HPC HW5

maoyiluo

April 2020

1 Github repo for this homework

https://github.com/maoyiluo/HPC2020-hw5

2 MPI-parallel two-dimensional Jacobi smoother

2.1 Implementation

The idea of implementation is straight forward. Each process i has a matrix whose size is $N_l + 2 \times N_l + 2$. The interior $N_l \times N_l$ points is used for local update and the boundary point will used to receive the data from other process. For example, on the left side of the local matrix, we send the 2rd column, which is a local column, to the previous process. Then we store the data sent by the previous process in the 1st column, which is a boundary column. Repeat the similar step for sending and receiving data to and from the next, above and below process. Then repeat the Jacobi iteration step.
Here is the code of sending and receiving data to and from the process below.

```
if (row > 0)
{
/* If not the first row, send/recv bdry values to the process below */
    memcpy(send_bottom_boundary, lunew[1], lN*sizeof(double));
    //async send the second rows to the below process.
    MPI_Isend(send_bottom_boundary, lN,MPI_DOUBLE,
    mpirank - process_per_line, iter, MPI_COMM_WORLD, send_bottom_quest);
    //async receive the row from the below process.
    MPI_Irecv(receive_bottom_boundary, lN,MPI_DOUBLE,
    mpirank - process_per_line, iter, MPI_COMM_WORLD, receive_bottom_quest);
}
```

Since we are using the **MPI_Isend** and **MPI_Irecv** we have to wait for the process return.

```
/*wait for the updated data*/
if(row > 0) MPI_Wait(receive_bottom_quest, &bottom_status);
if(row < process_per_line - 1) MPI_Wait(receive_up_quest, &up_status);
if(col > 0) MPI_Wait(receive_left_quest, &left_status);
if(col < process_per_line - 1) MPI_Wait(receive_right_quest, &right_status);
```

2.2 Weak scaling study

Before showing the plot, Table 1 shows the timing table of the weak scaling study.
Figure 1 shows both the plot of the real time and the theoretical time. You can see that for a small number of processes it holds but once the process number increases the communication time domains the time so it's no longer close to the theoretical time.

Number of process	N	time
4	200	1.041784
9	300	1.081043
16	400	1.081285
25	500	1.920170
36	600	3.152661

Table 1: Time of 10000 iterations of Jacobi method for an $N \times N$ matrix under different number of processors

2.3 Strong scaling study

Before showing the plot, Table 2 shows the timing table of the strong scaling study.
Figure 2 shows both the plot of the real time and the theoretical time under the assumption. I'm using the run time of 4 processes as the baseline. Theoretical time for N processes is computed by $T_{baseline} \times \frac{4}{N}$.

Number of process	Time
4	43.316991s
9	19.337573s
16	10.643397s
25	9.401817s
36	8.812104s

Table 2: Time of 10000 iterations of Jacobi method for an 1440×1440 matrix under different number of processors

3 Parallel sample sort

3.1 Implementation

For each process, I'm taking the $i \times \frac{N}{p}$th entry as the sample.

```
int *sample = (int *)malloc(p * sizeof(int));
    for (int i = 0; i < p - 1; i++)
        sample[i] = vec[N / p * i];
```

Then using **MPI_Gather** to send all these sample array to the root process.

```
int *gathered_sample;
if (rank == 0)
    gathered_sample = (int *)malloc(p * (p - 1) * sizeof(int));
MPI_Gather(sample, p - 1, MPI_INT, gathered_sample, p - 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Pick up the $i \times p$th entry as the ith splitters

```
int *splitters = (int *)malloc((p - 1) * sizeof(int));
if (rank == 0)
{
    std::sort(gathered_sample, gathered_sample + p * (p - 1));
    for (int i = 0; i < p - 1; i++)
    {
        splitters[i] = gathered_sample[i * p];
    }
}
```

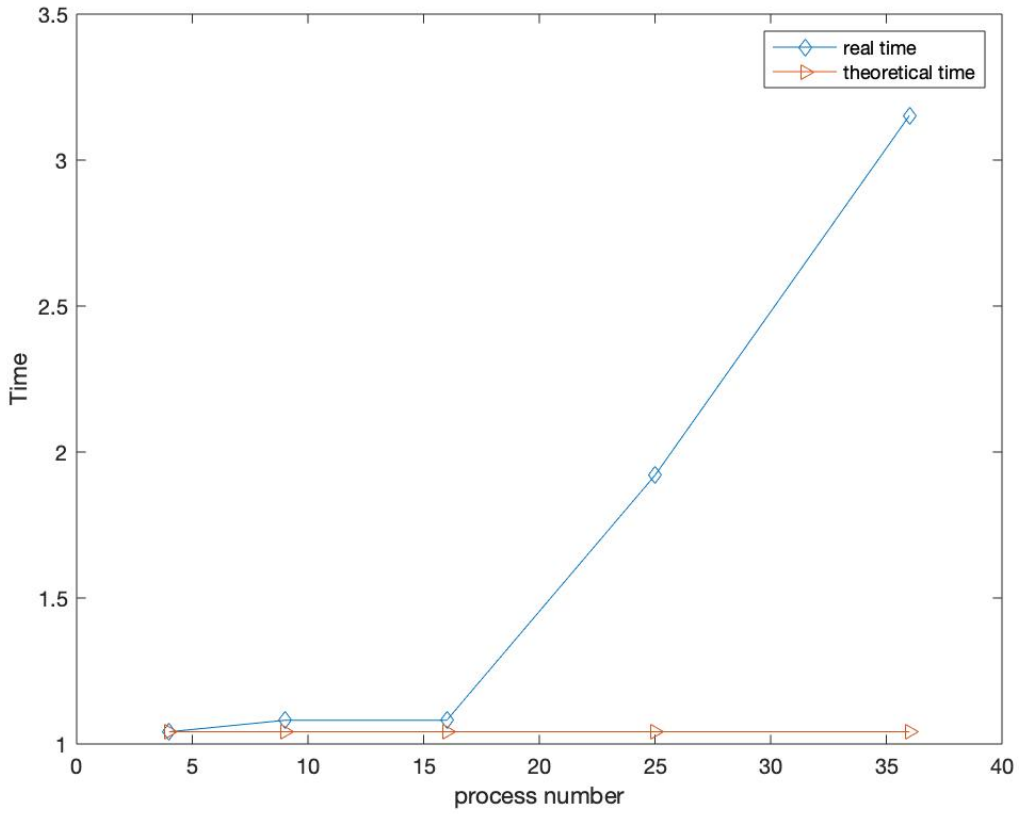

Figure 1: real and theoretical time on a plot for the weak scaling study

Compute the send displacement and the bucket size for each process, use **MPI_Alltoall** to exchange the bucket size.

```
MPI_Alltoall(bucket_size_each_process, 1, MPI_INT,receive_bucket_size, 1,
MPI_INT, MPI_COMM_WORLD);
```

So now each process now how many number it's going to receive and what the receive displacement is. Now we have all the information we need to do the **MPI_Alltoallv**.

```
MPI_Alltoallv(vec, bucket_size_each_process, send_displacement,
MPI_INT, bucket, receive_bucket_size,
receive_displacement, MPI_INT, MPI_COMM_WORLD);
```

Finally, do the local sort of the bucket and output to the file.

3.2 result

Table 3 shows the timing for N = 1e4, 1e5 on 64 nodes. I failed to test $N = 1e6$ because it takes so long and so many node it canceled by prince.

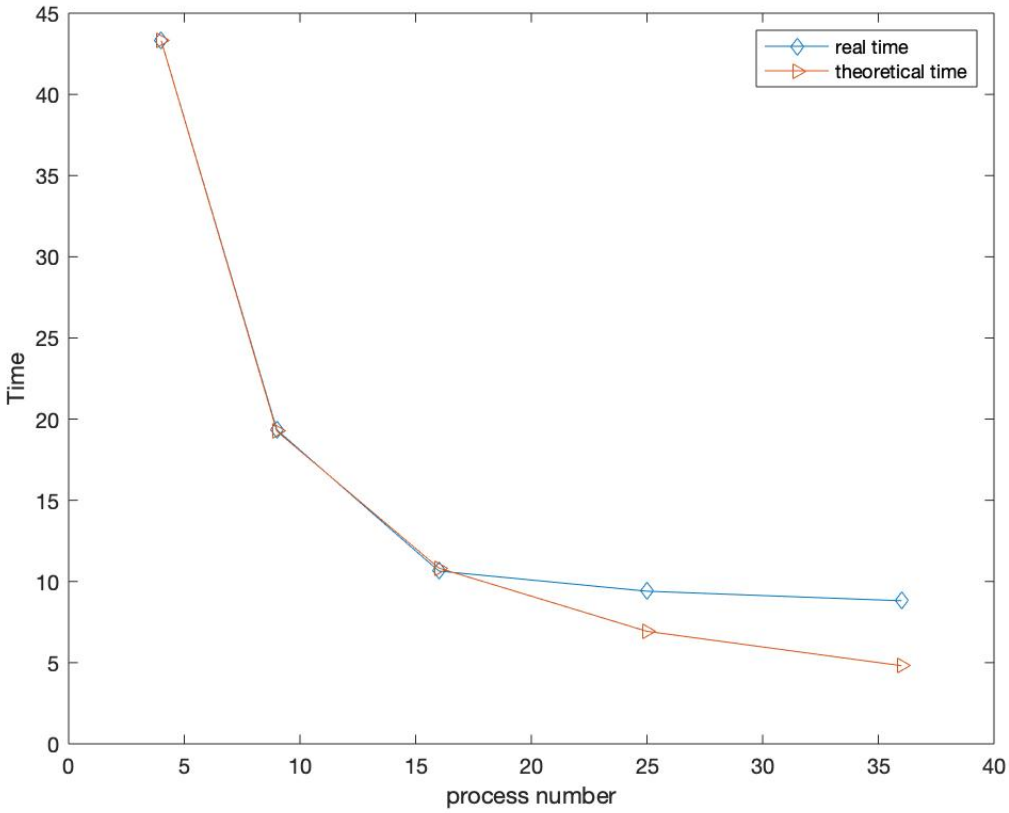

Figure 2: real and theoretical time on a plot for the strong scaling study

Length of the array per process	Time
N = 10000	4.571116s
N = 100000	21.057605s

Table 3: Running time for parallel sample sort with 64 process and each process start with an array of size N

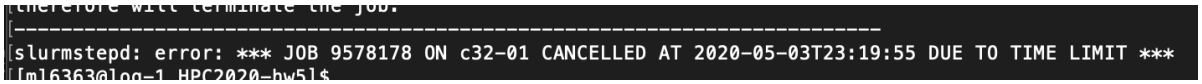

Figure 3: For N = 1e6, Prince cancelled my task due to the time limit.