COMP90025
Parallel and Multicore Computing
Project1B Report

Chuan Yang, chuany1
Mengchen Wang, mengchenw1

Introduction

In this project, we utilize parallel computing to accelerate the computing of Mandelbrot Set, the set of values of *c* in the complex plane for which the orbit of 0 under iteration of the quadratic map remains bounded. OpenMPI as well as OpenMP are used in experiments, the task is run on multiple nodes and multiple threads on each node.

Parallelization and Optimization

1. Optimization in Mandelbrot Algorithm

The provided sequential algorithm for calculating points in mandelbrot set is a working but not very efficient algorithm. The reasoning behind that algorithm is a point is considered in a mandelbrot set if after a number of "inset" functions the produced point is still in the set. After observation, we found that if a point is converging, the value used to determine inset, namely "z_real*z_real + z_img*z_img" must also be converging. Also from observation, given a maximeter of 10000, a majority of points which are inset converges before the 1000th iteration. So we have made an assumption that:

If a point is inset, then the z_real*z_real + z_img*z_img value, which we call determinant will converge at 10% of total iteration.

This assumption allows an early escape for the inset points, thus greatly improves the algorithms runtime. For implementation, we set a small epsilon equal to 1E-10, that determinant must be within this region of change for the point to get an early escape. For a dense region, this would imply a potential 900% improvement on performance. By including this approach, we are introducing false positive to the result, namely points that iterates out of the determinant > 4 bound after the 10% of iterations, but points that are inset cannot be eliminate early. Several test on the algorithm are conducted to make sure that result collected using this method doesn't deviate from the original algorithm.

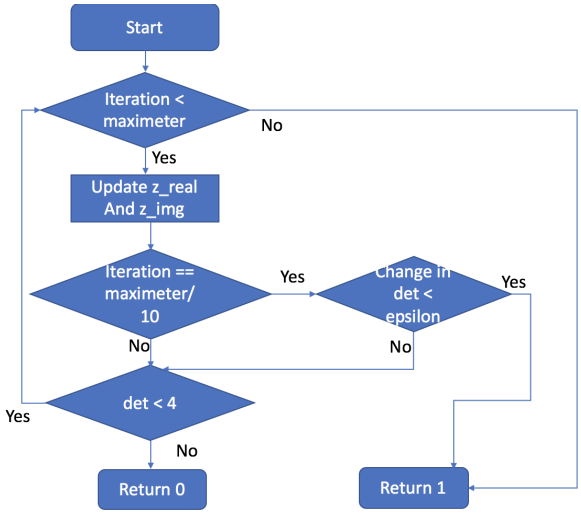

Figure 1. Flowchart for Algorithm Optimization

2. Parallelization

The first step is to utilize openMPI to run multiple instances on multiple nodes to accelerate the computing. Due to the reason that a specified area and number of points are given to compute the Mandelbrot Set, we can split the area into multiple sub-and assign those sub-areas to children instances on others nodes. To make to process simple, we

allow all the processes on all nodes read the parameters from input, and split the area by dividing the area with the number of children process. That is, each child process only compute their assigned area by dividing the real number axis but with full imagine axis. We also modify the algorithm so that the steps on different axises can differ to compensate for the parallelization we made.
Pseudocode for openMPI

```
Span = (real_upper - real_lower)/ processNumber
For id: 1 to maxProcess
   new_real_lower =  real_lower + span(id-1)
   New_real_upper = real_lower + span*id
   Img_range = old_img_range
   mandelbrot( range)
```

Then, in the algorithm of computing mandelbrot set of an area in each process, we can use OpenMP to compute the for loop in parallel. That means we compute all the points in the area in parallel. But for computation of a specific point within a concrete process, we still use serial computing due to the dependency in the inset algorithm. To solve the concurrent write issue when several threads writes their output to the same memory location, we use atomic operator in the fold operation. And because the point density each thread gets can differ drastically, dynamic scheduling is selected to improve performance.

Pseudocode for openMP part

```
Func mandelbrot :
  #parallel for schedule(dynamic)
          for real: real_lower to real_upper
                  for img: img_lower to img_upper
                  #omp atomic
                  func insert();
```

Result and Analysis

Test Number	Test Regime	Result		Time (s)		
		Orignial	Improved	Original	Improved	Speedup
1	-1.0 1.0 -1.0 1.0 100 1000	3485	3486	0.028267	0.030897	0.91487847
2	-1 1.0 0.0 1.0 100 10000	3508	3508	0.258317	0.150964	1.71111656
3	-1 1.0 0.0 1.0 100 1000	3515	3515	0.038266	0.018907	2.02390649
4	-1 1.0 0.0 1.0 100 100	3594	3594	0.004601	0.005151	0.89322462
5	-2.0 1.0 -1.0 1.0 100 10000	2538	2538	0.18827	0.073139	2.57413965
6	-2.0 1.0 -1.0 1.0 500 10000	62879	62879	4.561241	1.726859	2.64135115
7	-2.0 1.0 -1.0 1.0 1000 10000	251327	251327	18.423977	6.832105	2.6966765
8	-0.5 0.5 -0.5 0.5 1000 10000	848933	848933	61.620579	8.43484	7.30548285

Table 1. Result from Original Inset Function and Improved Inset Function

We conduct a series of test on the improved inset function to see whether it is safe to use. As it is shown in the table 1, the improved algorithm passes the correctness test for all test cases. We also notice that for a small number of maximeter, the improved algorithm even slows down the runtime. This is because after failing the escape route for a small number of iteration, the algorithm constantly checking if the 10% iteration condition is met, introducing an additional workload to the program. It does perform well otherwise. And from test 7 and 8, we can see as the region getting denser, the speedup is improved by a lot, converging to the 9-fold theoretical limit.

Test Regime	Time(s)		Speedup
	openMP	Sequential	
-0.5 0.5 -0.5 0.5 100 10000	0.02	0.097	4.85
-0.5 0.5 -0.5 0.5 500 10000	0.467	2.44	5.2248394
-0.5 0.5 -0.5 0.5 1000 10000	1.871	9.75	5.21111705
-0.5 0.5 -0.5 0.5 2000 10000	7.65	38.74	5.06405229

Table 2. Runtime Improvement from Using OpenMP

Test Regime	Time(s)		Speedup
	OpenMP+MPI	OpenMP	
-0.5 0.5 -0.5 0.5 100 10000	0.01	0.02	2
-0.5 0.5 -0.5 0.5 500 10000	0.18	0.467	2.59444444
-0.5 0.5 -0.5 0.5 1000 10000	0.95	1.871	1.96947368
-0.5 0.5 -0.5 0.5 2000 10000	4.16	7.65	1.83894231

Table 3. Runtime Improvement from Using MPI using 8 nodes

From the above 2 tables we can see that with openMP, the program can be accelerated by 5 times, and the acceleration from OpenMPI is about 2 times.

Discussion

For the improvement of Mandelbrot algorithm, we tried some other methods to improve the performance, for example, using a fixed-length vector to store the computed point when the current iteration time is greater than some threshold, eg, 1/10 of the max iteration number. Then we check the new-computed points, if they are already in the vector, the point will not diverge.
Another method we tried is to check whether the current point is the same as the next point periodically (because if we check it every time, the performance decreases). But none of them are as good as the method mentioned before.
In the parallel part, using openMP achieves the acceleration we expected. However, the utilization of openMPI indeed accelerate the program, but not as good as we expected, this may result from the way we adopt to divide workload among slave nodes.

Conclusion

In conclusion, we accelerate the program from mainly two aspects, the improvement of algorithm and parallelization. By using our scheme with 8 nodes, we can expect a speed up factor of 30, which is efficient enough for this problem.