455303 Parallel Programming 2014
Assignment: Computing the two-point angular correlation function
Student name: Vu Nguyen
Student number: 71141

1. Decomposition of the problem

Based on the sequential implementation of the problem, we can see that the parts of the code that can be parallelized are for loops used for calculating histograms DD, DR, and RR using function add_histogram, and they take quite a significant amount of time. Therefore, the parallel implementation will concentrate on these heavy tasks.

Calculation of histograms DD and RR is done in the same way, while calculation of histogram DR is a bit different. Therefore, two data decomposition methods are applied for calculations of histogram DR and of histograms DD and RR. Data composition of histograms DD and RR are described as follows.

P0	P1	P2	…	…	…	…	P1	P0	

0	1	2	3	4	….	…	n-3	n-2	n-1
1	2	3	4	5	…	…	n-2	n-1	
2	3	4	5	6	…	…	n-1		
3	4	5	…	…	…	…			
4	5	…	…	n-2	…				
…	…	…	n-2	n-1					
….	…	n-2	n-1						
n-3	n-2	n-1							
n-2	n-1								
n-1									

n-1 n-2 n-3 n-4 n-5 …. …. 2 1

Figure 1. Iterations of loop1 vs iterations of loop2 for calculation of histograms DD and RR

The number of lines read from the input data file is n. To calculate histograms DD and RR, a loop called loop1 goes from 0 to n-1. The green row in Figure 1 illustrates the iterations of loop1, the vertical columns show the iterations of loop2. With each iteration of loop1, there is a nested loop inside it called loop2. The number of iterations of loop2 decreases by 1 as loop1 moves forwards by 1. For the first iteration of loop1, the number of iterations of loop2 is n-1. For the second iteration of loop1, the number of iterations of loop2 is n-2. It can be seen that there is an unbalanced number of iterations of loop2 for each iteration of loop1. The solution is to pair the iterations of loop1 with each other and assign that pair to a particular processor. To make sure that the number of iterations is distributed evenly to the processors, the first iteration of loop1 is paired to the (n-2)th iteration of loop1. The second iteration of loop1 is paired to the (n-3)th iteration of loop1 and so on. Therefore, the total number of iterations for each pair is always n. If there is an odd number of iterations in loop1, the middle iteration of loop1 will be assigned to processor 0. Regarding the assignment of each pair of iterations in loop1 to each processor, each pair will be assigned to each processor consecutively. For example, process 0 is assigned to pair with iterations 0th and (n-2)th, processor 1 is assigned to pair with iterations 1th and (n-3)th, and so on until the middle iteration of loop1 is reached.

Now data decomposition for calculating histogram DR is discussed.

P0	P1	….				

0	1	2	3	4	…	…	n-3	n-2	n-1
0	0	0	0	0	…	…	0	0	0
1	1	1	1	1	…	…	1	1	1
2	2	2	2	2	…	…	2	2	2
3	3	3	3	3	…	…	3	3	3
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
m-3	m-3	m-3	m-3	m-3	m-3	m-3	m-3	m-3	m-3
m-2	m-2	m-2	m-2	m-2	m-2	m-2	m-2	m-2	m-2
m-1	m-1	m-1	m-1	m-1	m-1	m-1	m-1	m-1	m-1
m	m	m	m	m	m	m	m	m	m

Figure 2. Iterations of loop1 vs iterations of loop2 for calculation of histogram DR

Calculation of histogram DR is based on a nested loop called loop2 inside the main loop called loop1. Loop1 goes from 0 to n-1, where n is the number of lines read from the real data file. Loop2 goes from 0 to m-1, where m is the number of lines read from the random data file. It can be seen that each iteration of loop1 needs m iterations of loop1. From that reasoning, we can divide a group of iterations of loop1 to each processor. For example, processor 0 takes care of iterations from 0 to 2 in loop1, while process 1 takes care of iterations from 3 to 5 in loop1 and so on.

2. Implementation

In the implementation phase, the loop iterations are considered as elements since it can be understood easier that way (like an array)

Variables

```
#define binsperdegree 4                 /* Nr of bins per degree */
#define totaldegrees 64                 /* Nr of degrees */

int         rank,                       // Process id
            size,                       // Number of processes
            err,                        // error return value
            first_index,                // Start index assigned to a process in data composition
            last_index,                 // End index assigned to a process in data composition
            center_index,               // Center index of an array in data composition
            nmin,                       // Number of elements per process
            nleft,                      // Number of elements left over
            first,                      // Index of first element handled by this process
            M;                          // Number of elements handled by this process

double startTime, endTime;              // Start time and end time for measuring execution time
int Nooflines_Real;                     // Nr of lines in real data
int Nooflines_Sim;                      // Nr of lines in random data
float *xd_real, *yd_real, *zd_real;     // Arrays for real data
```

```
float *xd_sim , *yd_sim , *zd_sim;                                  // Arrays for random data
long int *histogramDD, *histogramDR, *histogramRR;                  // Arrays for histograms
long int *histogramDD_tmp, *histogramDR_tmp, *histogramRR_tmp;      // Temporary arrays for
                                                                    histograms
FILE *infile_real, *infile_sim, *outfile;
int nr_of_bins = binsperdegree * totaldegrees;          // Total number of intervals/bins
int i, j;
float pi, costotaldegrees;                              // Value of PI and cosine of totaldegrees
long int TotalCountDD, TotalCountRR, TotalCountDR;
double NSimdivNReal, w;
```

Reading data files

-All processes open the real data file and the random data file.
-Process 0 count how many lines the real data file and the random data file contain and broadcast these lines to the rest of the processes.

Allocate arrays

Arrays for x, y, and z real values

0	1	2	3	4	4	5	…	….	Nooflines_Real -1

Arrays for x, y, and z random values

0	1	2	3	4	4	5	…	….	Nooflines_Sim -1

Arrays for histogram DD, DR, and RR and temporary histograms DD, DR, and RR

0	1	2	3	4	4	5	…	….	nr_of_bins

-All processes allocate arrays for x, y, and z real and random values. Three arrays of size Nooflines_Real are allocated by all processes. Three arrays of size Nooflines_Sim are allocated by all processes.

-All processes allocate 3 arrays for histograms DD, DR, and RR with the size of nr_of_bins+1.
-All processes allocate 3 arrays for temporary histograms DD, DR, and RR with the size of nr_of_bins+1. These 3 arrays for calculation of histograms DD, DR, and RR are used for MPI_Reduce later on.

Calculate histograms DD and RR

-Each process figures out the middle iteration of loop1 using the following formulas
center_index = (Nooflines_Real – 1)/2
center_index = (Nooflines_Sim – 1)/2

-Allocating iteration number of loop1 to each process is done using the following for loop
for (i = rank; i < center_index; i += size);
where rank the the process id, size is the number of processes

The loop repeats until the first half of loop1 is reached.

-After each process is allocated its share of the iterations, each of them will calculate the histograms pair by pair.
The first member of the pair is determined as
first_index = rank, rank+size, rank+2size, ….

The second member of the pair is determined as
last_index = Nooflines_Real/Sim – 2 – rank, Nooflines_Real/Sim – 2 – size, Noflines_Real/Sim – 2 – 2size, ….

-Since only (i,j) pairs are calculated, we need to multiply each histogram element with 2. Every process does this step.
- For (j,j) pair, process 0 takes care of this calculation.

Calculate histogram DR

-Since the number of iterations of loop1 cannot always be evenly distribute among the processes, some processes may get one iteration from loop1 more than the other. The remaining iteration is assigned to the process which is smaller than the iteration number (index).

```
nmin = Nooflines_Real/size;                 // Number of indices per process
nleft = Nooflines_Real%size;                // Number of indices left over

if (rank < nleft) M = nmin + 1;
else M = nmin;

first = rank * nmin + min(rank, nleft); // Count which iteration number in this process is the first

-Calculate histogram DR
for (i = first; i < first+M; i++) {
   for (j = 0; j < Nooflines_Sim; j++)
      add_histogram (xd_real[i], yd_real[i], zd_real[i], xd_sim[j], yd_sim[j], zd_sim[j],
                     histogramDR, pi, costotaldegrees);
}
```

Gathering data

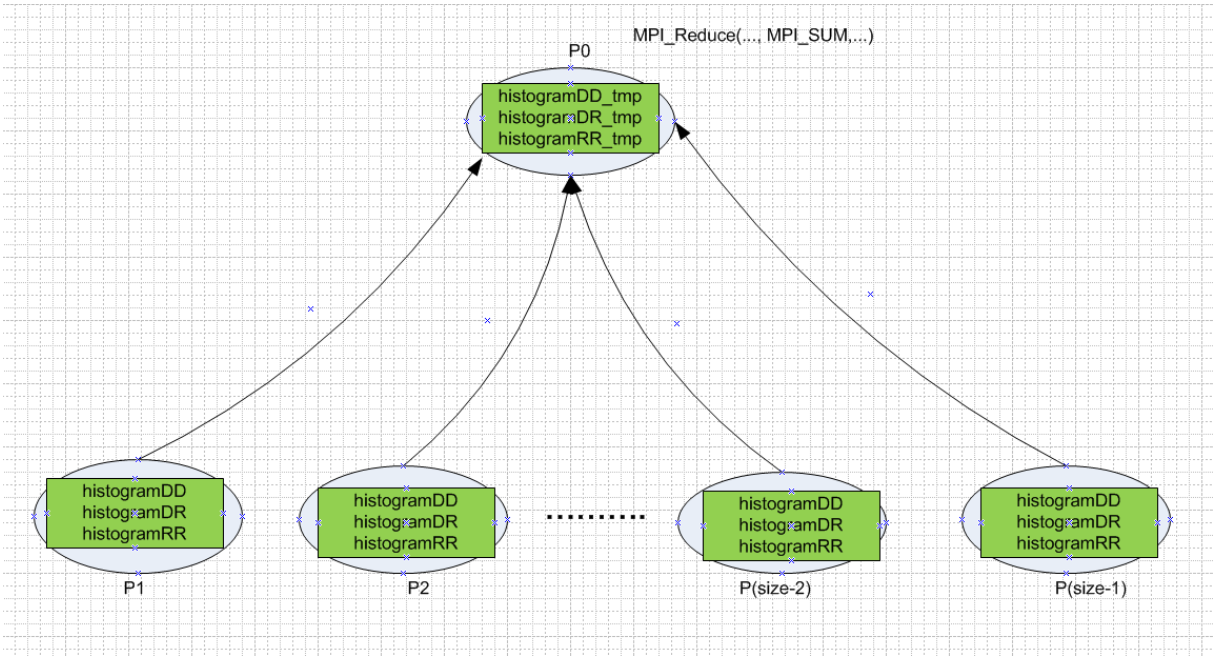

Figure 3. Summing of arrays from histograms DD, DR, and RR

-All processes contain their shares of the histograms DD, DR, and RR. MPI_Reduce with MPI_SUM operation is used to sum up all the arrays.

```
MPI_Reduce(histogramDD, histogramDD_tmp, nr_of_bins+1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(histogramDR, histogramDR_tmp, nr_of_bins+1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(histogramRR, histogramRR_tmp, nr_of_bins+1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
```

3. Results and scalability

The parallel program can be scaled from 2 to N processors. From the test results, the program can be run any number of cores ranging from 2 to 48 (the maximum number of cores in the cluster). The following tables below concern any core numbers between 2 and 48. Execution time of sequential code is listed in the tables for comparison purpose.

Result of running the parallel program galaxyz_p with small data files and various numbers of cores

Number of cores	Execution time(s)	Speedup
1	21.7	1
2	11.6	1.87069
5	6.8	3.191176
10	3.5	6.2
15	2.3	9.434783
20	1.8	12.05556
30	1.2	18.08333
35	1.1	19.72727
40	0.9	24.11111
45	0.8	27.125
47	0.8	27.125
48	0.8	27.125