EECS 587 Parallel Computing Homework 3

Ethan Zhang

2018/10/10

1 matrix decomposition

In this task, I decompose the whole $m \times n$ matrix into p stripes as shown in figure 1, where p is the number of processors.

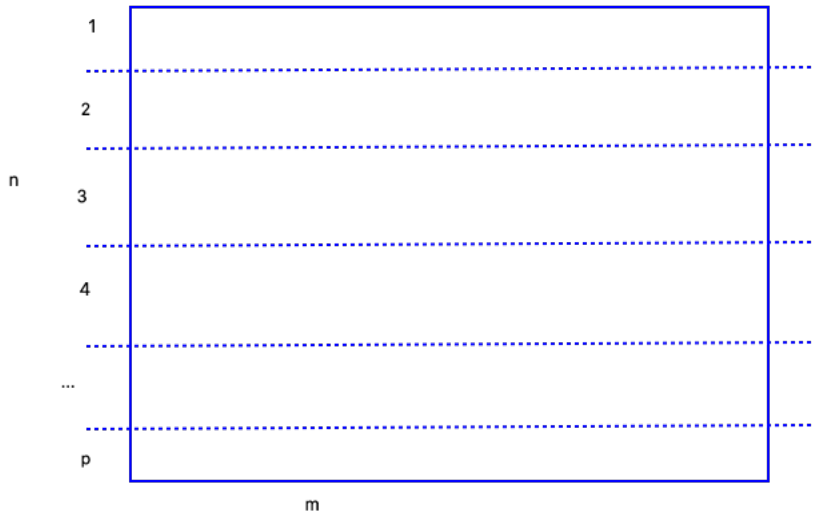

Figure 1: matrix decomposition

By doing that each processor, expect the processor p, process $ceil(\frac{n}{p})$ rows, while the processor p handles the rest rows of the matrix. And each processor can run in parallel with the designed communication strategy in section 2 below.

2 communication strategy

2.1 iteration step

In each iteration, for each processor, it actually process part of the whole matrix A. For example,according to the matrix decomposition in section 1, the processor 0 only need to process row 0 to row 124 when $n = 500$. While initializing the local matrix, I use a local A to do the task. In order to initialize an $m \times n$ matrix, I only initialize a small local A matrix while mapping the corresponding $m \times n$ matrix to the local matrix.

To facilitate the communication by sending values to or receiving values from other processor, since the calculation of z needs values from nearby "pixels", I add two row of ghost cells on each local matrix as shown below.

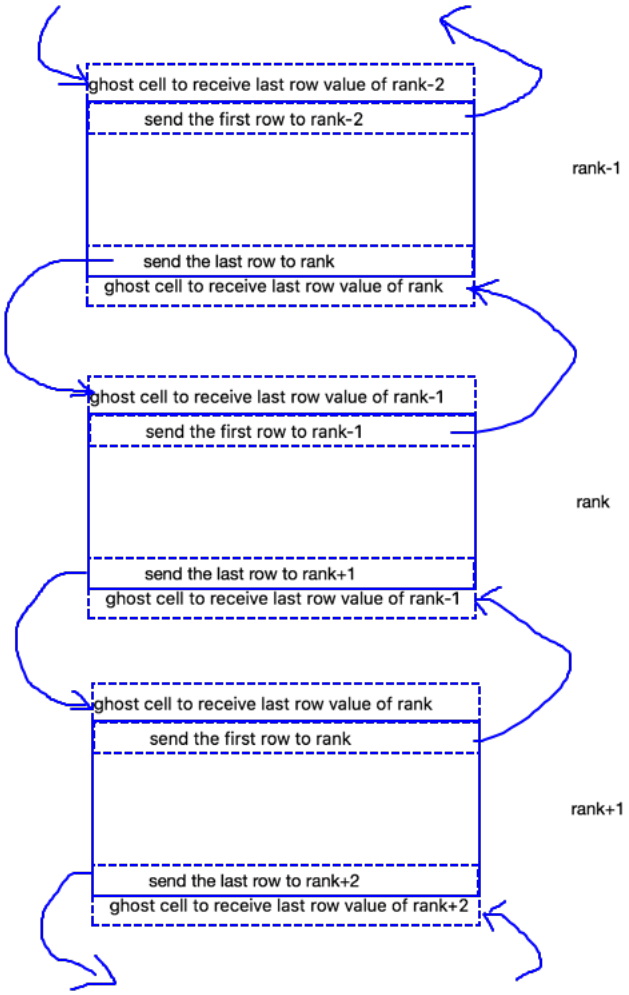

Figure 2: communication strategy at iteration steps

For processor 1 (rank 0), it only receives first row of rank 1 and send its last row to rank 1 during the iteration steps. For processor p (rank p-1), it only sends first row to rank p-2 and receive the last row of rank p-2. For other intermediate processors, they need to send first row and last row to nearby processors and receive corresponding ghost cell values. In the code file, check the iteration block, which looks like:

```
for(int t=0; t<10; t++){
                // send self_prev, self_tail
                double self_prev[m];
                double self_tail[m];

                for(int num =0; num<m; num++){
                        self_prev[num] = A[num][1];
                        self_tail[num] = A[num][n_row-2];

                }  // two ghost cells
```

```
    double prev[m];
    double tail[m];
    int flag_prev = 0; // check if prev has value or not
    int flag_tail = 0;
    ...
    ...
}
```

2.2 sum step

When calculating verification sum, another communication strategy is introduced. I also send information of one processor to its nearby processor. The communication graph is shown below.

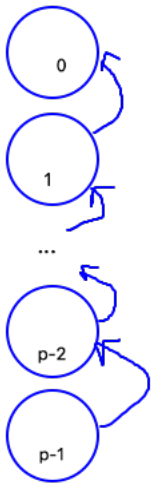

Figure 3: communication during verification sum step

Each processor calculates their own sum and square sum of all "valid" entries of local matrix. "Valid" means the values that can be mapped to the original $m \times n$ matrix A. And finally, rank 0 processor will get the total sum and total square sum. If the number of processor is 1, it is a serial case and no communications are needed, and the calculation is shown in function *serial* in the code file.

3 verification

The total result for matrix $(m = 2000, n = 500)$is shown below.

number of p	p=1	p=4	p=16	p=36
sum	3608956.401324	3608956.401324	3624912.222651	3586660.533163
square sum	4677423951.717626	4677423951.718682	4677306668.874214	4676224197.451056
time (in seconds)	44.704517	11.885118	4.606751	2.177365

The total result for matrix $(m = 1000, n = 1000)$is shown below.

number of p	p=1	p=4	p=16	p=36
sum	6536887.112126	6572568.494917	6563077.302879	6521883.630315
square sum	37151211116.024208	37151523391.080154	37150839317.893959	37151573399.266899
time (in seconds)	173.909204	72.284858	22.151456	9.960572

An interesting thing that I noticed is that every time I run your code with same parameter in different CPU cores, the answers would be a little different. Lately I realize that different CPU has different Instruction Set Architecture (ISA), which means different would provide a different precision when you calculating $sin(x), cos(x)$ and $sqrt(x)$. As you can see from the results, ideally all for same m and n, the sum and square sum should be exactly the same. However, it is not. That is because they are running on different CPUs with different ISA and it is a common issue when calculating function values like $sin(x)$ and so on . When the number of processors increases exponentially, the precision will be influenced at the same time. On flux, even running the same program at different time will give you different precision result as every time you run your program on different processors, which may has different ISA.

The timing result is shown in the figure below.

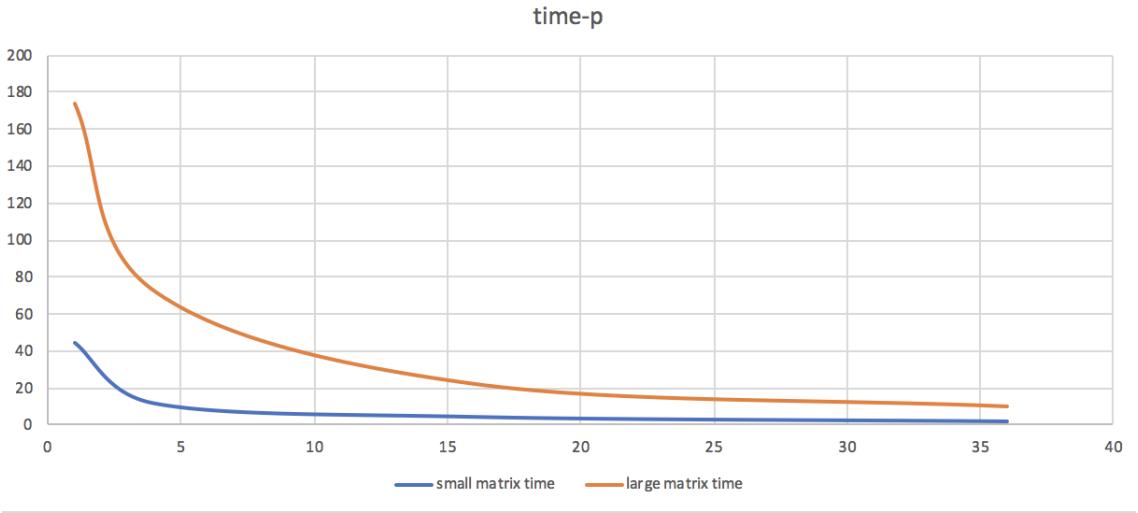

Figure 4: two matrix timing

Obviously, it it not perfectly speed up and scaling. When the number of processors are too large, the final time only does not decrease exponentially anymore due to the large amount of communication time. When the number of processor increases, the communication time keep increasing and the total time would not decrease like the small number of processors case.

4 Parallel optimization

In the program, I only considering parallelize the matrix decomposition. Slice the big matrix into small pieces and let each processor calculate it in serial, although communication with other processors are included. Actually, we ca further parallellize the algorithm. Rather than decompose the matrix into stripes, we can decompose it into a $\sqrt(p) \times \sqrt{p}$ matrix and let let each processor handle one grid as shown below.

Another optimization method is to divide the matrix into small grid cells and p processor handle one small cell each time. In this way, the processor can be load balanced.

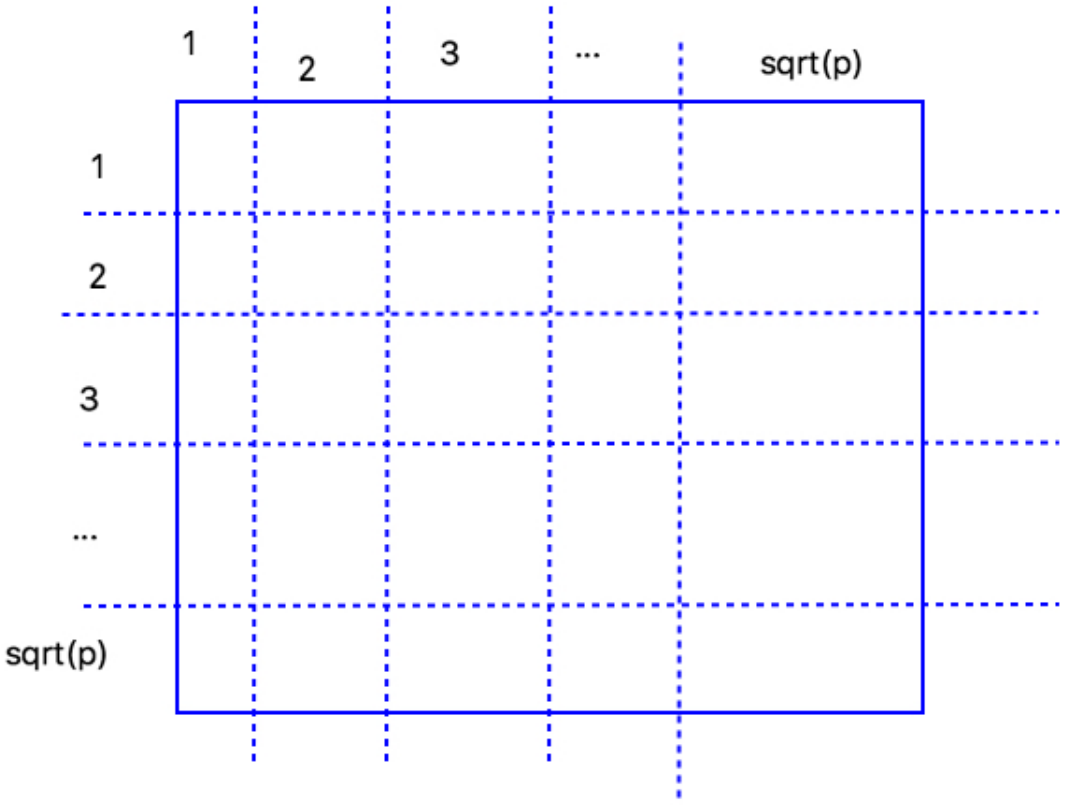

Figure 5: Optimization 1