) {
                float triangle_ij = innerProduct(processVector[i], sendVector[j], m);
                /* the same as 0 iter, just need to notice j starts from the vectorPerProcs
                 * as vector per process is 2, it can calculate (0,0), (0,1)
                 * as vector per process is 3, it can calculate (0,0), (0,1), (0,2)
                 */
                int triangle_j = iter * vectorPerProcs + j;
                int counter = vectorPerProcs * ((numprocs + 1)/2) * i - (i-1) * i / 2 + triangle_j - i;
                localResult[counter] = triangle_ij;
            }
        }
    }
    MPI_Wait(&sendRequest, &status);
    MPI_Wait(&receiveRequest, &status);
    if(iter < (numprocs+1)/2 - 1){
        float** temp;
        temp = receiveVector;
        receiveVector = sendVector;
        sendVector = temp;
    }
}
```

Fig 5 Parallel computation part

There are many functions which are also important and need some calculations to determine who to write the equation for variables. As the limitation of pages, more details can be found in the program, I have provided the necessary notes and explanations in the program.

C.2 Test

I perform 6 test cases to test my program.

1.Sequential

(1). mpirun -np 1 inner 9 2

(2). mpirun -np 1 inner 15 3

2.Parrallel

(1). mpirun -np 3 inner 3 2

(2). mpirun -np 3 inner 9 2

(3). mpirun -np 5 inner 5 2

(4). mpirun -np 5 inner 15 3

To ensure the results of my program is correct, I use hand calculations to check whether every result is correct. The test case I choose is mpirun -np 3 inner 3 2. The results are shown as below. As I can get the results for both sequential and parallel calculations, then I use the initial vector's value to check the result. After the calculation, I can make sure the results are correct.

```
The initial N*M vector is:
0.445 0.586
0.362 0.273
0.386 0.707
Triangle result is:
0.541     0.321     0.586
          0.206     0.333
                    0.649
Then perform self check !
Sequential computation:
Triangle result is:
0.541     0.321     0.586
          0.206     0.333
                    0.649
```

Fig 6 Test case

D. Possible improvement

I think in this program I meet all important requirements and I can also get the right results for the calculations. There are only some small parts I think I need to improve, like when I am using memcpy function, the data I copied will be changed in the parallel calculation part, may be it is the memory leak problem, but I don't figure out why. I have to use for loop to copy the data one by one to get the correct result.

E. Manual

The example steps to run the program.

1. unzip the a2_ls.zip file
2. cd a2_ls (Make sure change to the directory contains file A2_inner.c)
3. make
4. mpirun -np 3 inner 9 3 (just an example)
5. After that use "make clean" to clean the project

Input:

The 2 inputs we need to input are the N which represents N vectors, M which represents the length of every vectors. Besides we can use the number after np to control the number of processors.

Output:

When the number of processors is one we will run sequential computation, we can see the value of initial vectors and the results of sequential calculation. If the number of processors is larger than one, the output will have the value of initial vectors, results of parallel calculation, and followed by a self-check sequential calculation. If the number of processes is not an odd number, or the N cannot be divisible by number of processes, the program will notice you input the right value.