DD2356 High Performance Computing

Yuchen Gao,Yuze Cui

May 17, 2022

Introduction

Code repository link:[1] This is the report for DD2356 High Performance Computing assignment3, MPI programming

Questions

Exercise1

Q1

The code is in github called q1.c

Q2

Compile: cc q1.c -o q1

Q3

srun -n 4./q1

Q4

By changing the size variable of processors we use in 'srun command'.

Q5

- LAM-MPI;
- MPICH;

[1] https://gits-15.sys.kth.se/yuchenga/DD2356Assignment3

Q6

I think the implementations below are the most used:

- MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);
- MPI_Finalize();

Since the creation of MPI processes requires those two commands.

Exercise2

Q1

The code can be seen as 2.c

Q2

To check to results, we can firstly assign an number of error tolerance because "dx" is using to calculate function f and the first order derivative. To check the domain edge derivative, we can compare the edge of array dfdx with the cos of the edge of the input array. To check the ghost cell, we can compare the beginning cell of each process with the sin() whose input is the last number of previous process. And comparing the last cell of each process with the sin() whose input is the first number of next process. Code fragment can be seen below:

```
if (fabs(f[0]-sin(L_loc*rank-dx))>error || fabs(f[nxn_loc-1]-sin(L_loc*(rank+1)+dx))>error) {
    printf("Check NOT OK in rank %d\n",rank);
}
else printf("Check OK in rank %d \n", rank);

if(rank==0){
  if(fabs(dfdx[1]-cos(0))>error){
    printf("Left derivative check NOT OK in rank \n");
  }
  else printf("Left derivative check OK in rank \n");
}

if(rank==size-1){
  if(fabs(dfdx[nxn_loc-2]-cos(L))>error){
    printf("Right derivative check NOT OK in rank \n");
  }
  else printf("Right derivative check OK in rank \n");
}
```

Figure 1: Codes for checking

Q3

Because these two functions don't return anything until the the send-recv communication is done (block). Not finished communication may stall the system. And the send-recv communication block other communicators. Blocking communication is easy to use. And the buffer to send and receive can be reused due

to its block characteristics.

Exercise3

Q1

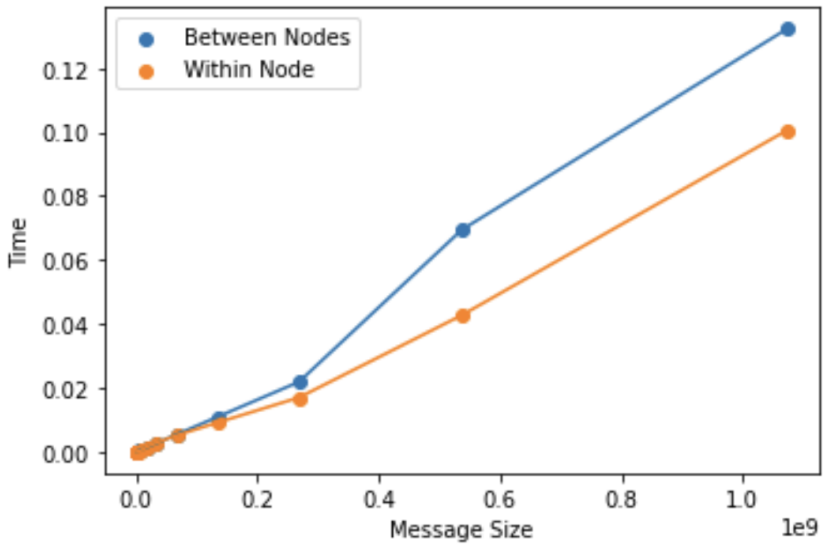

Figure 2:

Q2

The out node bandwidth I acquire is 1/1.22696714e-10 ; The time I got is -6.30487782e-04s, according to the tutorial, 1.6e-6s will be used

The intra node bandwidth I acquire is 1/8.99696586e-11; The time I got is -4.69959142e-04.

Q3

In file RMA.c.

Q4

The two-side communication requires the synchronization between two processes, if the receiving process is delayed, the sending process will also be delayed too.

The one-side communication requires no synchronization between processes. Each process exposes part of its memory to others so that the remote data movements require no synchronization.

Q5

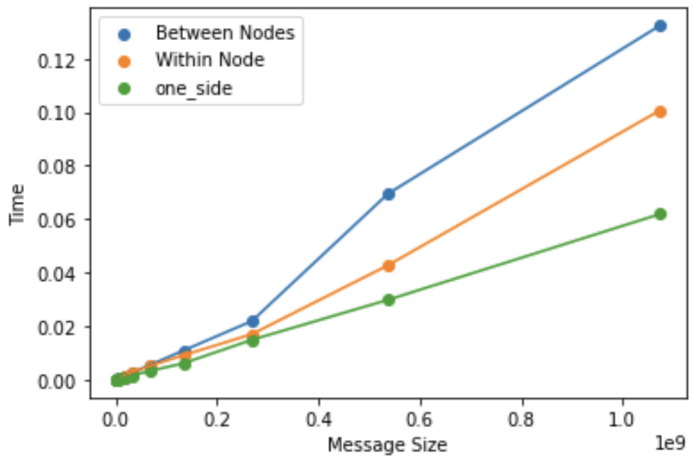

Figure 3:

The bandwidth I got from the one-side communication is 1/5.71463499e-11, time is -1.20096268e-04. The bandwidth is larger than the point-to-point communication, and this performance model is more liner related comparing with former ones. This means the one-side generates less overhead, and is more accurate to fit the bandwidth performance.

Q6

1.The problem with postal model is that it does not account for the interface between the nodes. In addition, the network can not sustain communication at a rate of a single process node. Another problem is that MPI implementations typically use different methods depending on the message length. So the bandwidth varies when different message are passed. 2.The proposed model is refined in two ways:

- Recognize the eager/rendezvous threshold;
- Take into account the limits of bandwidth into and out of a node.

The model is modified to be :

$T = \alpha + kn/min(R_N, kR_c)$.
A refined model is :

$T = \alpha + kn/min(R_N, R_{Cb} + (k-1)R_{Ci})$.

Exercise4

4.1

Q1 The code can be seen in 4.1.c

Q2 Performance is measured below:

MPI processes	_Execution time(s)
8	3.141124
16	1.734064
32	1.319861
64	0.700918
128	0.320417
256	0.344016

Table 1: Performance:Sequential

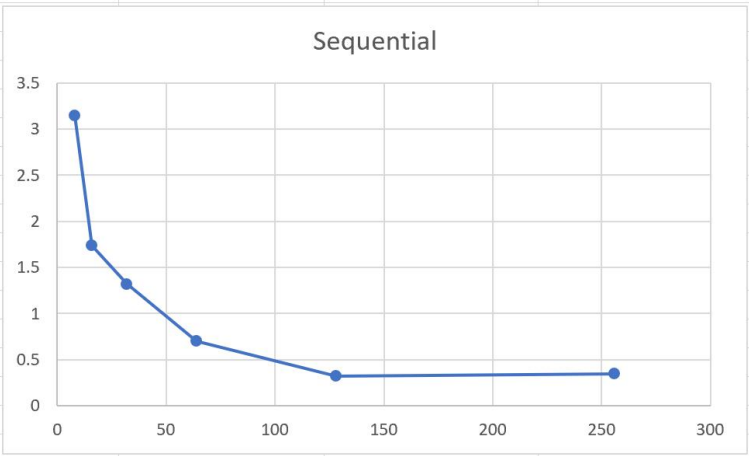

Figure 4: Performance measured using sequential algorithm

The performance scales with the number of processes very quickly at the beginning. When the number of MPI processes increased above 64, the improvement of performance by increasing process number is slower. It is non-linear. The timing function is MPI_Wtime(), we can get the start and end return value

to get the operation time.

Q3 According to what we obtained in Q2 in previous exercise, the bandwidth is 12.2696714 GB/s (1.23e10 byte/s) and latency is 0.00046996 second.
We can divide the analyzation into communication part and calculation part. Using postal communication model, consider the process size is n, there are n-1 processes sends data and 1 process receive. And sizeof(int)=4. So the time of communication time is $T_{comm} = (n-1) \times (latency + \frac{4}{bandwidth})$
Consider the computation after timer starts, we have a calculation below:

execution	number of computation
Generate x	1*iter_perth(iteration per process=NUM_ITER/n) division
Generate x	1*iter_perth division
Generate y	2*iter_perth multiplication, 1*iter_perth add and 1*iter_perth sqrt
calculate count	(pi/4)*iter_perth add
combination of results	(n-1)*add
calculation of pi	1 division+1 multiplication

Table 2: Performance:Sequential

We usually count addition, subtraction, multiplication, division, exponentiation, square root, etc. as a single FLOP.
So, $T_{cal} = (\frac{NUM\_ITER}{n} \times (6 + \frac{pi}{4}) + (n-1) \times 1 + 2) \times (\frac{1}{CPUFreq})$
The whole model is$T_{model} = (n-1) \times (latency + \frac{4}{bandwidth}) + (\frac{NUM_ITER}{n} \times (6 + \frac{pi}{4}) + (n-1) \times 1 + 2) \times (\frac{1}{CPUFreq})$
We can calculate and compare the result:

MPI processes	Model predicted Execution time(s)	Measured Execution time(s)
8	0.380256	3.141124
16	0.195532	1.734064
32	0.108810	1.319861
64	0.076728	0.700918
128	0.083245	0.320417
256	0.131620	0.344016

Table 3:

We can see that the predicted results is smaller than the time measured. I think that is because creating process has overhead and the operations such as loop and assignment has its computation time, the division and multiply

operation cannot end in one clock cycle. When number of process increases, the communication time grows linearly and computation time decreases a bit slower, which leads to measured time to have a trend below:

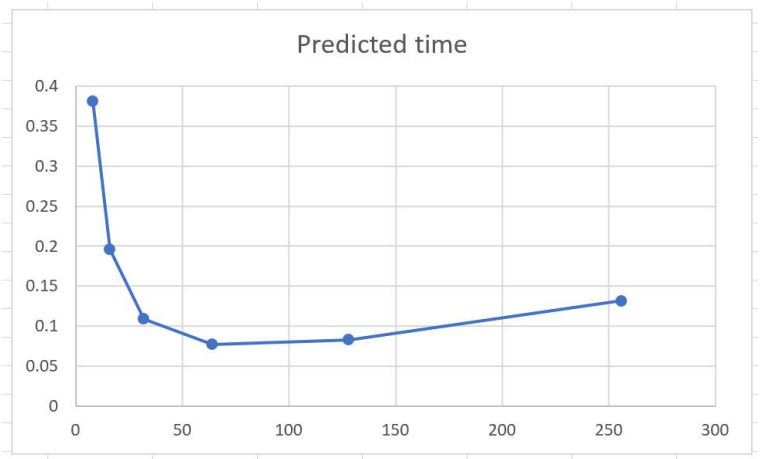

Figure 5:

4.2

Q1 The code is included in 4.2.c

Q2 The performance is measured below:

MPI processes	_Execution time(s)
8	3.028874
16	1.700965
32	1.117549
64	0.57675
128	0.313077
256	0.343951

Table 4: Performance:Tree

Figure 6: Performance measured using tree algorithm

Q3 As in exe 1,we can give a similar model but we can parallelize the communication due to tree reduction(there is no conflict when they are communicating with different process):

$T_{model} = log_2(n) \times (latency + \frac{4}{bandwidth}) + (\frac{NUM\_ITER}{n} \times (6 + \frac{pi}{4}) + (n - 1) \times 1 + 2) \times (\frac{1}{CPUFreq})$

MPI processes	Model predicted Execution time(s)	Measured Execution time(s)
8	0.378376	3.141124
16	0.190363	1.734064
32	0.096591	1.319861
64	0.049940	0.700918
128	0.026850	0.320417
256	0.015540	0.344016

Using tree reduction, the communication time is reduced significantly.

Q4 We can plot a graph to make a cpmarison:

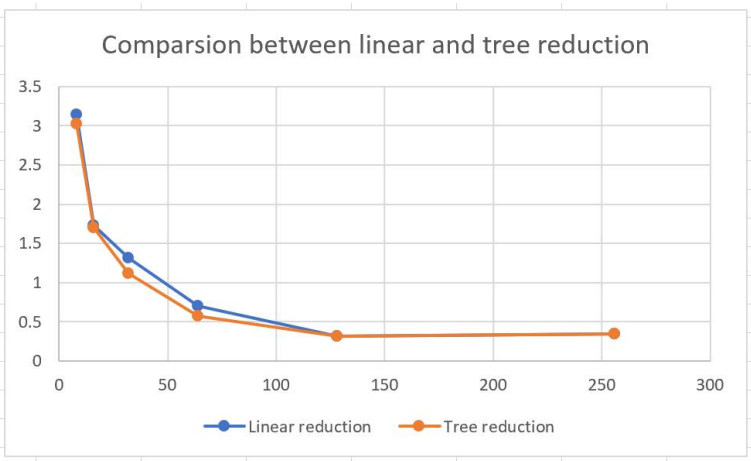

Figure 7: Comparsion between linear and tree reduction

We can concluded that tree reduction algorithm performs really better than linear reduction. In 256 processes, the performance is nearly same.

Q5 If using 10000 process, the modeled result is 0.0065507(Tree) and 4.69944(Linear). The tree reduction will perform better because their calculation time is the same but tree reduction will decrease the communication time, where this cost in linear reduction grows linearly but tree reduction has logarithmic growth.

4.3

Q1 The code is included in 4.3.c

Q2 The performace is below:

MPI processes	Execution time(s)
8	2.985004
16	1.660719
32	1.027869
64	0.55623
128	0.318101
256	0.348336

Table 5: Performance:MPI$_I recv$

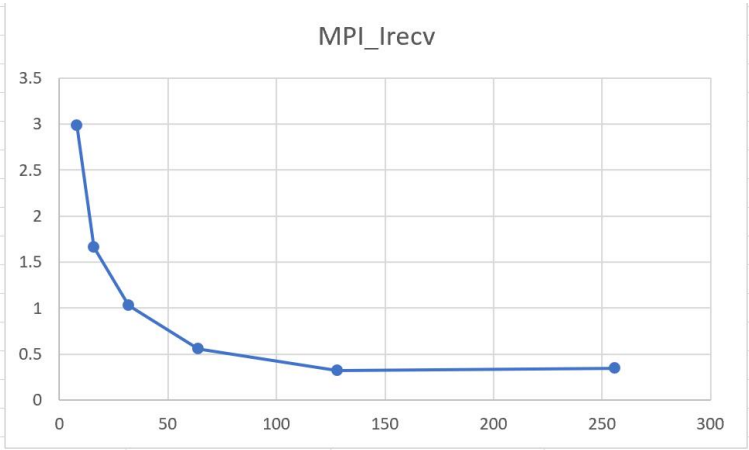

Figure 8: Performance measured using non-blocking receiving

Q3 We use MPI_Isend, MPI_Irecv, MPI_Irsend and so on to operate non-blocking communication, often with the help of MPI_Wait, MPI_Waitall to finalize.
The "I" means immediate return. it does not block until the message is sent or received.

Q4 Compared with the results measrued in 4.1. The performance is better(but not too much better) in different number of MPI processes.

4.4

Q1 The code is included in 4.4.c

Q2 The performance is measured below:

MPI processes	_Execution time(s)
8	2.880853
16	1.662115
32	1.111481
64	0.587167
128	0.377924
256	0.405988

Table 6: Performance:Reduced

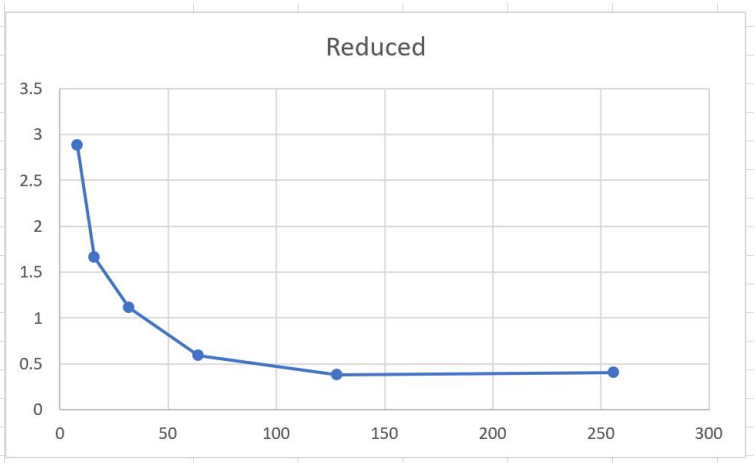

Figure 9: Performance measured using MPI_*reduceoperation*

Bonus Exercise

Q1

In bonus_fox.c

Q2

I reused the matrix calculation function to calculate the matrix multiplication on one processor and compare the result with the calculation using fox algorithm. If the difference between each element is less than 1e-6, then the answer is correct.

```
void normal_cal(double *a,double *b,double *c,double *MatrixC,int n)
{
  clock_t start, finish;
  double duration;

  start = clock();
  for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
      for(int k=0;k<n;k++){
        c[n*i+j]+=a[n*i+k]*b[n*k+j];
        if(c[n*i+j]-MatrixC[n*i+j]>1e-6 || MatrixC[n*i+j]-c[n*i+j]>1e-6 ){
          printf("wrong");
        }
      }
    }
  }
  printf("correct\n");
 // finish= clock();
  //duration = (double)(finish- start) / CLOCKS_PER_SEC;
//  return duration;
}
```

Figure 10: Codes for checking

Q3

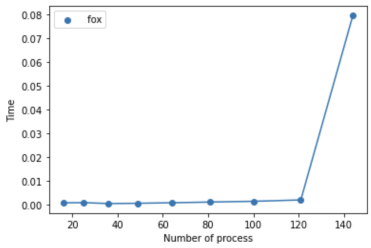

Figure 11: The calculation time for 0-144 processes

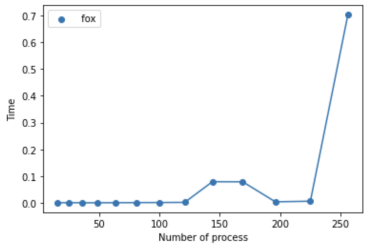

Figure 12: The calculation time for 0-256 processes

Q4

I fitted the performance curve using the np.polyfit function using 6-order polynomial.

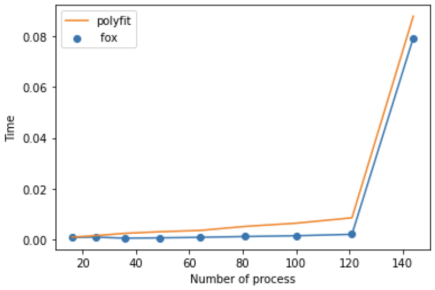

Figure 13: Fitting calculation time for 0-144 processes

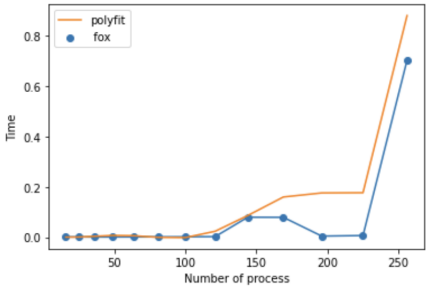

Figure 14: Fitting for calculation time for 0-256 processes

The result shows the calculation time grows rapidly when the matrix size is large enough. Figure5 shows the fitting between 16-144 processes can be accurate, the performance when processes equal to 196 and 225 is really good but when the size grows to 256, the time is growing rapidly. I guess maybe there is only 15 processes one node can allocate, when the size is larger than this number, the performance degrades.