Parallel Mandelbrot

Luhan Cheng
Faculty of IT
Monash University
Melbourne, Australia
lche0021@student.monash.edu

***Abstract*—Mandelbrot set defines a set of complex number which are bounded by constant radius under quadratic recurrence iteration $Z_{n+1} = Z_n^2 + c$ where Z initialized at 0. The set is visualized though coloring nonmember according to how quickly it diverges. This report illustrates how pixelwise partition scheme help improving the parallelization of such computation task. The experiments are implemented with OpenMPI in C and tested with multiple configurations. The result shows near linear scalability with purposed pixelwise partition scheme.**

Keywords-component; mandelbrot set, recurrence relation, parallel computing, message passing interface, visualization

I. INTRODUCTION

The visualization of Mandelbrot set has been used for studying the property fractal geometry, it can also be used to generate artistic image for appreciation.

Mandelbrot set is defined as a set of complex number c which remains bounded by constant under the following recursive relation[1].

$$Z_{n+1} = Z_n^2 + c \quad (1)$$
$$Z_0 = 0$$
$$c \in D^2, where\ D^2\ is\ square\ complex\ plain$$

In practice, to test if a number is inside Mandelbrot set, the recursive function is computed under user-defined bound B where B greater or equal to 2. And it has been proved by Benoit Mandelbrot that the value of Z will eventually goes to infinity if it ever exceeded 2[1].

In order to obtain the visualization of number points on complex plain. The iteration taken for non-Mandelbrot set number to escape is recorded and being normalized into RGB color component in 0-3 scale by

$$CC = 3 * \frac{\log(Iteration)}{\log(MaxIteration - 1)} \quad (2)$$

Each number in problem domain corresponds to one pixel. Real/Imaginary part of c corresponding to x/y axis in output image.

II. THEORETICAL ANALYSIS

A. Proof of parallelism

Bernstein's condition is used for testing if operators u and v can be parallelized [2]. The condition is defined as following.

$$M(u) \cap M(v) = \varnothing$$
$$M(u) \cap R(v) = \varnothing$$
$$R(u) \cap M(v) = \varnothing$$
$$M: p \rightarrow set\ of\ memory\ address\ p\ modify$$
$$R: p \rightarrow set\ of\ memory\ address\ p\ read$$

The input of equation (1) c is determined by problem domain D and resolution of pixels. Output iteration number is feed to equation (2). The final output is written to separate memory address before being saved to disk. Therefore, the I/O of algorithm is independent across different data points, which indicates rules of Bernstein's condition is satisfied.

B. Theoretical speedup analysis

The speedup of parallelization is estimated by Amdahl's law[3].

$$S(p) = \frac{1}{r_s + \frac{r_p}{p}}$$

where r_s is serial ratio, r_p is parallel ratio and p is number of processors. They are approximated by measuring the runtime of different portion of the code.

$$r_p = \frac{T_p}{T_{total}}, r_s = 1 - r_p$$

Total runtime T_{total} is measured by linux time function. T_p is measured by build-in C clock function.The theoretical speedup factor is shown in figure 1 and the following table.

p	2	4	6	8	10
$S_{theory}(p)$	1.942	3.675	5.230	6.633	7.906

III. PARTITION SCHEME AND ALGORITHM

For the purpose of comparison, this section will start with the discussion of some other potential parallel partition scheme. Here are two common partition schemes.

- row-based partition
- column-based partition

The row-based/column-based partition schemes distribute the data points along column/row axis. And each processor gets a number of rows/columns. The advantage of applying such method comes from the ease of managing indices of data points. It can potentially utilize parallel I/O feature provided by MPI implementation, but it may not work well for non-square domain, which could potential create imbalanced workload across processors.

The purposed algorithm is described as following pseudo-code.

Algorithm 1: Parallel Mandelbrot

Input: Max Iteration $MaxIter$, Square Complex Domain D, processor numer P, $iXmax$, $iYmax$
Output: RGB image with shape $(D, D, 3)$

```
1
2  MandelbrotTest(c)
3      return computeColor(min(MaxIter, iteration(Z_{n+1} = Z_n + c)))
4  all process do
5      localWorkload ← D^2 / P
6      masterWorkload ← D^2 mod P
7      pixelWorkload ← (iXmax × iYmax) / (size − 1)
8      pixelRemainder ← (iXmax × iYmax) mod (size − 1)
9  master process do
10     for p in P do
11         spawn MPI_Irecv(p) and store result in Image
12     end
13     for pixel in pixelRemainder do
14         compute mandelbrotTest(getXYCoordinate(pixel)) end
15     MPI_Waitall(P)
16     outputfile ← Image
17 compute process do
18     for every pixel from (rank − 1) × 3 to end with step size
         (size − 1) × 3 do
19         localimage ← mandelbrotTest(getXYCoordinate(pixel))
20     end
21     MPI_Send(localimage) to root
22
```

The algorithm starts with variable initialization. Local workload is computed by dividing size of complex domain by number of processors. And each processor then locates specific indices of target data points by inferring from MPI rank/size number which is globally accessible for all processors. If total amount of data points is not perfectly divisible by the number of compute processors, then remaining data points are handled by master process. The computed color components are sent back to master processor by blocking send, the corresponding receive on master node is non-blocking for better performance. Master processor will not write received data into output file until all results are received.

MaxIteration need to be reached for those data points in the set, which indicate more computation, and those data points are cluster around center of complex plain[4]. If datapoints assigned to each processor is contiguous, significant amount of performance decay can be observed. For example, with p=4, rank 1, 3 will only take 2 seconds to complete whereas rank 2 will take around 50 seconds as shown in figure 3.

To address this issue, D^2 is divided into $D^2 / size$ contiguous blocks and each block contains one datapoint from each compute process ordered by its rank. This ordering avoids one type of data point being cluster on a small subset of processors and helps achieving load balance. A customized MPI vector datatype is introduced to help managing the ordering during communication.

The following table provides an illustration with the configuration $|D| = 5\ unit, p = 4$ with master process ranks 0. Each cell represents one pixel being computed by process number identified.

1	2	3	1	2
3	1	2	3	1
2	3	1	2	3
1	2	3	1	2
3	1	2	3	0

From the table we can see that the workload on master process is trivial compare to the workload on compute process. The number of pixels is computed at line 8 within algorithm 1. It indicates that the number of data points on master process is bounded by processor number, which is magnitude smaller compare to problem size. The reason of assigning less computation on master is when we scale up the application of parallel algorithm, the communication overhead between processors will be the major boundary of performance improvement [5]. One possible disadvantage of adopting such task decomposition would be performance degradation when there is small amount of computation resources. It is caused by not utilizing the computational resources on master process while it waits for other processes.

IV. EXPERIMENT

The experiment is conducted on MASSIVE[6], partition m3j with high density CPUs and high memory capacity. Serial code is compiled with GCC version 5.4.0, parallel code is complied with OpenMPI version 1.10.7. Both of them are compiled with -O3 optimization. Each MPI process is allocated one gigabytes memory. Parallel version of code is run on one node with 2,4,6,8,10 process. Results of both parallel and serial experiments are averaged over 5 trials. Outputs are shown in figure 2 and figure 4.

It can be concluded that the performance for size range from 4 to 10, the actual speedup approximately converges to theoretical speedup. For size 2, there is no performance gain with one extra processor compare to serial implementation, since the master process only manages I/O, MPI communication and trivial amount computation. For size 10, we can see that the actual time taken is close to optimal point, in this configuration the program achieves 0.94 percent of theoretical speedup. A near linear speedup can be observed in

figure 1. The outputs indicate successful implementation of parallelization.

V. Conclusion

The report demonstrated how to apply domain decomposition on the complex number plain to speed up the computation of Mandelbrot number. Due to high parallelization potentiality, it may be used in benchmark HPC facility or generating fractal image in real time[7]. To further improve parallel Mandelbrot algorithm, one may explore mathematical properties of Mandelbrot set in order to place a heuristic function for more precise estimation of computation cost. Furthermore, the algorithm implements serial I/O, it is possible to extent it utilizing parallel MPI I/O for further performance gain.

VI. Bibliography

[1] Benoit B. Mandelbrot, *Fractals and chaos: the Mandelbrot set and beyond : selecta volume C*. New York ; London: Springer, 2004.

[2] P. Feautrier, “Bernstein’s Conditions,” 2011, pp. 130–134.

[3] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS ’67 (Spring)*, Atlantic City, New Jersey, 1967, p. 483.

[4] K. Keller, “The Abstract Mandelbrot set,” in *Invariant Factors, Julia Equivalences and the (Abstract) Mandelbrot Set*, K. Keller, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 73–139.

[5] A. Jokanovic, G. Rodriguez, J. C. Sancho, and J. Labarta, “Impact of Inter-application Contention in Current and Future HPC Systems,” 2010, pp. 15–24.

[6] “Welcome to the M3 user guide!” [Online]. Available: https://docs.massive.org.au/. [Accessed: 02-Sep-2019].

[7] “Mandelbrot Viewer.” [Online]. Available: http://math.hws.edu/eck/js/mandelbrot/MB.html. [Accessed: 05-Sep-2019].

I. Appendix

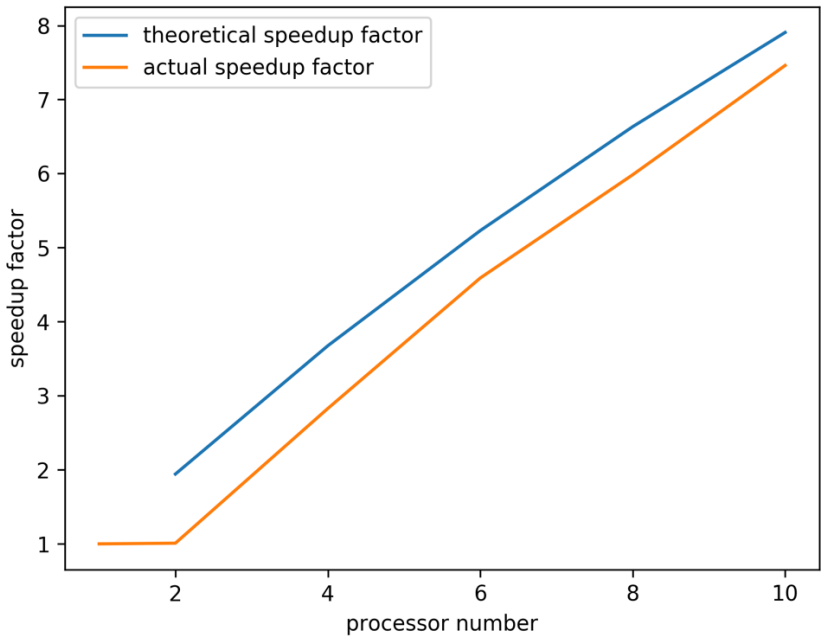

Figure 1. Theoretical vs actual speedup factor

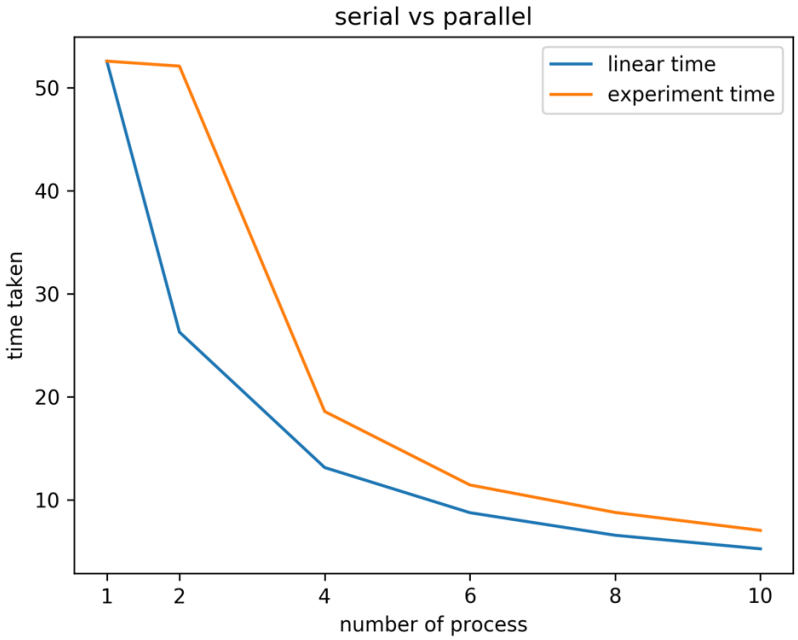

Figure 2. Theoretical vs actual time

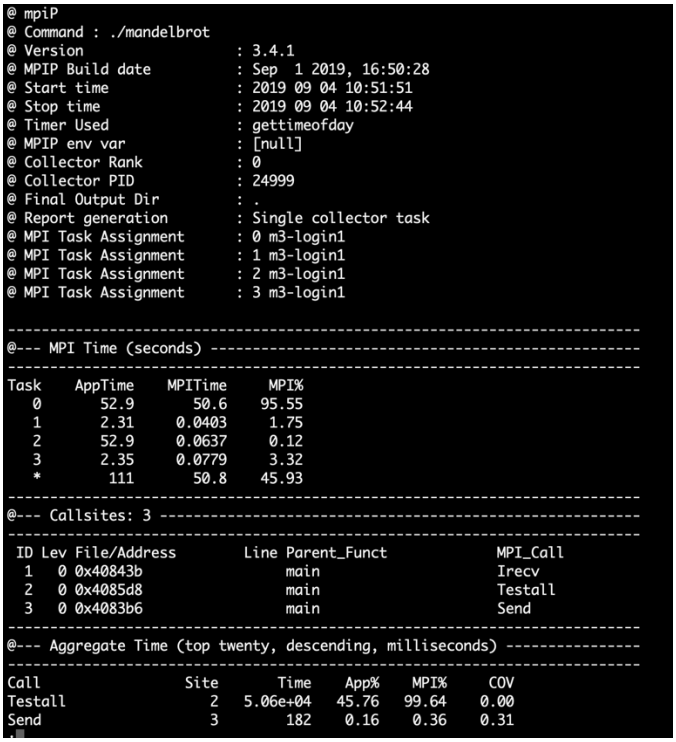

Figure 3. output from MPIP profiling

Computer specificatior	cpu: Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz					
	number of logical core: 36					
	memory size: 1GB per task					
value of iXmax	8000					
value of iYmax	8000					
value of IterationMax	2000					
	serial program	parallel program				
		MPI				
		2	4	6	8	10
run #1	53.037	52.13	18.63	11.36	8.54	7.07
run #2	52.062	52.13	18.59	11.42	8.88	7
run #3	52.647	52.02	18.63	11.39	8.78	7.06
run #4	52.648	52.1	18.46	11.52	9.18	7.1
run #5	52.42	52.03	18.5	11.55	8.59	6.99
average time	52.5628	52.08	18.58	11.45	8.78	7.05

Figure 4. output from MPIP profiling