Cannon's Matrix Multiplication Algorithm Using MPI

Chenyin Zhang
Pennsylvania State University
State College, Pennsylvania, USA
cbz5089@psu.edu

Abstract

In this paper, I report on the implementation of Cannon's Matrix Multiplication using Message Passing Interface (MPI). During the implementation phase, I analyzed the algorithm to decide the implementation detail for data transmission. During the performance analysis part, I tested for a various number of nodes under the situation of different data sizes and evaluated the output for each case. By comparing and justifying the result of the actual performance, I managed to gain a better understanding of MPI.

1 Introduction

MPI is a communication protocol for parallel programming under the distributed memory environment. By providing the functionality of virtual topology, synchronization, and communication between a set of processes (nodes), MPI could help the programmer to coordinate the data transmission process in distributed computing. However, working with MPI also requires extra effort when designing the code: programmers should check for the possibility of deadlocks when implementing the communication between nodes. To have a better understanding of MPI, we are assigned to implement Cannon's Matrix Multiplication algorithm using MPI.

The remainder of this paper is organized as follows. Section 2 briefly inspects Cannon's Matrix Multiplication in two stages: the initial shifting stage and the computation stage. Section 3 addresses non-algorithm-related code, including pre-computation data scattering and correctness verification. In section 4, I analyze the structure for both stages and point out the corresponding implementation. Performance Scalability Analysis is conducted in Section 5, which evaluates the performance of implementation under various task sizes and thread numbers. I provide a summary in section 6.

2 Case Inspection

Cannon's Matrix Multiplication algorithm is a distributed algorithm that is suitable for N*N mesh layout. By shifting the data after each round of computation, Cannon's algorithm is managed to complete the matrix multiplication using a constant amount of memory regardless of the number of processors. The following parts would discuss two stages of Cannon's algorithm based on the example of squared matrix multiplication A*B=C. It also assumes that the data for matrix A and matrix B are properly scattered into each processor.

Before the initial shift, each processor with coordinate [i, j] in the mesh would be having the submatrix of A_i,j, and B_i,j. The initial shift for Cannon's algorithm requires that for each processor in row i, the submatrix A_i,j should be shifted left by i. Also, for each processor in col j, the submatrix B_i,j should be shifted up by j. After the initial shifting, each processor iterates through the following three steps N times: 1) calculate the local result for each submatrix and add to the result submatrix C' 2) send its submatrix A' to the left node and its submatrix B' to the up node 3) receive the new submatrix A'' and submatrix B'' from other nodes for the next iteration. After N iterations, we would receive the result for the whole matrix multiplication by gathering each result submatrix C' inside each processor.

3 Data Distribution And Verification

Before we implement Cannon's algorithm, we should first create a reliable way to distribute the data into each processor and also to evaluate the correctness of our computation. One of the challenges is to generate an efficient way of scattering the data into the corresponding processors. Notice that the basic MPI_scatter requires the sender's data to be stored in contiguous memory addresses, which would require us to reschedule the generated matrix A and B before sending. One workaround for this is to use `MPI_scatterv()` to scatter the submatrix. By creating an `MPI_Datatype` to represent the submatrix, I could use `MPI_scatterv()` to scatter matrix A and B into each processor. Another issue is that we are required to verify the result of each processor with the serial matrix multiplication. My solution is to precompute the matrix multiplication result and send it to each processor. With that method, each processor could compare its output with the serial result locally after the algorithm.

4 Code Implementation

For the actual code, I construct the 2-D mesh topology outside the implementation of Cannon's algorithm using `MPI_Cart_create()`. I then distributed the data into corresponding processors. I also pass in the `MPI_Comm` of the 2-D mesh to the function where I implemented Cannon's algorithm to persist the layout of our topology.

4.1 Initial shift

As we discussed in section 2, Cannon's algorithm requires the processors in the mesh to shift its submatrix A to the left by row number and shift its submatrix B to the above by column number. To achieve this, we could use `MPI_Cart_shift()` to get the destination. It should be awarded that for each processor we are trying to send and receive the data in the same position, so using an `MPI_sendrecv()` function would cause potential memory leaks. Although MPI also provides an `MPI_sendrecv_replace()` function to address the situation of sending and receiving in the same location, I decide to use a helper buffer to handle the overlapping send/receive issue.

Another thing to be noticed is that we cannot regard the initial shift as a group of ring value passing. Take the 10 * 10 mesh as an example, we could see from **Figure 1** that row 8 could have 2 rings when doing the initial shift. Thus, the strategy to handle the initial shift phase is different from the matrix multiplication computation phase.

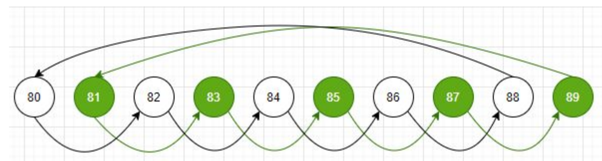

Figure 1 Initial data shifting for submatrix A at row 8 under 10*10 mesh

The following is the code to implement the initial shift for submatrix A. The code to implement initial shift for submatrix B would have the same structure.

```
MPI_Request request;
// Create the buffer to handle the sendrecv() for the same address
double *temp_buffer_a = new double[temp_n * temp_n];
// Get the initial left shift for Matrix A on row nodes, notice tag is just an indication
MPI_Cart_shift(comm, 1, -coord[0], &source, &dest);
// DO the initial shift for submatrix A
if (coord[0] != 0) {    // avoid interacting with the first row
    MPI_Isend(a, temp_n * temp_n, MPI_DOUBLE, dest, 1, comm, &request);
    MPI_Recv(temp_buffer_a, temp_n * temp_n, MPI_DOUBLE, source, 1, comm, &status);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    memcpy(a, temp_buffer_a, temp_n * temp_n * sizeof(double));
}
```

4.2 Matrix multiplication computation

After the initial shift, then we go on and compute the matrix multiplication. We first run the serial matrix multiplication code for each processor, then we shift the submatrix A' to the left node and submatrix B to the up node. In this phase, all the data would be only shifted by one node, thus we could view this as a set of ring communication. In order to avoid the deadlock, I invert the send/receive sequence for one processor in each ring (for submatrix A shift, each processor at column 0 would be inverted; for submatrix B shift, each processor at row 0 would be inverted). After iterating the process by N times, we could get the result for each submatrix.

The following is the code to implement the matrix multiplication computation.

```
for (int i = 0; i < dim; i++) {
    // compute the Local result
    mm(temp_n, a, b, c);
    // Do the left shift for Matrix A by 1
    if (coord[1] == 0) {
        MPI_Recv(temp_buffer_a, temp_n * temp_n,
MPI_DOUBLE, right, 0, comm, &status);
        MPI_Send(a, temp_n * temp_n, MPI_DOUBLE, left,
0, comm);
        memcpy(a, temp_buffer_a, temp_n * temp_n *
sizeof(double));
    } else {
        MPI_Send(a, temp_n * temp_n, MPI_DOUBLE, left,
0, comm);
        MPI_Recv(a, temp_n * temp_n, MPI_DOUBLE,
right, 0, comm, &status);
    }
    // Do the up shift for Matrix B by 1
    if (coord[0] == 0) {
        MPI_Recv(temp_buffer_a, temp_n * temp_n,
MPI_DOUBLE, down, 0, comm, &status);
        MPI_Send(b, temp_n * temp_n, MPI_DOUBLE, up,
0, comm);
        memcpy(b, temp_buffer_a, temp_n * temp_n *
sizeof(double));
    } else {
        MPI_Send(b, temp_n * temp_n, MPI_DOUBLE, up,
0, comm);
        MPI_Recv(b, temp_n * temp_n, MPI_DOUBLE, down,
0, comm, &status);
    }
}
```

5 Performance Scalability Analysis

Before we conduct the actual performance analysis, we could try to estimate the upper bound of the speed up for Cannon's algorithm. One estimate is that the algorithm would have a T time speed-up in computation when using T processors. One flaw of this estimation is that for each processor, the access speed of the local data would be affected by the data size. The ACI-b cluster we used for performance analysis is installed with E5-2650V4 CPUs, which should have a **32KiB** write-back L1 cache. This means we could store up to 4000 8-byte double-precision floats or (integer-typed) long inside the L1 cache. Thus, with this information, we could estimate that the computation speed of Cannon's algorithm should be similar to the computation speed for a N*N/T serial matrix multiplication. However, we should also be aware of the data transmission cost for the algorithm.

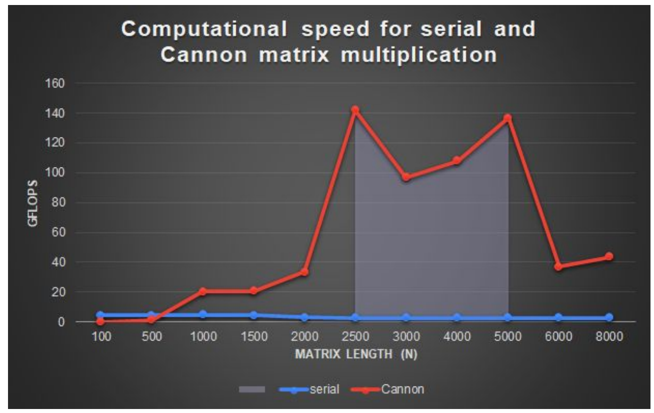

Figure 6 Computational speed for serial and Cannon matrix multiplication

For the performance scalability analysis, I took 12 data sizes N ranging from 100 to 10000 and executed with 100 processors. Each N value would be tested three times and calculate its mean as result. I also define the number of FLOPS for N*N matrix multiplication is *2(N*N*N)*. The following graph shows the performance of my code when calculating double-precision floating-point matrix multiplication (see appendix for exact data). We could see that our Cannon's algorithm is less efficient than the serial algorithm under $N = 100$ and *500* due to the data shifting overhead. However, after the N size becomes bigger than 500, the efficiency of Cannon's algorithm becomes significantly greater than the serial implementation. We could also observe that for N size between *2500* and *5000*, there is a huge spike in the computational efficiency for Cannon's algorithm. If we focus on the data where the N size is big enough, we could see that Cannon's algorithm only gains 13.535 times (for $N = 6000$) and 16.083 times (for $N = 8000$) speed up than our serial code. One proper explanation for this difference is that each processor could handle the data using their L1 cache when N size is small, thus resulting in high GFLOPs. When N size becomes bigger, the processors are forced to use a lower level of cache, which causes a significant drop in speed. Further research is required to prove this hypothesis.

6 Summary

In this project, I implement Cannon's matrix multiplication algorithms in the context of a distributed system using the MPI protocol. I also evaluate the implementation for double precision floating-point data type in a wide range of matrix sizes. This process has helped me to gain a better understanding of using MPI protocol.

Reference

[1] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Montana State University, USA, 1969.

[2] "MPI partition matrix into blocks," *Stackoverflow.com*. [Online]. Available: https://stackoverflow.com/questions/7549316/mpi-partition-matrix-into-blocks. [Accessed: 06-Dec-2020].

[3] "MPI Send and Receive," *Mpitutorial.com*. [Online]. Available: https://mpitutorial.com/tutorials/mpi-send-and-receive/. [Accessed: 06-Dec-2020].

Appendix A

A.1 Running time for matrix multiplication under 100 processors

Running time for matrix multiplication under 100 processors(seconds)					
Matrix size/ Trial	Serial	Trial 1	Trial 2	Trial 3	Average
100	0.000443	0.029423	0.031697	0.030956	0.030692
500	0.055011	0.195568	0.096688	0.225134	0.172463333
1000	0.41625	0.040183	0.043891	0.20866	0.097578
1500	1.587444	0.057963	0.69367	0.227091	0.326241333
2000	5.028117	0.199253	0.831478	0.39493	0.475220333
2500	11.17312	0.315146	0.161449	0.18301	0.219868333
3000	19.537321	0.691344	0.712082	0.268615	0.557347
4000	45.670061	1.563616	0.863385	1.126196	1.184399
5000	91.258314	2.21788	1.630841	1.632498	1.827073
6000	157.801053	13.520836	10.278461	11.176677	11.658658
8000	377.850202	44.196348	13.038735	13.246049	23.49371067

A.2 Running time for matrix multiplication under 100 processors

Computation efficiency for matrix multiplication under 100 processors(GFLOPS)		
Matrix size/ Implementation	serial	Cannon
100	4.514672686	0.065163561
500	4.544545636	1.449583486
1000	4.804804805	20.49642337
1500	4.2521185	20.6902048
2000	3.182105747	33.66859302
2500	2.79689111	142.1305175
3000	2.76394087	96.88757632
4000	2.802711387	108.0716887
5000	2.739476427	136.8308765
6000	2.737624317	37.05400742
8000	2.710068685	43.58613309