香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Mandelbrot Set Computation

Assignment II

CSC4005: Parallel Programming

Name: *Derong Jin*
Student ID: *120090562*

Date: October 31, 2022

1 Introduction

Fractuals are a topic of great interests to mathmaticians, revealing to us that the world is not as smooth as it ideally should be. Common sense says that for most mathematical models (e.g., sets, functions, etc.), otherwise complex structures or contours may become simple and smooth if we zoom in infinitely on a local, e.g., Taylor polynomials are used to approximate functions in engineering. However the existence of fratuals tells us complex geometric structure can be developed even from simple algebraic laws. One of the most famous examples is the Mandelbrot Set.

The Mandelbrot Set describe $c \in \mathbb{C}$ for which the sequence $z_{n+1} = z_n^2 + c$ started from $z_0 = 0$ does not diverge to infinity as $n \to \infty$. The algebraic structure is quite simple, however leads to a coarse but recursively self-similar structure. To visualize the Mandelbrot Set on the complex plane, we can simply follows its definition to iteratively find z_n and see if it goes out of a certain bound. The bound is chosen to be the circle with its origin at $(0, 0)$ and radis of 2. It can be proved that any z_n out of this range will definitely diverge to infinity. The only thing that is related to the the accuracy of our decision is the stop iteration we choose, we use K_{max} to denote this constant in the following.

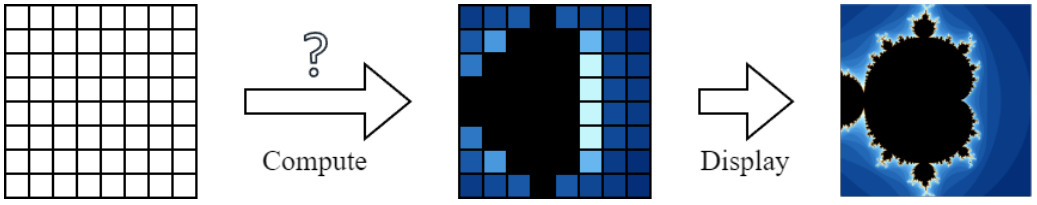


Figure 1: The theoretical basis of visualizing Mandelbrot Set is well established. With the help of morden computer, it is now possible to paint the structure of Mandelbrot Set in great details, however, the point is, how can we speed up this visualizing procedure using parallel computing?

As everything we need to calculate the set is offered, an intuitive way to visualize such a set is to first assign an appropriate sampling point to each pixel in the view area, and then use the calculation of the sampling point to determine the pixel's color. Suppose the procedure is conducting pixel by pixel, the evaluation is conducted for $O(N) = O(H \times W)$ times where N is the number of pixels, H and W are the height and the width of the sampling matrix respectively. In this project, I have tried to use `MPI` and `Pthread` to parallize such computaion. In `MPI` implementaion, the calculation is distributed to several processes on mutiple processors, while in `Pthread` version, the job is done by multiple threads cooperate to finished the evaluation. Additionally, when using multi-threads to parallize such procedure, I have implemented both static task distribution as well as dynamic job assigning in this project. The performance of these implementaions are compared in several selected dimensions.

2 Method

2.1 Sequential Implementaion

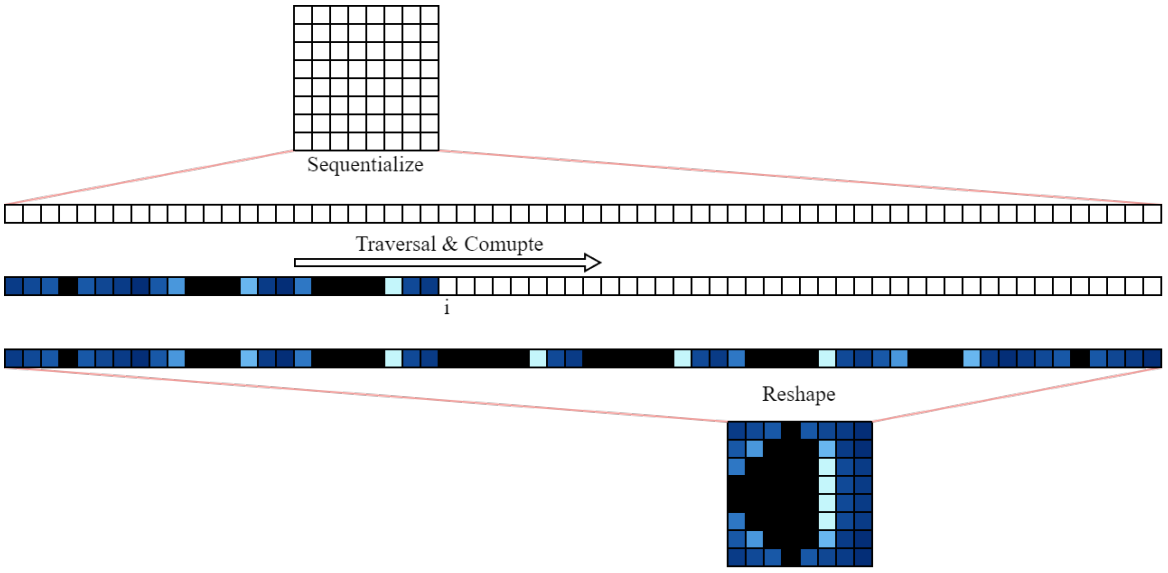


Figure 2: he work flow of evaluations on sample points by sequential method

The process of sequentially computing the values of each sample point is as follows:

1. Sequentialize the two-dimensional $N = H \times W$ sample points into a sequence of sample points of length N
2. Iterate through each sample point and calculate the number of iterations at the point
3. Remap the color information of each sample point to a two-dimensional buffer (is not counted in total runtime)

2.2 Parallization Using MPI

By observation, such a process can be computed by simply partitioning the sequence of sample points into parts, and then compute each part parallelly. For computations on different samples are independent, the parallelization is an example of embarrassingly parallel problems. Therefore, no matter how I partition the sequence, the correctness of the total algorithm will not be affected.

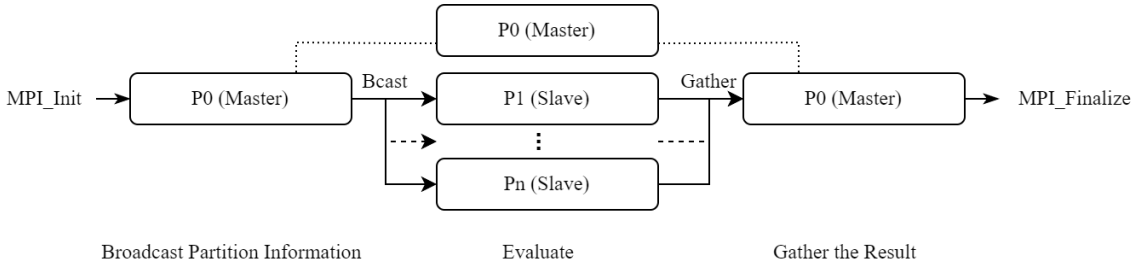


Figure 3: Flowchart of The Idea of MPI Parallelizing

The process of parallel computing the values of each sample point using MPI is as follows:

1. Sequentialize the two-dimensional $N = H \times W$ sample points into a sequence of sample points of length N
2. Partition the sequence into N_p consecutive parts of approximately equal lenghts.
3. On each processor, iterate through each assigned sample point and calculate the number of iterations at the point
4. Gather the computed results to the master processor.
5. Remap the color information of each sample point to a two-dimensional buffer (is not counted in total runtime)

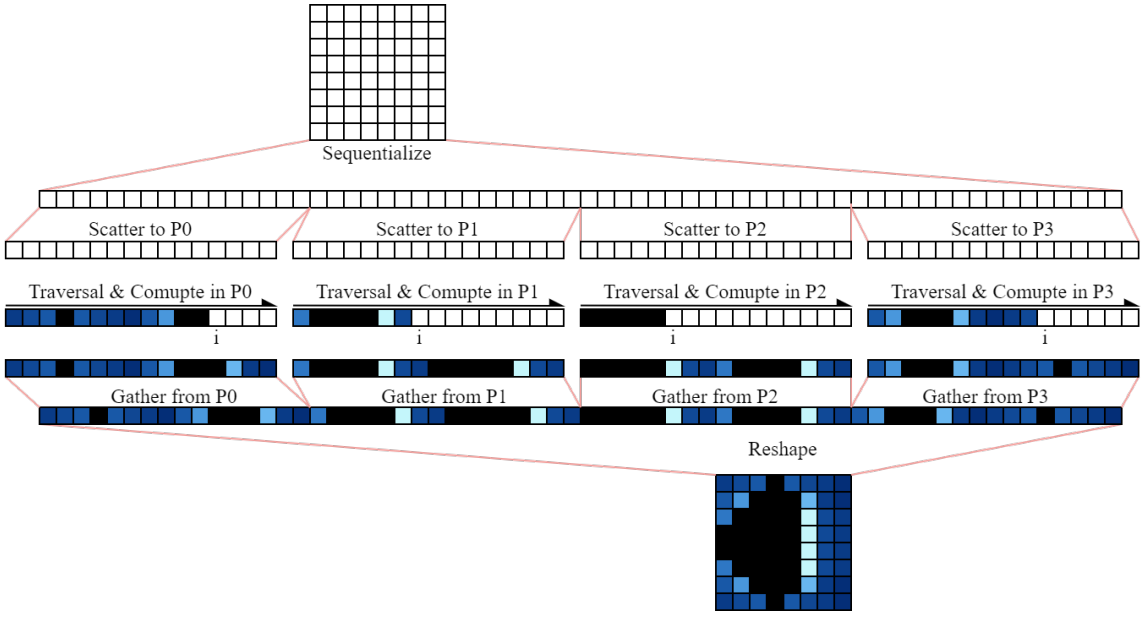


Figure 4: The work flow of evaluation by parallel method implemented by MPI

2.3 Parallization Using Pthread (Static Version)

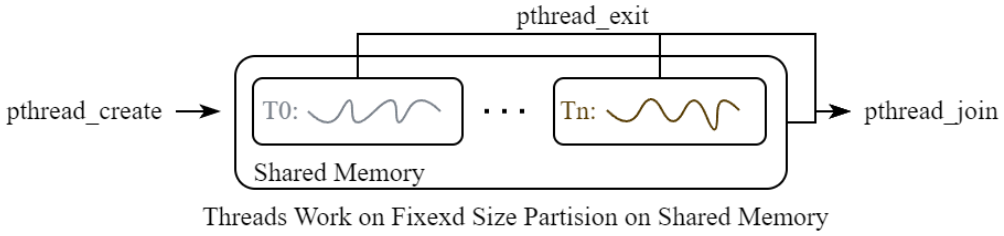


Figure 5: Flowchart of The Idea of Pthread Static Parallelizing

The idea of implementing static Pthread parallelization is very similar to that of MPI, that is:

1. Sequentialize the two-dimensional N sample points into a sequence of sample points.
2. Create N_t threads, assigning equal-sized parts of the sequence to each thread.
3. On each thread, iterate through assigned sample points and calculate the number of iterations at the points.
4. After calculation, the shared memory ready and there is no needs for gathering.
5. Remap the color information of each sample point to a two-dimensional buffer (is not counted in total runtime)

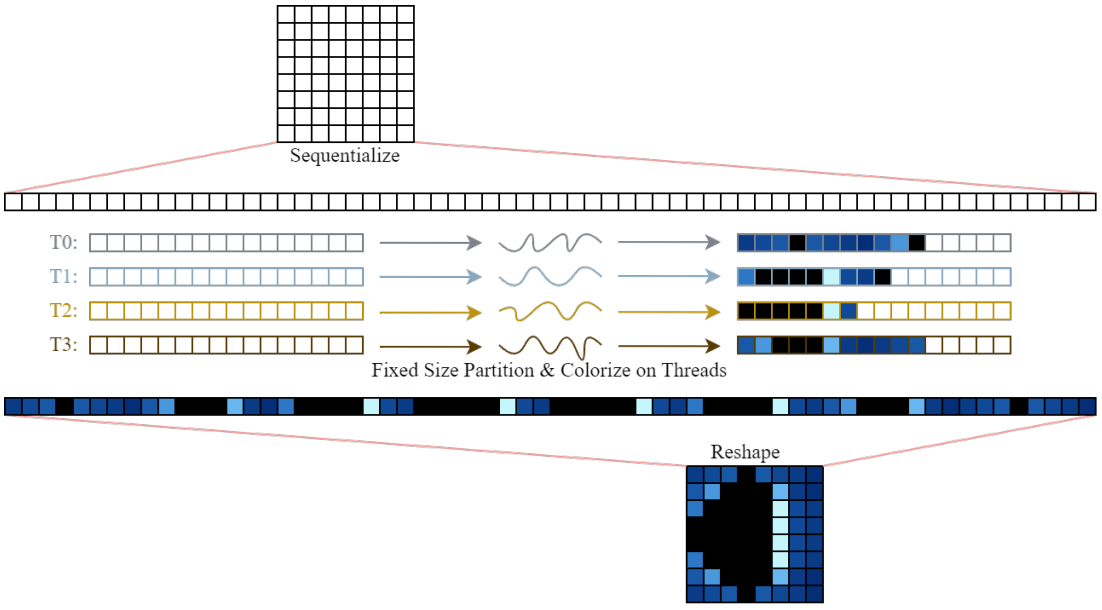


Figure 6: The work flow of evaluation by static-parallel method implemented by pthread

2.4 Dynamic Task Allocation Using Pthread

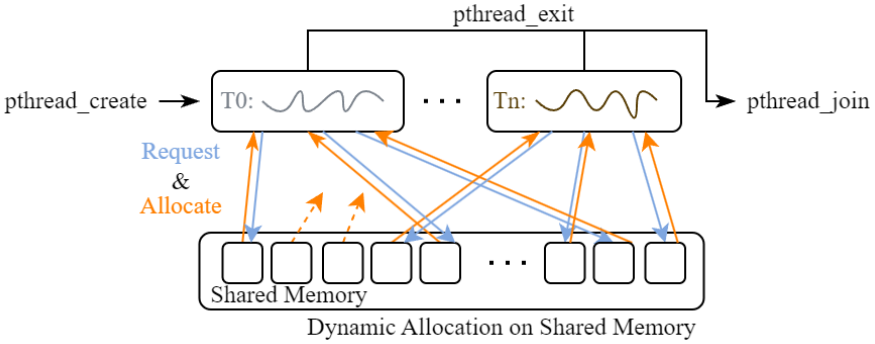


Figure 7: Flowchart of The Idea of Pthread Dynamic Parallelizing

The only difference between static partitioning and dynamic allocation is that, when threads are created, they are not assigning to their jobs immediately. Instead, they runs a loop to request for jobs. When one thread is initialized or finishing its job it will request for its new job. This job-allocating procedure is at risk in concurrent requesting, so is protected to be thread-safe by a mutual-excluded lock `pthread_mutex_t busy`. When a thread is asking for a new job, the mutex is locked to ensure there is only one thread allocating its job. And after allocation, the mutex is unlocked to admit the next requester. Dynamic job-Allocation is theoretically more efficient than static partitioning, for it better ensures the fairness between threads.

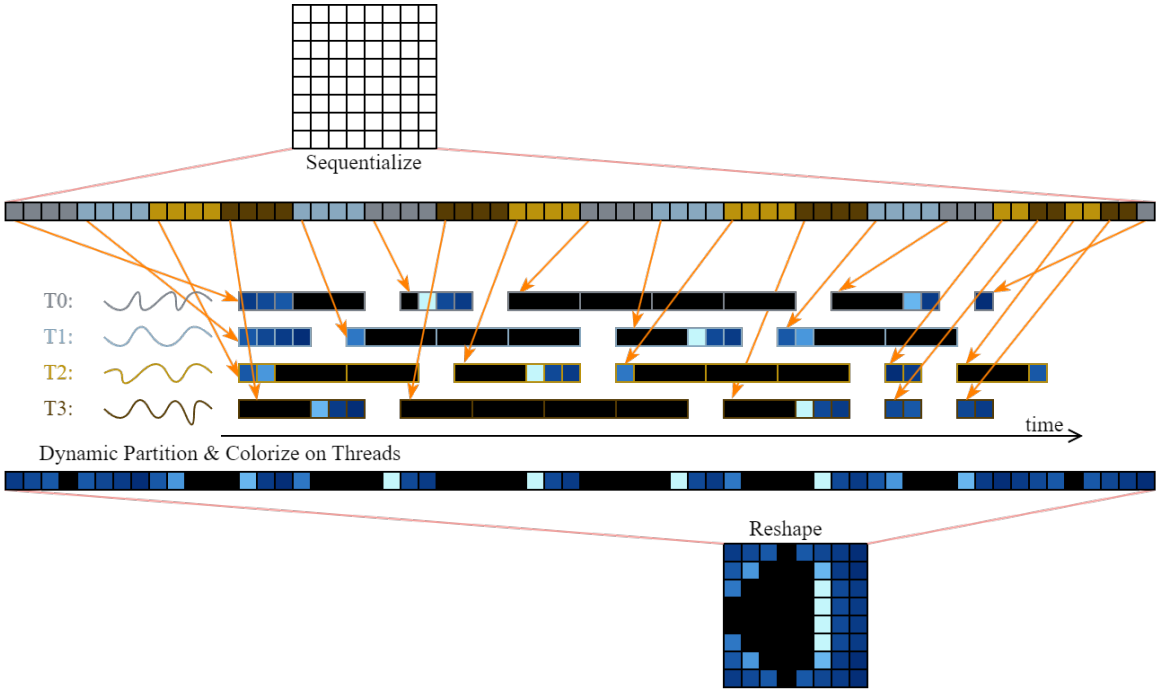


Figure 8: The work flow of evaluation by dynamic-parallel method implemented by pthread

2.5 Colorization

Suppose the maximum number of iteration is K_{max}, then, any point $z_0 \in \mathbb{C}$ on the complex plane is regarded to as a part of Mandelbrot Set if z_n does not diverge after K_{max} iterations. Additionally, for z_0 not in Mandelbrot Set, there is a rational number $k = \frac{num\_it}{K_{max}}$ denotes the point's "relationship" with Mandelbrot Set, i.e. the greater k is for a given point, the "closer" it is regarded to the Mandelbrot Set. Such a naive approach provides an excellent way to visualize the structure of Mandelbrot Set.

The colorization based on the following equations:

$$\begin{aligned} R(k) &= 0.5 + 0.5\cos(2\pi k + \pi) \\ G(k) &= 0.5 + 0.5\cos(2\pi k + 1.2\pi) \\ B(k) &= 0.5 + 0.5\cos(2\pi k + 1.4\pi) \end{aligned} \tag{1}$$

where $k = \frac{num\_it}{K_{max}} \in [0, 1]$, and R, G, B are RGB color channels respectively.
(credit: adopted from `https://iquilezles.org/articles/palettes/`)

By adopting this colorization method, it is ready to paint the following Mandelbrot Set to screen.

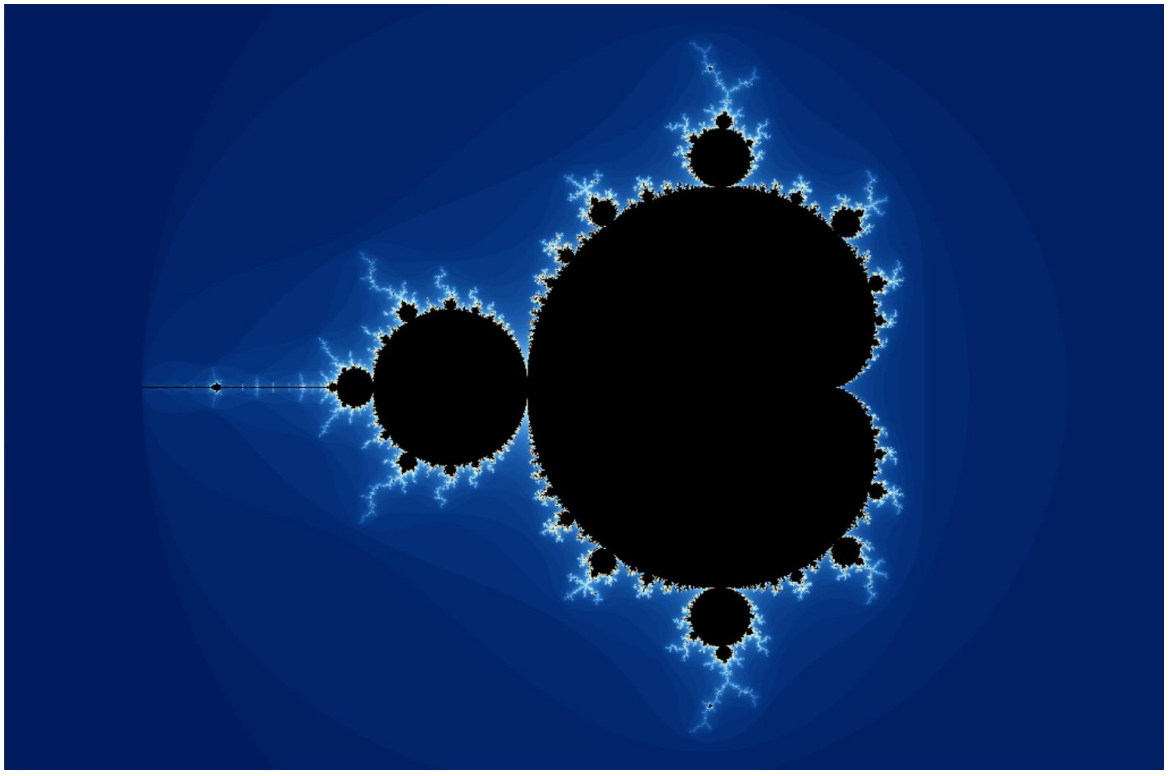
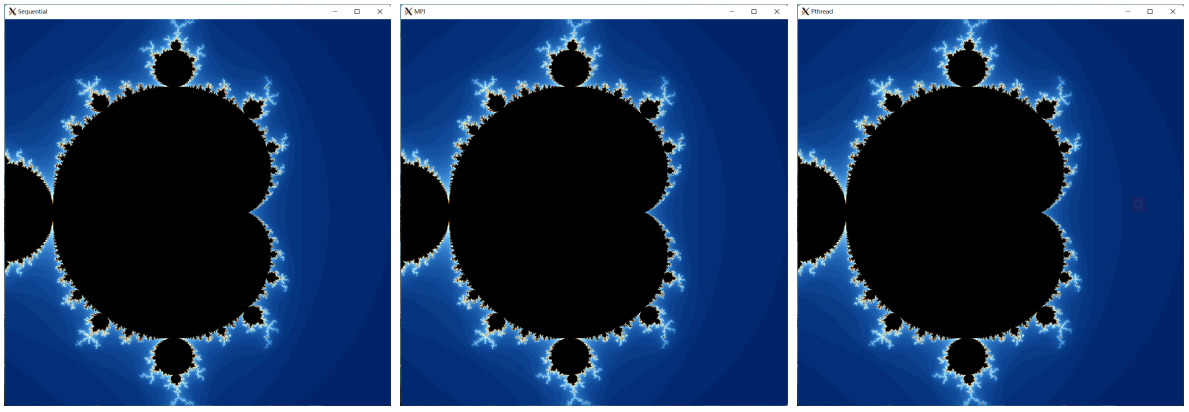


Figure 9: HD Mandelbrot Set Figure by setting: $K_{max} = 100$

3 Results & Analysis

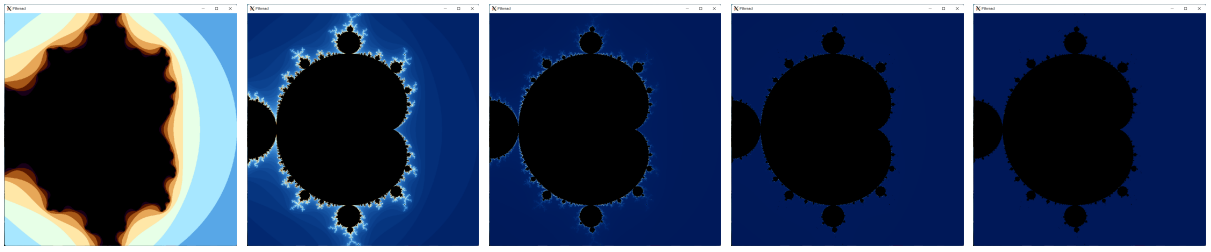
3.1 GUI Results

The following shows the colorized GUI and gray-scale GUI (needs modifications on source code) under different configurations.

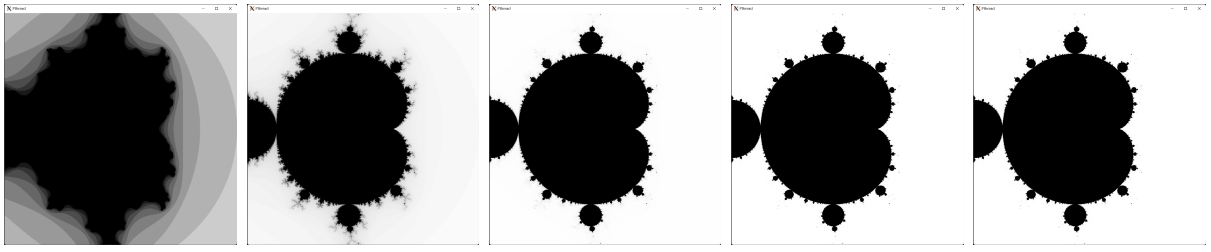


(a) Sequential (b) MPI (c) Pthread

Figure 10: GUI version of each implementaion



(a) $K_{max} = 10$ (b) $K_{max} = 100$ (c) $K_{max} = 1000$ (d) $K_{max} = 10000$ (e) $K_{max} = 100000$

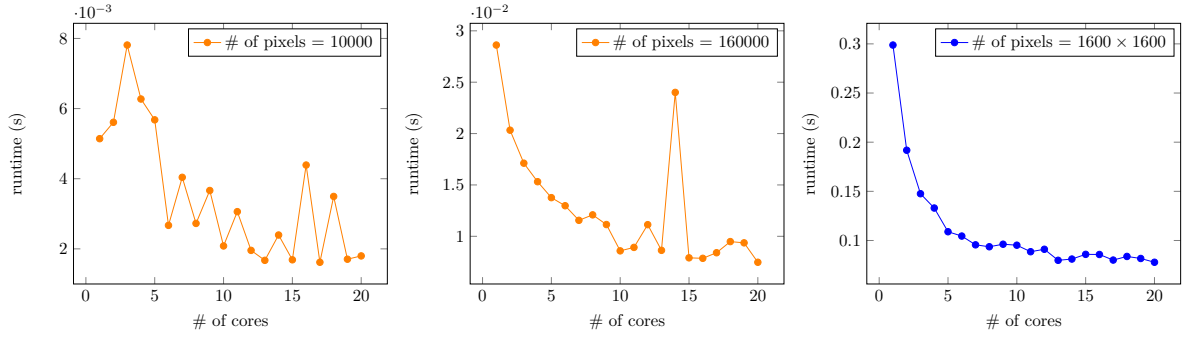


(f) $K_{max} = 10$ (g) $K_{max} = 100$ (h) $K_{max} = 1000$ (i) $K_{max} = 10000$ (j) $K_{max} = 100000$

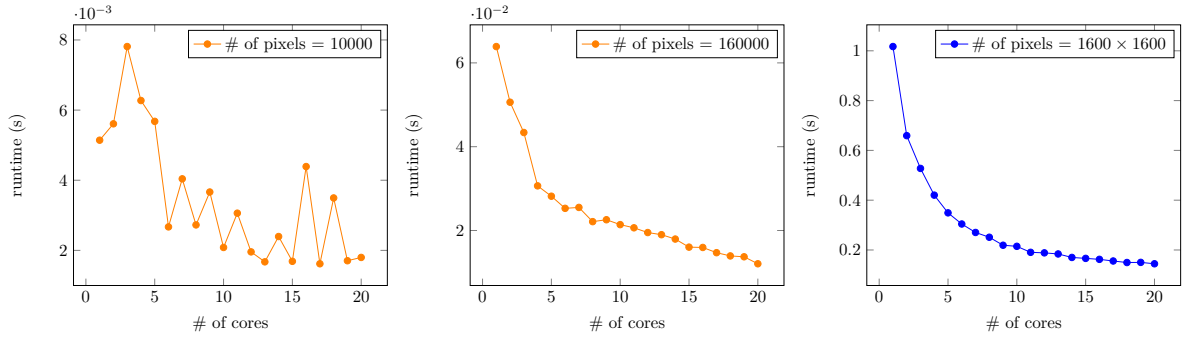
Figure 11: GUIs showing the result of $N = 1000^2$, where (a)-(e) are gui adopting colorization method, while (f)-(j) are gui using only k as gray scale factor

3.2 Performance of MPI Implementation

Performance Under Different Number of Cores



(a) $N = 100^2$, $K_{max} = 10$ (b) $N = 400^2$, $K_{max} = 10$ (c) $N = 1600^2$, $K_{max} = 10$



(d) $N = 100^2$, $K_{max} = 100$ (e) $N = 400^2$, $K_{max} = 100$ (f) $N = 1600^2$, $K_{max} = 100$

Figure 12: The six figures illustrates the relationship between MPI program's runtime and the number of cores used in six different problems scales as Table 3.2 displays

# of cores(N_p)	1	2	3	4	5	6	7	8	9	10
$N = 100^2$, $K_{max} = 10$	0.0032	0.0032	0.0016	0.0015	0.0013	0.0021	0.0012	0.0012	0.0019	0.0010
$N = 400^2$, $K_{max} = 10$	0.0286	0.0203	0.0171	0.0153	0.0138	0.0130	0.0116	0.0121	0.0111	0.0086
$N = 1600^2$, $K_{max} = 10$	0.2988	0.1918	0.1476	0.1330	0.1089	0.1045	0.0956	0.0936	0.0962	0.0952
$N = 100^2$, $K_{max} = 100$	0.0051	0.0056	0.0078	0.0063	0.0057	0.0027	0.0040	0.0027	0.0037	0.0021
$N = 400^2$, $K_{max} = 100$	0.0639	0.0506	0.0434	0.0307	0.0282	0.0253	0.0255	0.0221	0.0226	0.0214
$N = 1600^2$, $K_{max} = 100$	1.0171	0.6592	0.5276	0.4203	0.3491	0.3044	0.2704	0.2512	0.2191	0.2147
# of cores(N_p)	11	12	13	14	15	16	17	18	19	20
$N = 100^2$, $K_{max} = 10$	0.0030	0.0010	0.0013	0.0019	0.0034	0.0010	0.0011	0.0017	0.0011	0.0016
$N = 400^2$, $K_{max} = 10$	0.0089	0.0111	0.0086	0.0240	0.0079	0.0079	0.0084	0.0095	0.0094	0.0075
$N = 1600^2$, $K_{max} = 10$	0.0886	0.0911	0.0798	0.0811	0.0859	0.0858	0.0801	0.0838	0.0817	0.0779
$N = 100^2$, $K_{max} = 100$	0.0031	0.0020	0.0017	0.0024	0.0017	0.0044	0.0016	0.0035	0.0017	0.0018
$N = 400^2$, $K_{max} = 100$	0.0206	0.0195	0.0190	0.0180	0.0160	0.0160	0.0147	0.0139	0.0137	0.0121
$N = 1600^2$, $K_{max} = 100$	0.1906	0.1885	0.1840	0.1700	0.1663	0.1624	0.1558	0.1497	0.1502	0.1446

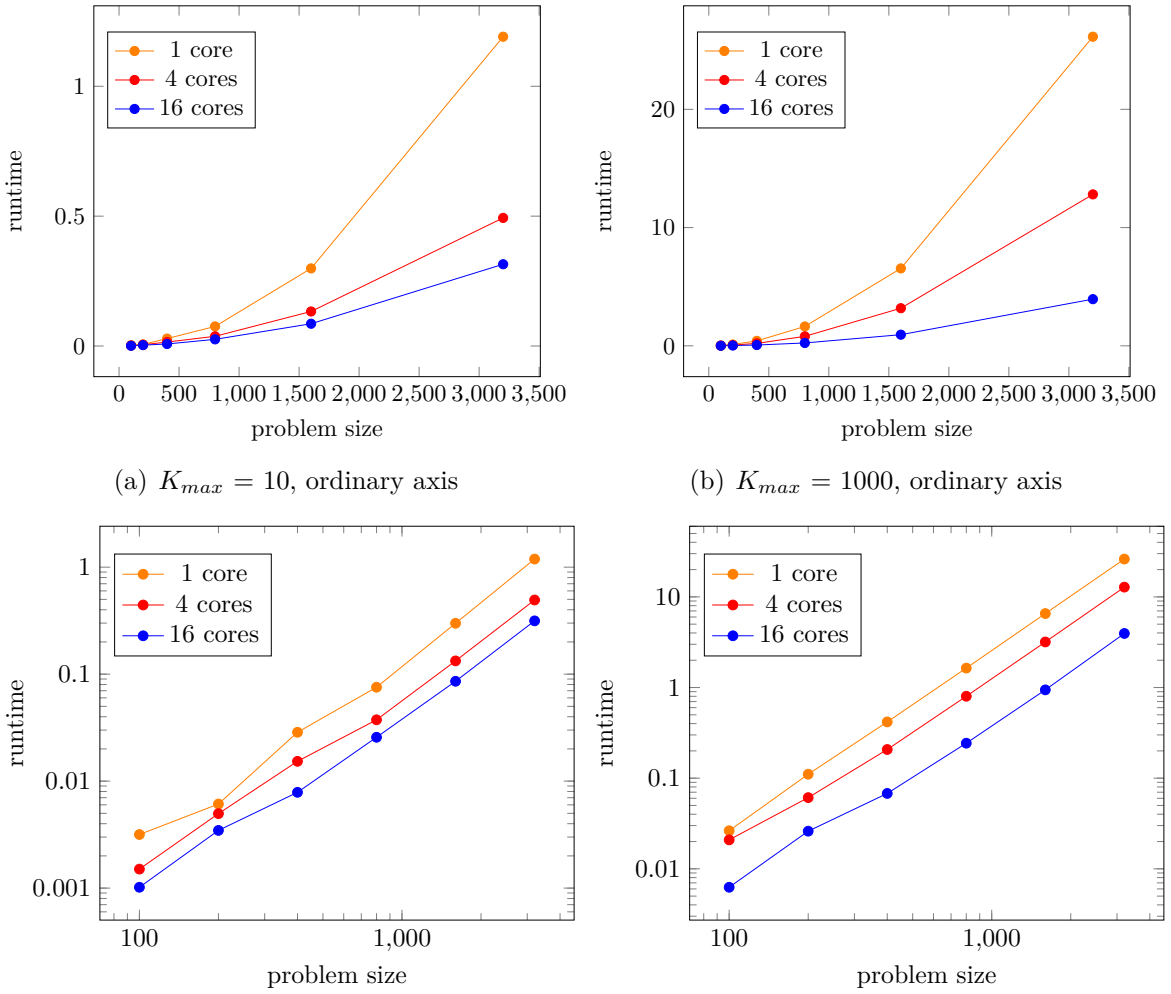
Table 1: Data on MPI program's runtime (unit: s) in 6 selected configurations

Theoretically, for the same size problem, the more the number of processors, the more efficient the MPI program will be. The experimental data of the six groups above show that when the

data scale is large, it does conform to this rule. When the data scale is small, the time saved by parallelization is consumed by transferring information because more information needs to be transferred concurrenty between processors. So the possibility of increasing the processors at small data volume does not further save the running time of the program.

Additionally, it is interesting to see that, when $N = 400^2$, $K_{max} = 10$ there is a surge in the data when the number of processors increases to 14. The reason might be related to the organization pattern of processors, or, for the problem is statically partitioned in this MPI implementation, the unfairness between processors may come to a peak in this situation.

Performance Under Different Problem Sizes



(a) $K_{max} = 10$, ordinary axis

(b) $K_{max} = 1000$, ordinary axis

(c) $K_{max} = 10$, logarithmic axis

(d) $K_{max} = 100$, logarithmic axis

Figure 13: The figures illustrates the relationship between MPI program's runtime and the problem size in different problems scales & number of cores used as Table 3.2 shows.

Problem Scale(N)	100^2	200^2	400^2	800^2	1600^2	3200^2
$N_{core} = 1$, $K_{max} = 10$	0.003168	0.006110	0.028616	0.075483	0.298804	1.190849
$N_{core} = 4$, $K_{max} = 10$	0.001505	0.004966	0.015315	0.037425	0.133023	0.493310
$N_{core} = 16$, $K_{max} = 10$	0.001018	0.003461	0.007858	0.025696	0.085771	0.314837
$N_{core} = 1$, $K_{max} = 100$	0.005142	0.019397	0.063896	0.254445	1.017074	4.088568
$N_{core} = 4$, $K_{max} = 100$	0.006273	0.015338	0.030652	0.112689	0.420280	1.670361
$N_{core} = 16$, $K_{max} = 100$	0.004390	0.004955	0.015952	0.044752	0.162397	0.597816

Table 2: MPI program's runtime (unit: s) v.s. its problem scale under different problem sizes

If we assume K_{max} as a constant, the time complexity of the sequential algorithm is $O(H \times W) = O(N)$. From Fig 13(a) and 13(b), the runtime increases with the increment of problem size, however reduces when new processor is introduced under the same problem size. Thus the problem is a of polynomial complexity.

When we display it in logarithmic coordinates, the $e^{runtime}$ will exhibit as a linear relationship with e^N. From Fig 13(c) and 13(d), it is clear that although new processor indeed speeds up the original program, the time complexity is the same (for the slope does not change after adding new cores), while the runtime is multiplied by a constant (for their is difference in baises)

3.3 Performance of Pthread Implementation

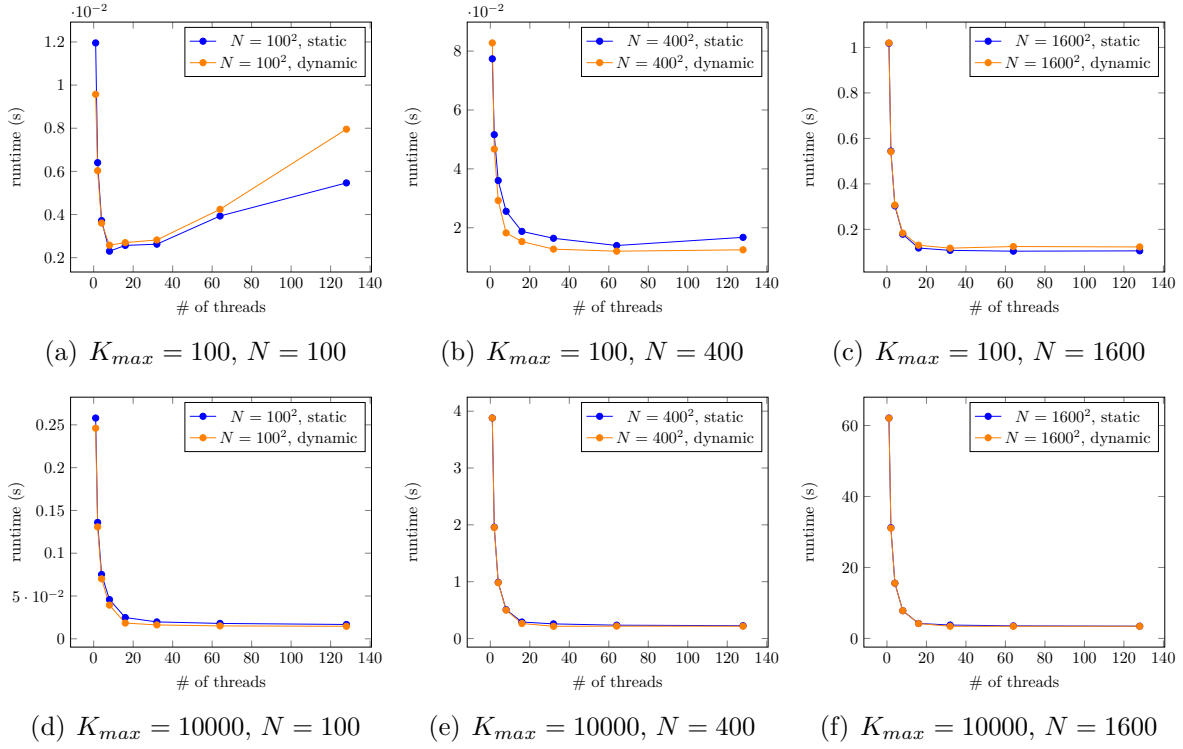
Performance Under Different Number of Threads

Number of threads (N_t)	1	2	4	8	16	32	64	128
$K_{max} = 100$, $N = 100$	0.011955	0.006405	0.003719	0.002301	0.002568	0.002625	0.003932	0.005465
$K_{max} = 100$, $N = 400$	0.077371	0.051611	0.036033	0.025589	0.018759	0.016424	0.013968	0.016741
$K_{max} = 100$, $N = 1600$	1.019894	0.541783	0.306898	0.183694	0.130259	0.117343	0.124491	0.122659
$K_{max} = 10000$, $N = 100$	0.257984	0.136020	0.075197	0.045856	0.024844	0.019748	0.017963	0.016740
$K_{max} = 10000$, $N = 400$	3.879824	1.959290	0.988481	0.507461	0.290961	0.257532	0.234902	0.225730
$K_{max} = 10000$, $N = 1600$	62.113852	31.204853	15.570008	7.813102	4.219854	3.760015	3.534236	3.443848

Table 3: Runtime (unit: s) of dynamic allocation implemented with Pthread

Number of threads (N_t)	1	2	4	8	16	32	64	128
$K_{max} = 100$, $N = 100$	0.011955	0.006405	0.003719	0.002301	0.002568	0.002625	0.003932	0.005465
$K_{max} = 100$, $N = 400$	0.077371	0.051611	0.036033	0.025589	0.018759	0.016424	0.013968	0.016741
$K_{max} = 100$, $N = 1600$	1.019894	0.541783	0.306898	0.183694	0.130259	0.117343	0.124491	0.122659
$K_{max} = 10000$, $N = 100$	0.257984	0.136020	0.075197	0.045856	0.024844	0.019748	0.017963	0.016740
$K_{max} = 10000$, $N = 400$	3.879824	1.959290	0.988481	0.507461	0.290961	0.257532	0.234902	0.225730
$K_{max} = 10000$, $N = 1600$	62.113852	31.204853	15.570008	7.813102	4.219854	3.760015	3.534236	3.443848

Table 4: Runtime (unit: s) of static sequence partitioning implemented with Pthread



(a) $K_{max} = 100$, $N = 100$ (b) $K_{max} = 100$, $N = 400$ (c) $K_{max} = 100$, $N = 1600$

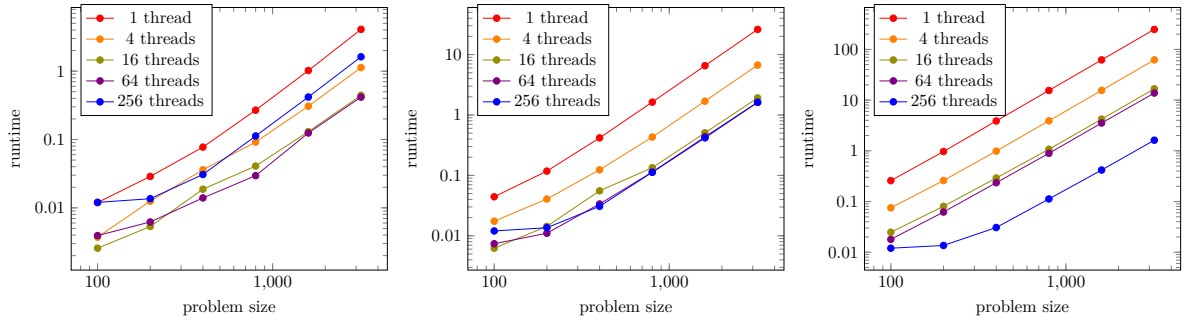
(d) $K_{max} = 10000$, $N = 100$ (e) $K_{max} = 10000$, $N = 400$ (f) $K_{max} = 10000$, $N = 1600$

Figure 14: Pthread program's runtime (unit: s) v.s. the problem size under different conditions (the data is consistent with Table 3.3)

Theoretically, for the same size problem, the more the number of threads, the more efficient the Pthread program will be. The experimental data above show that when $N_t \leq 20$, the runtime indeed reduces when the number of threads increases. However, when $N_t > 20$ the relation is not guaranteed to hold. For there is only 20 processors in the environment, more than 20 threads implies some threads are not physically parallel to each other. And the program is a CPU-bound one, so multiple threads sharing CPU does not have much speed up but suffers from lantency in context switch. Typically, when data scale is small (e.g., $K_{max} = 100$, $N = 100$) the runtime even increases with increment of number of threads.

Interestingly, the data shows there is nearly no speedup by adopting dynamic allocation in pthread program, or in some situation the dynamic approach even has a worse performance. This phenomenon will be discussed later.

Performance Under Different Problem Sizes



(a) $K_{max} = 100$ (b) $K_{max} = 1000$ (c) $K_{max} = 10000$

Figure 15: logarithmic axis, runtime (unit: s) v.s. problem size (consistent with 3.3)

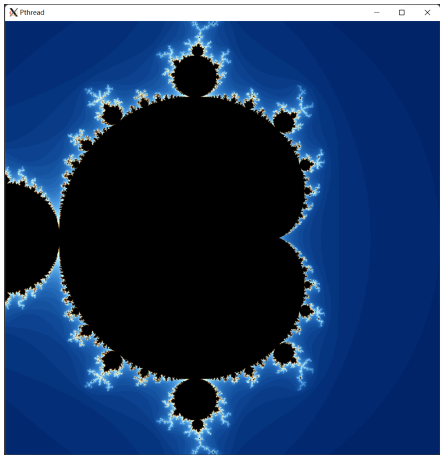
Problem Scale (N)	100^2	200^2	400^2	800^2	1600^2	3200^2
$K_{max} = 100$, $N_t = 1$	0.011955	0.028811	0.077371	0.268741	1.019894	4.081640
$K_{max} = 100$, $N_t = 4$	0.003719	0.012517	0.036033	0.091703	0.306898	1.126677
$K_{max} = 100$, $N_t = 16$	0.002568	0.005332	0.018759	0.040860	0.130259	0.442098
$K_{max} = 100$, $N_t = 64$	0.003932	0.006219	0.013968	0.029568	0.124491	0.416701
$K_{max} = 100$, $N_t = 256$	0.008286	0.011503	0.018490	0.031122	0.123895	0.434656
$K_{max} = 1000$, $N_t = 1$	0.044225	0.117671	0.417392	1.632025	6.540377	26.110877
$K_{max} = 1000$, $N_t = 4$	0.017420	0.040589	0.123549	0.430716	1.692545	6.698867
$K_{max} = 1000$, $N_t = 16$	0.006229	0.014315	0.055473	0.133995	0.506877	1.915994
$K_{max} = 1000$, $N_t = 64$	0.007384	0.011019	0.033606	0.113317	0.435029	1.619980
$K_{max} = 1000$, $N_t = 256$	0.012016	0.013608	0.030743	0.112481	0.418205	1.620951
$K_{max} = 10000$, $N_t = 1$	0.257984	0.969433	3.879824	15.516828	62.113852	248.322305
$K_{max} = 10000$, $N_t = 4$	0.075197	0.259674	0.988481	3.907011	15.570008	62.175442
$K_{max} = 10000$, $N_t = 16$	0.024844	0.080517	0.290961	1.076285	4.219854	16.734337
$K_{max} = 10000$, $N_t = 64$	0.017963	0.062129	0.234902	0.893390	3.534236	13.788240
$K_{max} = 10000$, $N_t = 256$	0.017165	0.059202	0.226375	0.870584	3.444632	13.686856

Table 5: Runtime data (unit: s) of static pthread program under different configurations

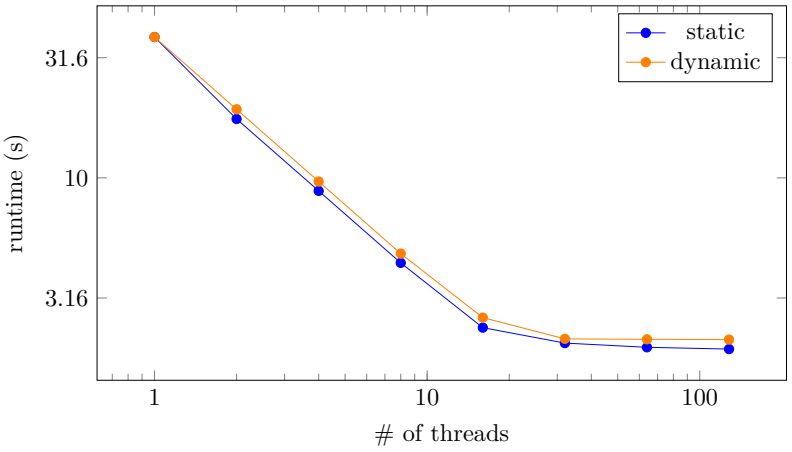
From the data above, on logarithmic coordinates, the programs' runtime is exhibited to be with the same time complexity, for there is little difference in the slope. When data scale is becoming larger, the program, whatever the configuration is, is converging to the same slope range, which highly conforms our prediction.

Additionally, it is still true when thread number exceeds the number of processors used. For thread number over 20, i.e., 64 and 256, although suffers from context switchs the time cost is converging to the same slope value.

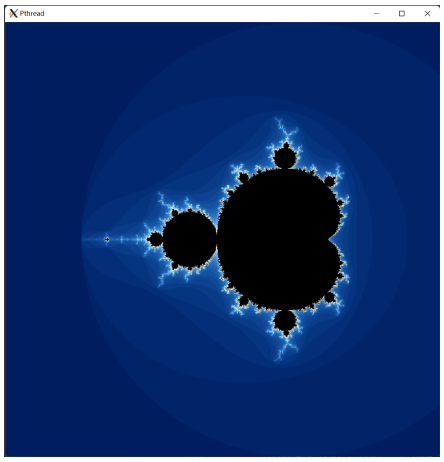
Discussion: About Dynamic Job Allocation



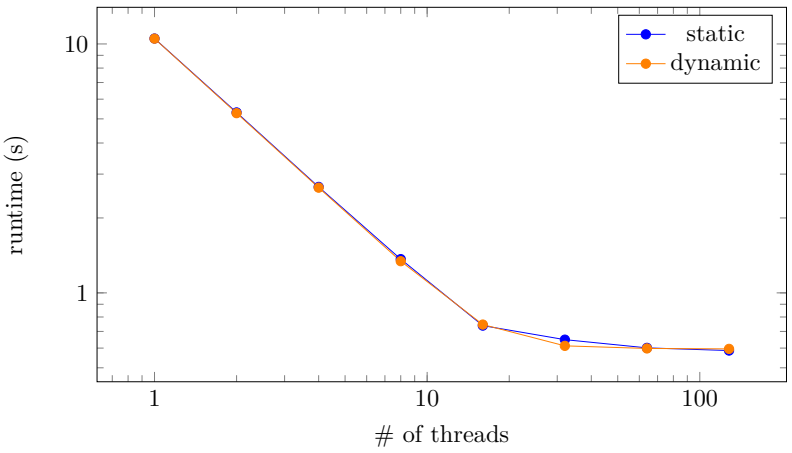
(a) $\mathbf{x} \in S_A = \mathbb{R}^2_{(-1,1)}$



(b) $N = 400^2$, $K_{max} = 100000$, $\mathbf{x} \in S_A$



(c) $\mathbf{x} \in S_B = \left(\mathbb{R}^2_{(-2,2)} + (-0.7, 0)\right)$



(d) $N = 400^2$, $K_{max} = 100000$, $\mathbf{x} \in S_B$

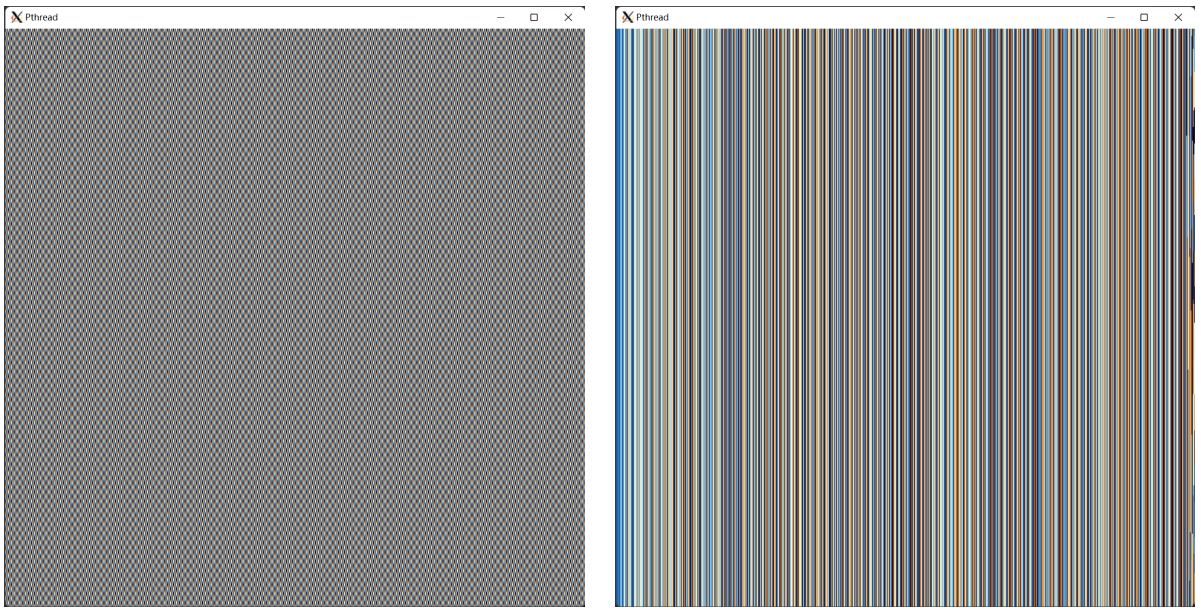
Figure 16: Comparison on performance between dynamic and static approaches, when the problem is different in sample unfairness, the result shows little difference in runtime.

Number of therads (N_t)	1	2	4	8	16	32	64	128
Static, $\mathbf{x} \in S_A$	62.113852	31.204853	15.570008	7.813102	4.219854	3.760015	3.534236	3.443848
Dynamic, $\mathbf{x} \in S_A$	38.550508	19.287333	9.65817	4.839764	2.620898	2.13823	2.1287	2.123165
Static, $\mathbf{x} \in S_B$	10.516374	5.30785	2.664892	1.366432	0.738744	0.648797	0.601302	0.586557
Dynamic, $\mathbf{x} \in S_B$	10.504283	5.272265	2.646427	1.340559	0.745969	0.612668	0.598065	0.59565

Table 6: Runtime Data (unit: s) of static & dynamic pthread program with respect to number of threads under fixed sample size and difference sampling points

Although theoretically the dynamic approach will perform better for it keeps the fairness between threads, the result seems no advantage to static approach. This phenomenon reveals that

enhancing fairness is not necessary in this program. By observation on the threads' behaviours, the reason might be there is originally little imbalance between threads' workload.



(a) Static (b) Dynamic

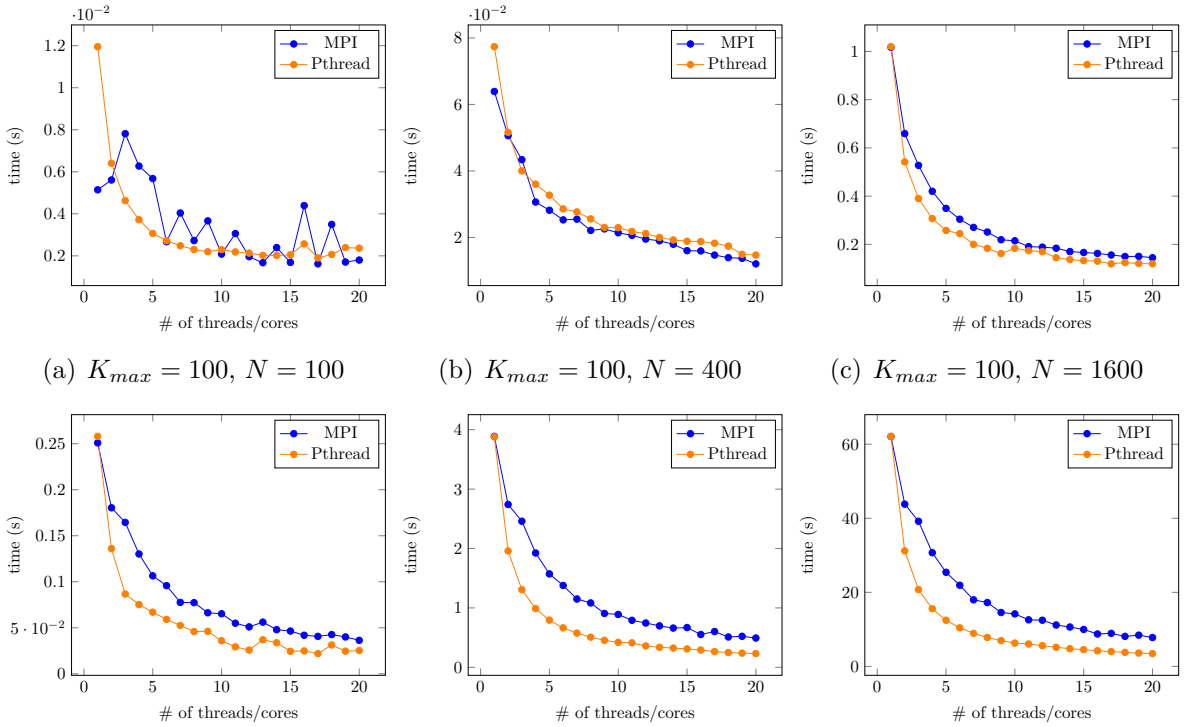
Figure 17: Same color of pixels implies they are computed by the same thread (worker).

From the visualization of Fig 17(a), the work of one thread is evenly scattered in the sampled works. So there is less chance for one worker to have significant more work than others. And from 17(b), the dynamic approach, is assigning the job in a larger period to avoid penalty from massive communication (on the right edge of 17(b), it can be seen that works are indeed addigned dynamically).

```
12  void* worker(void* arg) {
13      Args *args = (Args *) arg;
14      for(int i = args->tid; i < total_size; i += args->num_threads){
15          compute(data + i);
16      }
17      pthread_exit(NULL);
18  }
```

Figure 18: each worker is iterating through the whole sample with fixed step size, this is the key to the resulting static balanced workload

3.4 Comparison on Pthread & MPI Implementaions



(a) $K_{max} = 100$, $N = 100$ (b) $K_{max} = 100$, $N = 400$ (c) $K_{max} = 100$, $N = 1600$

(d) $K_{max} = 10000$, $N = 100$ (e) $K_{max} = 10000$, $N = 400$ (f) $K_{max} = 10000$, $N = 1600$

Figure 19: Comparison on runtime (unit: s) between MPI and Pthread implementations

In order to fairly compare the running time of MPI and Pthread implementations, the same range of processors, i.e., 1-20 processors, are used here.

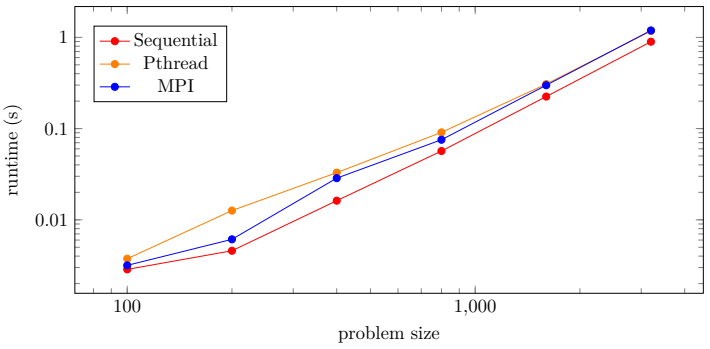
The results show that MPI is less efficient than pthread for the same number of processors, and the running speed of MPI fluctuates more with the number of processors. First, when the number of processors is 1, MPI may have a slight advantage over pthread, which may be related to the compilation optimization in the MPI implementation, but when the number of processors increases, the runtime of pthread decreases significantly and quickly becomes less than the runtime of MPI with the same amount of data. Secondly, MPI's runtime is very unstable at small data volumes, for example, as shown in 19(a), MPI's runtime fluctuates a lot, which may be related to MPI's need to massively merge data at the end of the computation. In contrast, the Pthread program's runtime varies steadily and converges to a very small runtime regardless of the configuration.

(N_t/N_p)	1	2	3	4	5	6	7	8	9	10
M, 100, 100^2	0.0051	0.0056	0.0078	0.0063	0.0057	0.0027	0.0040	0.0027	0.0037	0.0021
P, 100, 100^2	0.0120	0.0064	0.0046	0.0037	0.0031	0.0027	0.0025	0.0023	0.0022	0.0023
M, 100, 400^2	0.0639	0.0506	0.0434	0.0307	0.0282	0.0253	0.0255	0.0221	0.0226	0.0214
P, 100, 400^2	0.0774	0.0516	0.0400	0.0360	0.0327	0.0286	0.0277	0.0256	0.0230	0.0229
M, 100, 1600^2	1.0171	0.6592	0.5276	0.4203	0.3491	0.3044	0.2704	0.2512	0.2191	0.2147
P, 100, 1600^2	1.0199	0.5418	0.3902	0.3069	0.2576	0.2443	0.2001	0.1837	0.1616	0.1839
M, 10000, 100^2	0.2509	0.1804	0.1645	0.1302	0.1064	0.0958	0.0776	0.0773	0.0663	0.0652
P, 10000, 100^2	0.2580	0.1360	0.0867	0.0752	0.0669	0.0591	0.0526	0.0459	0.0462	0.0359
M, 10000, 400^2	3.8879	2.7414	2.4595	1.9245	1.5724	1.3779	1.1497	1.0835	0.9065	0.8900
P, 10000, 400^2	3.8798	1.9593	1.3065	0.9885	0.7946	0.6633	0.5757	0.5075	0.4543	0.4197
M, 10000, 1600^2	62.0604	43.8185	39.1934	30.7025	25.4174	21.9097	17.9803	17.2446	14.5571	14.1820
P, 10000, 1600^2	62.1139	31.2049	20.7288	15.5700	12.4661	10.4112	8.9248	7.8131	6.9622	6.3006

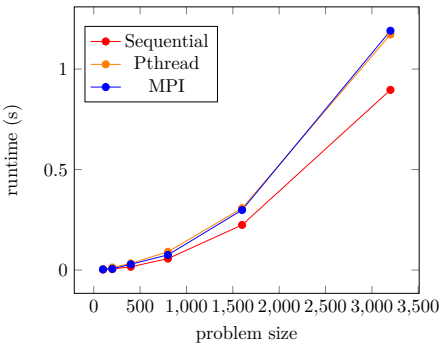
(N_t/N_p)	11	12	13	14	15	16	17	18	19	20
M, 100, 100^2	0.0031	0.0020	0.0017	0.0024	0.0017	0.0044	0.0016	0.0035	0.0017	0.0018
P, 100, 100^2	0.0022	0.0021	0.0020	0.0020	0.0021	0.0026	0.0019	0.0021	0.0024	0.0024
M, 100, 400^2	0.0206	0.0195	0.0190	0.0180	0.0160	0.0160	0.0147	0.0139	0.0137	0.0121
P, 100, 400^2	0.0218	0.0212	0.0200	0.0192	0.0189	0.0188	0.0183	0.0174	0.0149	0.0147
M, 100, 1600^2	0.1906	0.1885	0.1840	0.1700	0.1663	0.1624	0.1558	0.1497	0.1502	0.1446
P, 100, 1600^2	0.1736	0.1697	0.1447	0.1365	0.1327	0.1303	0.1190	0.1246	0.1203	0.1194
M, 10000, 100^2	0.0550	0.0510	0.0561	0.0479	0.0465	0.0419	0.0407	0.0426	0.0400	0.0364
P, 10000, 100^2	0.0292	0.0257	0.0369	0.0337	0.0244	0.0248	0.0219	0.0313	0.0246	0.0253
M, 10000, 400^2	0.7913	0.7455	0.6974	0.6604	0.6688	0.5521	0.6022	0.5105	0.5220	0.4930
P, 10000, 400^2	0.4119	0.3594	0.3367	0.3237	0.3095	0.2910	0.2650	0.2489	0.2386	0.2306
M, 10000, 1600^2	12.5822	12.4737	11.1784	10.6298	9.9798	8.7573	8.9301	8.1171	8.4396	7.7997
P, 10000, 1600^2	6.0441	5.5987	5.1625	4.7813	4.5189	4.2199	3.9946	3.7967	3.5966	3.4565

Table 7: the leftmost column shows the abbreviation of the configurations in the form of "type, K_{max}, N", e.g. "M, 100, 100^2" represent the row is the runtime under MPI implementation, $K_{max} = 100$ and $N = 100^2$

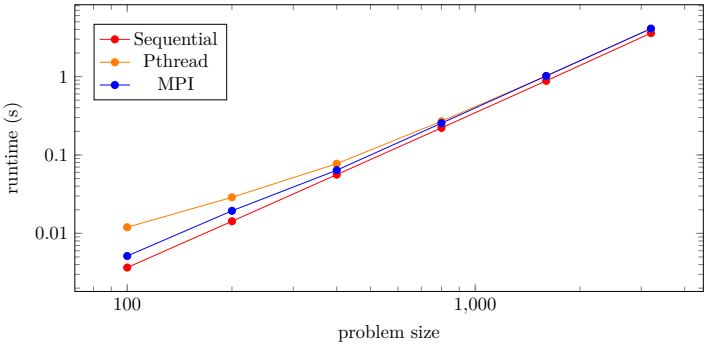
3.5 Comparison on Sequential & Parallel Implementaions



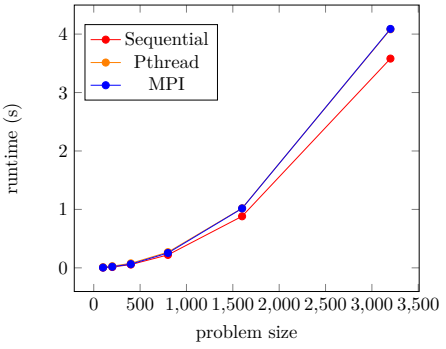
(a) logarithmic axis, $K_{max} = 10$



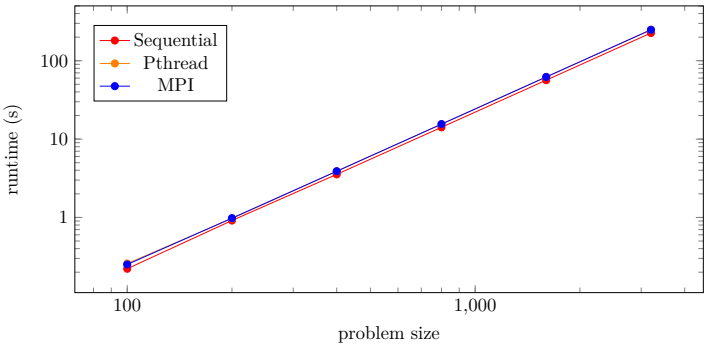
(b) ordinary axis, $K_{max} = 10$



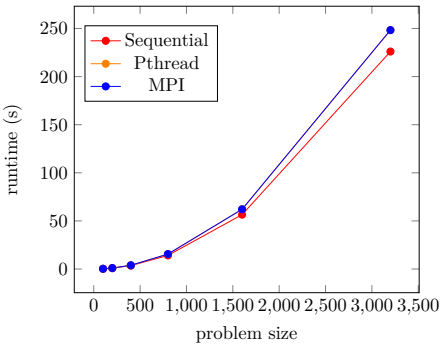
(c) logarithmic axis, $K_{max} = 100$



(d) ordinary axiss, $K_{max} = 100$



(e) logarithmic axis, $K_{max} = 10000$



(f) ordinary axis, $K_{max} = 10000$

The plots shows that when reducing the workers used in pthread and MPI programs to 1, i.e. work sequentially, their runtime is quite similar to that of a naive sequential program. This also well explains the speedup above are mainly from parallelization but not the compilation.

4 Conclusion

In this assignment, I used three methods (MPI, Pthread (static) and Pthread (dynamic)) to try to parallelize the computation of the Mandelbrot set and counted the running time under different configurations. Broadly speaking, there are 4 conclusions as follows:

1. Parallelization can reduce runtime for both large and small scale data, and this conclusion is more solid for large scale data and more processors with greater runtime acceleration.
2. MPI programs are not effective in speeding up small-scale problems because they take a lot of time to transfer the results to the main processor, and the increase in the number of processors is not effective in speeding up the computation.
3. For Pthread programs, the parallelization process does not add a lot of data transfer time, so the acceleration effect is more obvious, and in this problem, dynamic task assignment cannot effectively improve the running speed because the phenomenon of imbalance workload between threads is rare.
4. Both MPI and Pthread programs run with comparable efficiency to naive sequential programs when using 1 processor, which ensures the validity of the above conclusions.

Also at the end, I would like to thank TAs for providing the code framework that allowed me to finish this assignment quickly and keep the code clear and accurate.

Appendix A. How to Compile & Run the Program

Compile:

There are `comile_all_gui.sh compile_all.sh` in `src/` folder. To compile all the programs without GUI, run:

```
$bash compile_all.sh
```

or using compilation commands:

- sequential: `mpic++ mpi.cpp -o mpi -std=c++11`
- mpi: `g++ sequential.cpp -o seq -O2 -std=c++11`
- static pthread: `g++ pthread.cpp -lpthread -o pthread -O2 -std=c++11`
- dynamic pthread: `g++ pthread_dynamic.cpp -lpthread -o dpthread -O2 -std=c++11`

To compile all the programs with GUI, run:

```
$bash compile_all_gui.sh
```

or use compilation commands:

```
# MPI:
$mpic++ -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm \
    mpi.cpp -o mpig -DGUI -std=c++11

# Sequential:
$g++ -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm \
    sequential.cpp -o seqg -DGUI -O2 -std=c++11

# Pthread (Static)
$g++ -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm \
    -lpthread pthread.cpp -o pthreadg -DGUI -O2 -std=c++11

# Pthread (Dynamic)
$g++ -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm \
    -lpthread pthread_dynamic.cpp -o dpthreadg -DGUI -O2 -std=c++11
```

Run without GUI:

- Sequential: `./seqg <X_RESN> <Y_RESN> <max_iteration>`
- MPI version: `mpirun -np <num_proc> ./mpi <X_RESN> <Y_RESN> <max_iteration>`
- Static Pthread version: `./pthread <X_RESN> <Y_RESN> <max_iteration> <n_thd>`
- Dynamic Pthread version:`./dpthread <X_RESN> <Y_RESN> <max_iteration> <n_thd>`

Run with GUI:

- Sequential: `./seqg <X_RESN> <Y_RESN> <max_iteration>`
- MPI version: `mpirun -np <num_proc> ./mpig <X_RESN> <Y_RESN> <max_iteration>`
- Static Pthread version: `./pthreadg <X_RESN> <Y_RESN> <max_iteration> <n_thd>`
- Dynamic Pthread version: `./dpthreadg <X_RESN> <Y_RESN> <max_iteration> <n_thd>`

Demo Output

1. Sequential:

```
[root@Hrimfaxi src]# ./seq
Student ID: 120090562
Name: Derong Jin
Assignment 2 Sequential
Run Time: 0.220962 seconds
Problem Size: 1000 * 1000, 100
Process Number: 1
```

2. MPI:

```
[root@Hrimfaxi src]# mpiexec -np 2 ./mpi
Student ID: 120090562
Name: Derong Jin
Assignment 2 MPI
Run Time: 0.156980 seconds
Problem Size: 1000 * 1000, 100
Process Number: 2
```

3. Pthread (Static)

```
[root@Hrimfaxi src]# ./pthread
Student ID: 120090562
Name: Derong Jin
Assignment 2 Pthread (Static)
Run Time: 0.064148 seconds
Problem Size: 1000 * 1000, 100
Thread Number: 4
```

4. Pthread (Dynamic)

```
[root@Hrimfaxi src]# ./dpthread
Student ID: 120090562
Name: Derong Jin
Assignment 2 Pthread (Dynamic)
Run Time: 0.061214 seconds
Problem Size: 1000 * 1000, 100
Thread Number: 4
```