COMP90025 Parallel and Multicore Computing
Project 1B-Mendelbrot Set

Lu Chen Yiru Pan

Introduction

The Mandelbrot Set is a set of points in the complex plane. A complex number $C(a+bi)$ belongs to the Mandelbrot Set if the function $f_c(z) = \mathrm{z}^2 + C$ will not diverge when iterating z from z_0. In this project, we are given the original sequential algorithm, which aims to count the number of points in each input region that belong to the Mandelbrot Set. Our objective is to optimize this sequential algorithm by leveraging OpenMPI and OpenMP on the Spartan cluster.

Cardioid and Bulb Checking

At the very beginning, we explored the possibility of a faster way to check if a point is within the Mandelbrot set, instead of iterating the function to see if the result will diverge.

As shown in figure-1, Mandelbrot set consists of multiple bulbs with different sizes. The largest region in the centre (number 1) is cardioid-shaped. Studies show that this region is made up of all $C(a+bi)$ for which $\mathrm{P_c}$ ($z \to z^2 + C$) has an attracting fixed point. These C are called period-1 points and they satisfy the equation $f_c(z) = z$. Next to the cardioid region is a bulb (number 2). This region consists of all C which satisfy the equation $f_c(f_c(z)) = z$. It is called the period-2 bulb. Then we can find the period-3 bulb ($f_c(f_c(f_c(z))) = z$), period-4 bulb $(f_c(f_c(f_c(f_c(z)))) = z)$, etc.

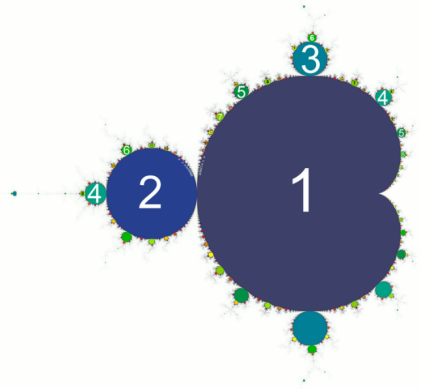

Figure-1 Mandelbrot Set

It has been demonstrated that equation (1) and (2) can be used to check whether the point is inside the centre cardioid and equation (3) for the period-2 bulb. Among them, x and y represent the real value and the imaginary value of C. [1]

(1) $\mathrm{p} = \sqrt{\left(x - \frac{1}{4}\right)^2 + y^2}$

(2) $x < p - 2p^2 + \frac{1}{4}$

(3) $(x+1)^2 + y^2 < \frac{1}{16}$

Thus, we decided to implement cardioid and bulb check in our program to first check if C is within the Mandelbrot set. If not, we continue to check using multiple iterations.

```
int inset(double real, double img, int maxiter){
   //Add cardioid and bulb check
double img2 = img*img;
 double check1 = (real-0.25)*(real-0.25) +
img2;
 double check2 = 4*check1*(real-
0.25+check1);
double check3 = (real+1)*(real+1)+img2;
if((check2<=img2)||(check3<0.0625))
return 1;
.....
}
```

OpenMPI Strategies

In order to achieve the maximum speedup of the program, we attempted two different scatter strategies - region scatter and uniform scatter.

- Region Scatter

With region scatter, each input region is assigned to one node and this node will count all valid points within this region. In the for-loop in main function, regions are evenly divided according to the node's rank-number.

Code block is as follows:

```
for (int region=rank; region<num_regions;
region+=size){
// scan the arguments
sscanf(argv[region*6+1],"%lf",&real_lower);
sscanf(argv[region*6+2],"%lf",&real_upper);
sscanf(argv[region*6+3],"%lf",&img_lower);
sscanf(argv[region*6+4],"%lf",&img_upper);
sscanf(argv[region*6+5],"%i",&num);
 sscanf(argv[region*6+6],"%i",&maxiter);
int localCount = mandelbrotSetCount
(real_lower, real_upper, img_lower,
img_upper, num,maxiter);
double end = MPI_Wtime() …;
..
```

Within each node, OpenMP is used to maximize efficiency. Through project 1a, we have concluded that single parallel-for using dynamic model can achieve the best performance for both single loops and nested loops. We decided to apply the same strategy into this project whenever the for-loop(s) can be parallelized.

In the function mandelbrotSetCount(), OpenMP reduction clause is also used. Therefore, each thread can have a separate copy of the variable 'count'. The initial value of 'count' in each thread is 0. When the loop is entirely done, the value of 'count' from each thread will be summed up. This atomic operation ensures the correctness of result and prevent write collision.

```
//count the number of points in the set, within
the region
int mandelbrotSetCount(double real_lower,
double real_upper, double img_lower, double
img_upper, int num, int maxiter){
int count=0;
double real_step = (real_upper-real_lower)/num;
double img_step = (img_upper-img_lower)/num;
#pragma omp parallel for schedule(dynamic)
reduction(+: count)
for(int real=0; real<num; real++){
for(int img=0; img<num; img++){
count+=inset(real_lower+real*real_step,img_lo
wer+img*img_step,maxiter);}}
return count;
}
```

We tested this strategy with 4 input regions using different point numbers (from 2000 to 8000) and different node numbers (from 1 to 4). Input parameters are as below:
-2.0 1.0 -1.0 1.0 8000 10000 -1 1.0 0.0 1.0 8000 10000-2.0 1.0 -1.0 1.0 8000 10000 -1 1.0 0.0 1.0 8000 10000

	region scatter running time(s)			
node num\point num	2000	4000	6000	8000
1	6.814	51.69	115.668	205.333
2	4.939	17.773	39.07	69.233
4	1.33	3.186	6.59	10.847

From Figure-2, it's obvious that 4 nodes achieved the maximum speedup. Even if there is a communication cost between nodes, it's trivial in this situation. With the number of points increasing, assigning task to multiple nodes is more efficient than assigning to single node. Take input size 8000 as example, the speedup of using 4 nodes is $\frac{T_s}{T_p} = \frac{205.333}{10.847} = 18.9$, which improves efficiency significantly.

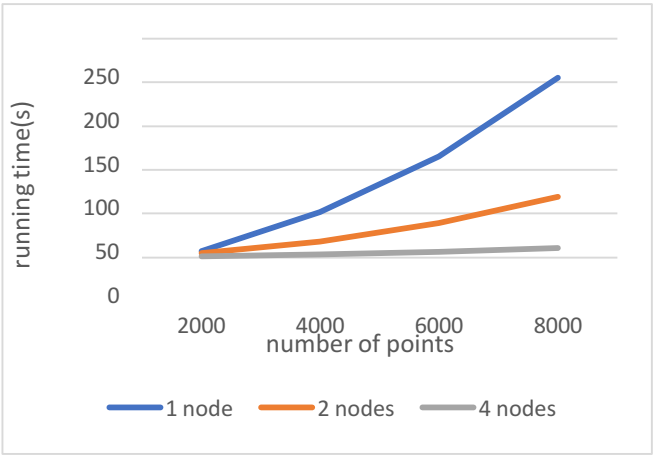

Figure 2- Region Scatter

- Uniform Scatter

Uniform scatter strategy is to distribute the workload in the mandelbrotSetCount() function. All the nodes will read in all the input regions. It is the points in each region that are assigned evenly to each node for checking and counting. After each node finishes its task, MPI_Reduce is used in the main function, which sums up all the local counts to a global count in each region.

```
for(int region=0;region<num_regions;region++){
sscanf(argv[region*6+1],"%lf",&real_lower);
sscanf(argv[region*6+2],"%lf",&real_upper);
sscanf(argv[region*6+3],"%lf",&img_lower);
sscanf(argv[region*6+4],"%lf",&img_upper);
sscanf(argv[region*6+5],"%i",&num);
sscanf(argv[region*6+6],"%i",&maxiter);
int mandelbrotCount =
mandelbrotSetCount(real_lower,real_upper,img_l
ower,img_upper,num,maxiter);
int madelbrotCountSum = 0;
MPI_Reduce(&mandelbrotCount,
&madelbrotCountSum, 1, MPI_INT,
MPI_SUM,0,MPI_COMM_WORLD);
double end = MPI_Wtime();
printf("Count is %d, run time is %f, by node
%d\n", mandelbrotCount, end-start, rank);
if(rank==0){ printf("%d\n",madelbrotCountSum);
}
}
```

```
// count the number of points in the set, within
the region
int mandelbrotSetCount(double real_lower,
double real_upper, double img_lower, double
img_upper, int num, int maxiter){
int count=0;
double real_step = (real_upper-real_lower)/num;
double img_step = (img_upper-img_lower)/num;
int real,img;
#pragma omp parallel for schedule(dynamic)
reduction(+: count)
for(real=rank; real<num; real+=size){
for(img=0; img<num; img++){
count+=inset(real_lower+real*real_step,img_low
er+img*img_step,maxiter);}}
return count;
}
```

This strategy was tested with 2 input regions, on different point numbers (from 2000 to 8000) on different node numbers (from 1 to 4). Input parameters are as below:
-2.0 1.0 -1.0 1.0 8000 10000 -1 1.0 0.0 1.0 8000 10000

The test result is as below:

	uniform scatter running time(s)			
node num\point num	2000	4000	6000	8000
1	7.892	31.058	57.832	102.595
2	4.572	16.029	36.11	62.291
4	1.012	3.041	6.111	10.664

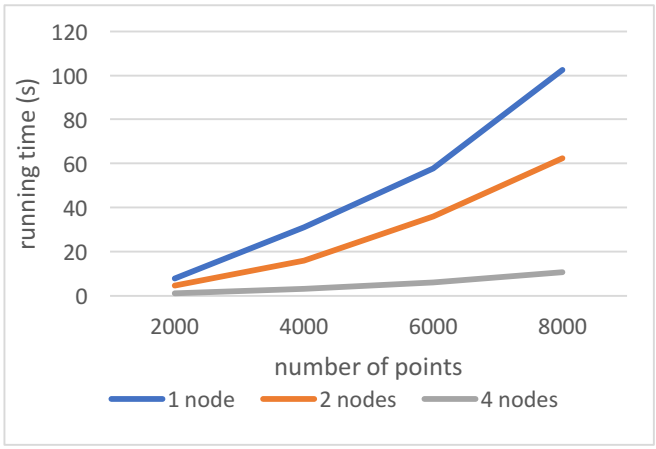

Figure 3- Uniform Scatter

From the chart, speedup of 4 nodes outperforms others. Take 8000 input size as example, the speedup of using 4 nodes is $\frac{T_s}{T_p} = \frac{102.595}{10.664} = 9.62$.

Comparisons Between Two Strategies

The main differences between the two strategies are that the region scatter assigns all input regions across different nodes, whilst the uniform scatter assigns all points across different nodes. Both strategies have their pros and cons. The efficiency of region scatter is easily influenced by the number of inputs. If the input region number cannot be divided by the node number, some nodes will have more workload than others. For example, there are 5 input regions and 4 nodes, node 0 will handle 2 inputs whilst others only handle 1 input, which will decrease the efficiency. By contrast, work distribution in uniform scatter is more even. However, uniform scatter will deal with more overheads because each node must read and handle all input regions. Also, the MPI_Reduce communication will consume some time.

We implemented comparison experiments in five scenarios: 2 inputs with 2 nodes on 2000-8000 numbers, 4 - 7 inputs with 4 nodes on 2000-8000 numbers. Here are the results.

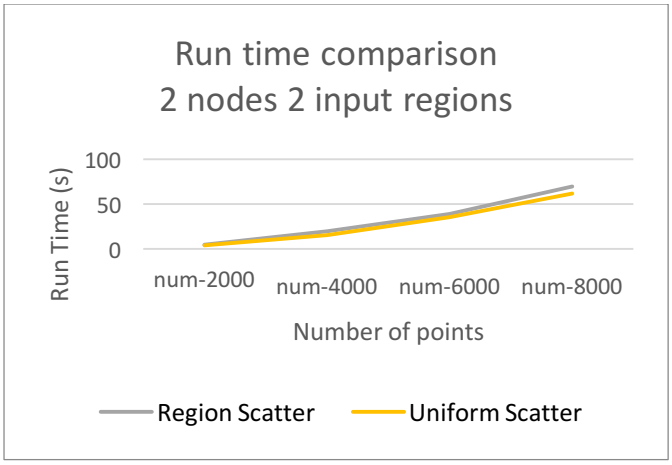

Figure-4. 2 nodes 2 input regions

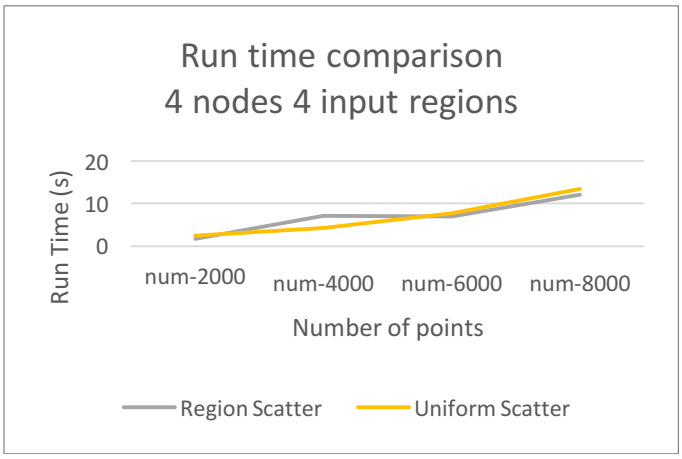

Figure-5. 4 nodes 4 input regions

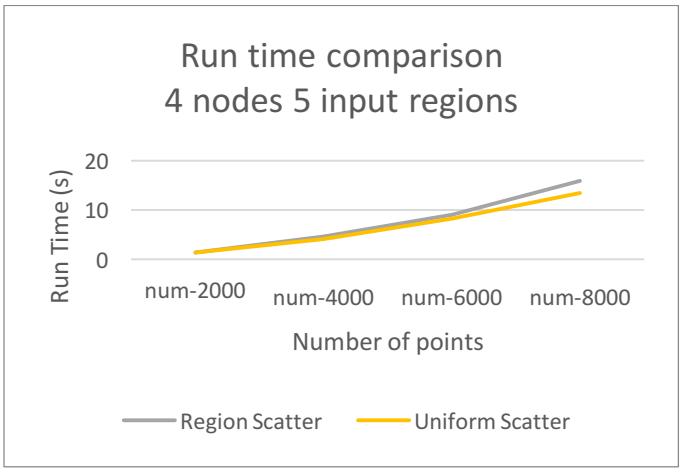

Figure 6. 4 nodes 5 input regions

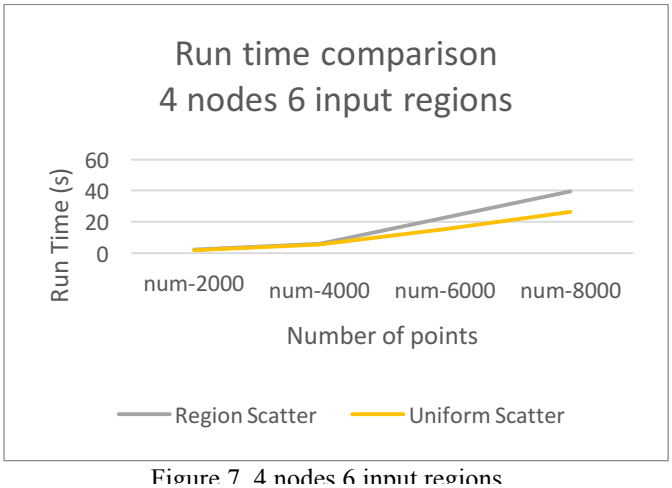

Figure 7. 4 nodes 6 input regions

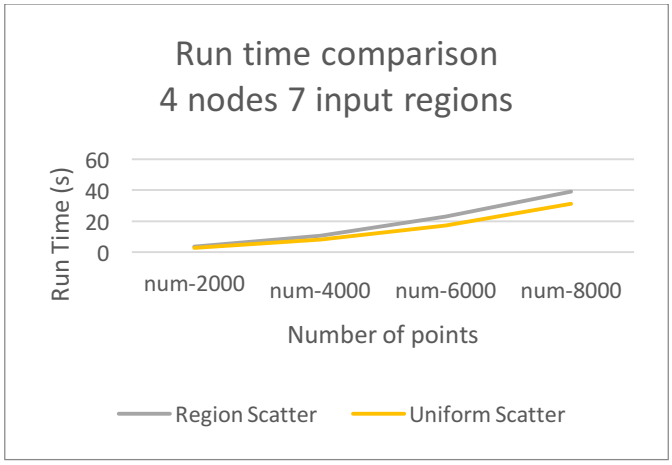

Figure 8. 4 nodes 7 input regions

From the results, we can clearly see that besides 4 nodes 4 inputs scenario, uniform scatter always performs better than region scatter, which means uniform scatter's even distribution overweighs its overheads. When there are 4 input regions and 4 nodes, where the tasks can be distributed evenly in region scatter, it still does not show clear advantage. Also, in practice, we cannot guarantee that the input region number is a multiple of the node number. Therefore, uniform scatter can achieve better performance in most cases.

Conclusion

In all, we decided to add cardioid and bulb check to quickly identify the qualified points. We then implemented OpenMP in each node according to project 1a's conclusion. Lastly, we decided to choose uniform scatter OpenMPI strategy, which divides the workload among nodes more evenly and possesses a higher performance.

Reference

[1] "Mandelbrot Set," Wikipedia The Free Encyclopedia, [Online]. Available: https://en.wikipedia.org/wiki/Mandelbrot_set. [Accessed 15 9 2018].

Group Member

Lu Chen 883241lchen12@student.unimelb.edu.au
Yiru Pan 889832 yirup@student.unimelb.edu.au