Report of Assignment 3

Hanzhang Song studentID:hs50

November 13, 2018

1 The Memory View of My Matrix

I use an 1-D array to represent my matrix. First, I divide the array by the number of blocks. Second, I flat each block to an 1-D array. Finally I concat these 1-D arrays. It is more clear to illustrate it by the following example and figures:

If we have a 4*4 matrix, which is divided by 2*2 blocks then the context of the matrix and the corresponding array's memory view are shown in fig 1.

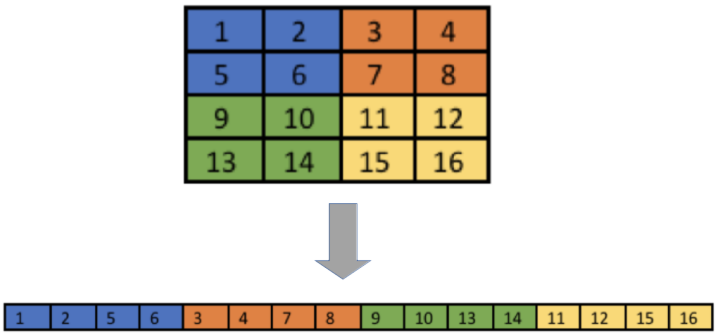

Figure 1: memory allocation of matrix.

2 The Performance of My Program

The results are shown in table 1.

processors	c	time(s)	processors	c	time(s)
4	1	379.778	32	2	63.6302
8	2	189.327	36	1	42.9067
9	1	125.796	48	3	64.0566
16	1	120.224	49	1	24.5961
18	2	88.4319	50	2	30.7748
25	1	46.7677	64	1	24.9107
27	3	45.9675	64	4	33.5122

Table 1: Performance

3 The Bandwidth and Latency Lower Bounds

Some definition of varialbles are list as below:

$$\begin{aligned} n &:= \text{The size of matrix, there are totally n*n elements.} \\ c &:= \text{The number of layers, so the matrix will be copied c times} \\ p &:= \text{The number of processors} \end{aligned} \tag{1}$$

The latency cost's lower bound actually is times that submatrix will be shifted, it is calculated as below:

$$\begin{aligned} \text{Number of processors in each layer: } p_{each\_layer} &= \frac{p}{c} \\ \text{The times of total shifts: } S_{total} &= \sqrt{p_{each\_layer}} = \sqrt{\frac{p}{c}} \\ \text{The latency cost: } S_{2.5d} &= \frac{S_{total}}{c} = \sqrt{\frac{p}{c^3}} \end{aligned} \tag{2}$$

Every time, we only need to send and receive the same size of data, so the bandwidth cost's lower bound is calculated as below:

$$\begin{aligned} \text{Local memory for storing the submatrix: } M &= n^2/p_{each\_layer} = \frac{c*n^2}{p} \\ \text{The bandwidth cost:} W_{2.5d} &= S_{2.5d} * M = \frac{n^2}{\sqrt{c*p}} \end{aligned} \tag{3}$$

4 Collective Communication

There are two kinds of collective communication that I used in my implementation of the 2.5D algorithm.

The first is MPI_Scatterv(...), I use it to allocate each processor in the first layer a block of submatrix. The reason why I do not use the MPI_Scatter is that when the size of the matrix is not divisible by the number of processors in the first layer, processors will be allocated different size of data, to be more specific the processors on boundary will have an incomplete submatrix. I need to calculated the offsets and strides in advance. Because MPI_Scatterv(...) involves all processors into communicating, it is faster than using point-to-point communication to do this job.

The second is MPI_Bcast(...), I use it to copy the submatrix in the first layer to all the processers in the below layers. In this job, we have only a small amount of layers. So actually MPI_Bcast(...) already lose its advantages compared with only using MPI_Send and MPI_Recv to do the job. However I still use MPI_Bcast(...), because the code will be more clear and the performance is still good enough.

5 Analysis with Jumpshot

Provided by the visualization of Jumpshot, I first discover a easy bug which make the program perform serially in the broadcast phase. The view of Jumpshot is as fig 2

I found that I only used processer 0 to create communicators, once a communicator is created I will use it to broadcast submatrix. The code is like following:

```
for(int row = 0; row < layer_size; ++row)
{
```

```
  for(int col = 0; col < layer_size; ++col)
  {
    for(int layer = 0; layer < layer_num; ++layer)
      MPI_Cart_rank(comm_3d, new int[3] {row, col, layer}, &stick_ranks[lay
    int index = FLAT_IDX(row, col, layer_size);
    MPI_Group_incl(group_all, layer_num, stick_ranks, &group_stick_array[in
    MPI_Comm_create(MPI_COMM_WORLD, group_stick_array[index], &comm_stick_a

    // Broadcast submatrix
    if(comm_stick_array[index] != MPI_COMM_NULL)
    {
      MPI_Bcast(sub_A, sub_size*sub_size, MPI_DOUBLE, 0, comm_stick_array[i
      MPI_Bcast(sub_B, sub_size*sub_size, MPI_DOUBLE, 0, comm_stick_array[i

    }
  }
}
```

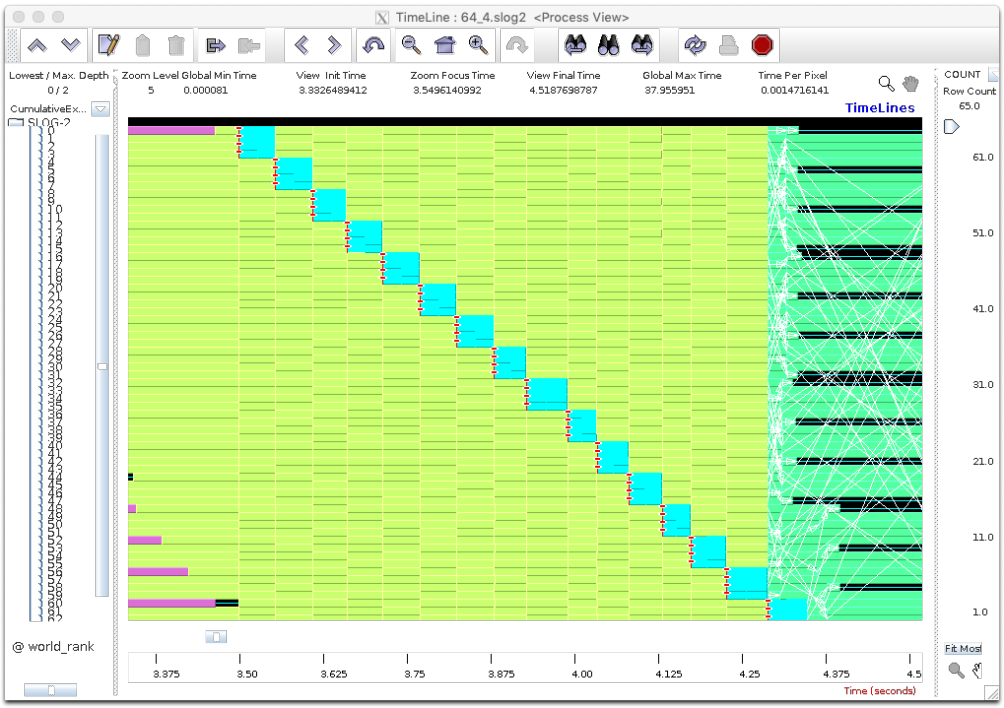

Figure 2: One bug causing serial performance.

This is a bad idea because communicators could be created at the same time. So I fixed the bug as following and the fixed result is shown in fig 3:

```
for(int layer = 0; layer < layer_num; ++layer)
  MPI_Cart_rank(comm_3d, new int[3] {my_coords[0], my_coords[1], layer}, &s

MPI_Group_incl(group_all, layer_num, stick_ranks, &group_stick);
MPI_Comm_create(MPI_COMM_WORLD, group_stick, &comm_stick);

// Broadcast submatrix
```

```
if(comm_stick != MPI_COMM_NULL)
{
  MPI_Bcast(sub_A, sub_size*sub_size, MPI_DOUBLE, 0, comm_stick);
  MPI_Bcast(sub_B, sub_size*sub_size, MPI_DOUBLE, 0, comm_stick);
}
```

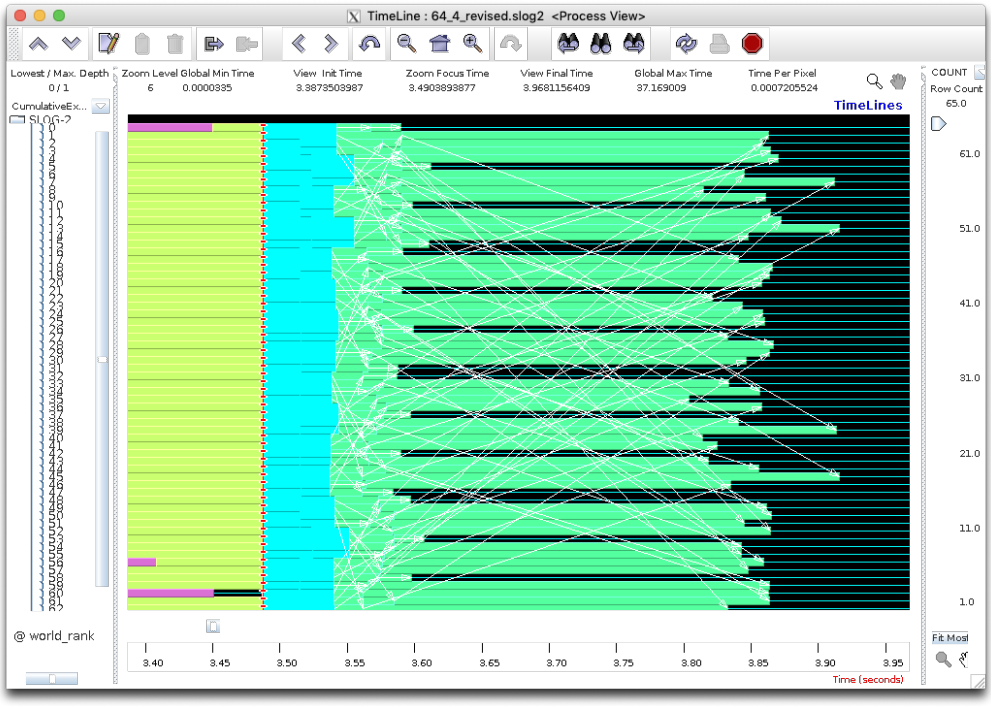

Figure 3: Create communicators in parallel.

The second thing I used Jumpshot for is to check the difference between block communication and non-block communication. I tried to overlap communcation with computation by using non-block communication. Once a processor received another submatrix from other processors it can start the computation, it will not need to wait for the send. So every processor will use MPI_ISend and MPI_IRecv, it will begin the computation after MPI_wait(recv_status) and then MPI_wait(send_status). It turns out that this does not work out. The reason is if a processer is computing it will spend less time to send the submatrix to others, and other processors will not begin computation until they receive the data they need. So some processors may begin their computation earlier, but it will cause much more waiting time for other processors.

So I only use non-block communicator and MPI_wait as below:

```
for (int idx = 0; idx < offset-1; ++idx)
{
  MPI_ISend(sub_A, ...);
  MPI_ISend(sub_B, ...);
  MPI_IRecv(sub_A, ...);
  MPI_IRecv(sub_B, ...);
  MPI_Wait(all_status);
  mat_multi(sub_A, sub_B, sub_C, sub_size);
}
```

In this way, it seems there are no difference between block communicator and non-block communicator, the results are shown in fig 4.

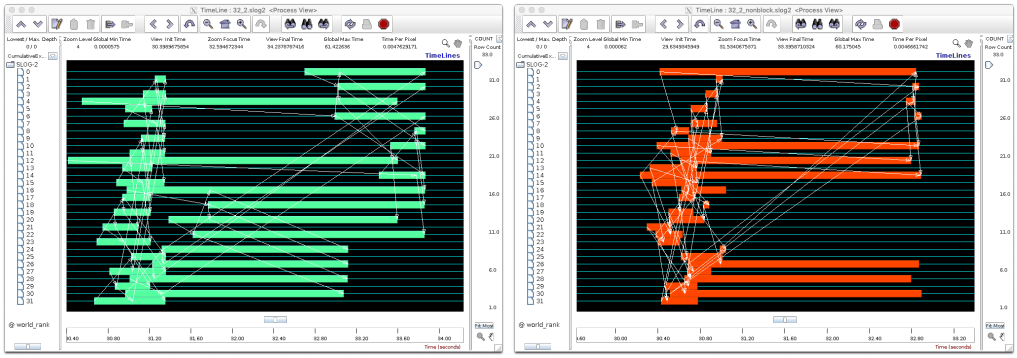

Figure 4: Left is block communicator and right is non-block communicator.

6 Analysis with HPCTools

The fig 5 shows how MPI_Scatter works: processor 0 will firstly send about half of its task to another processor and then these two processor will continue to split their task until the job is finished. It proves that using MPI_Scatter will be faster than only letting processor 0 do all the send jobs.

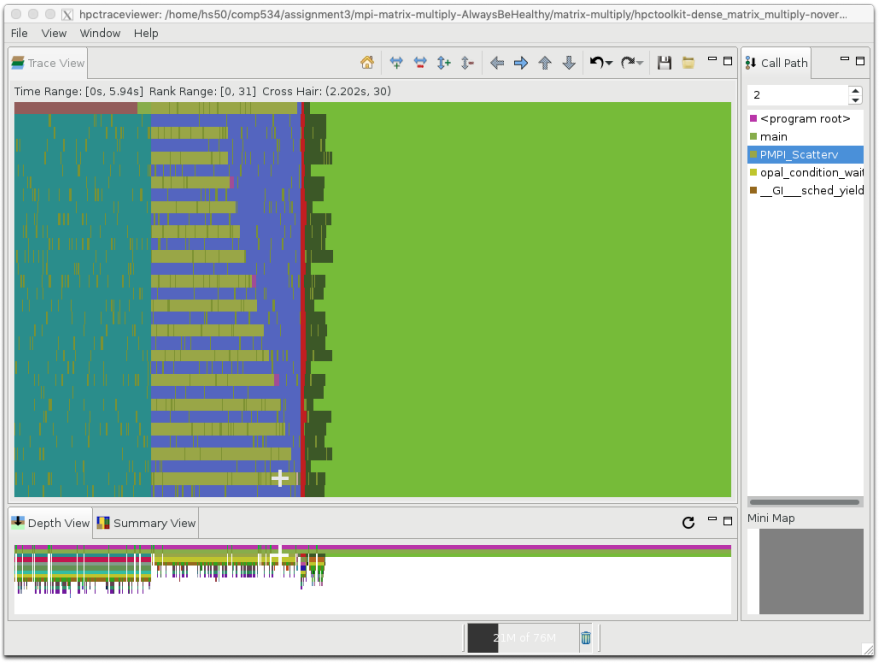

Figure 5: Hpctrace view of MPI_Scatter.

I used a 4*4*2 mesh-grid for the computation. And hpcviewer tells me that 93.2% of the time is spent on multi_matrix, which is good performance. The result is shown in 6

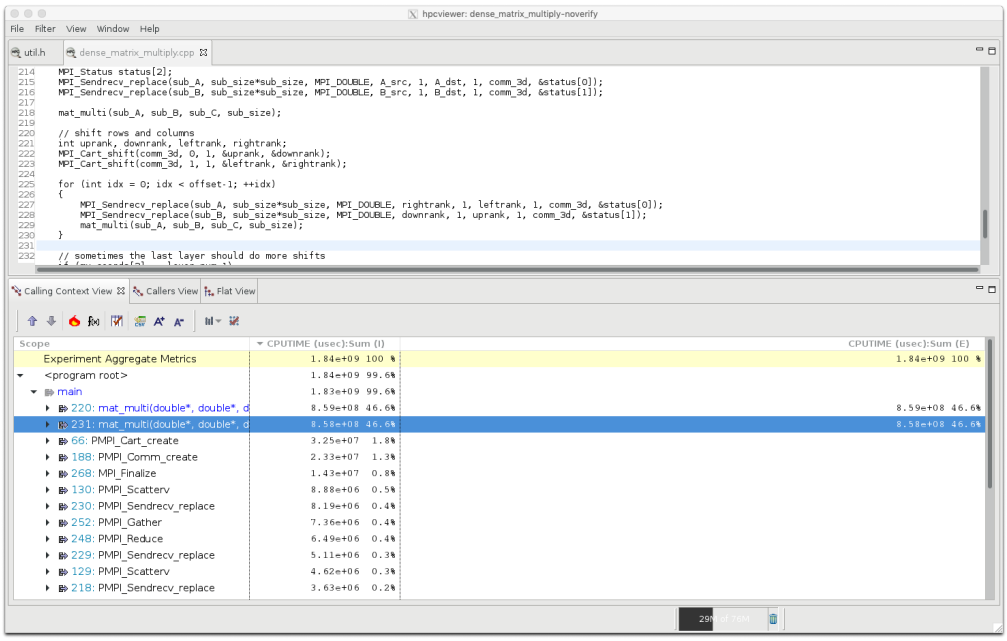

Figure 6: Hpcviewer of function call.