CS267 Project - Load Balancing in Fractal Art

Bo Pang and Zoey Liaowang Zou

May 2021

1 Introduction

Fractal arts are visually appealing but computational intense to generate. In this project, we seek to explore the generation of two kinds of fractal art: Mandelbrot set and Julia set. Figure 1 shows the example Mandelbrot set and Julia set.

Given an iterative algorithm on complex number z and c: $z_{n+1} = z_n^2 + c$ with $z_0 = 0$, the Mandelbrot set is the set of c that z_n does not diverge. To generate the plot, we try run the algorithm on each c (i.e., each pixel) located in the complex plane (i.e., our picture frame), and color each c depends on how the corresponding z changes over the iterations (e.g., white=converge; black/grey=fast/slow diverge).

Similarly, the Julia set is the set of z_0 that z_n does not diverge, given a fixed c. Each pixel thus represents each sample of z_0.

Since the calculation of each pixel (either c or z_0) is independent, we can easily parallelize the generation of each picture. However, each pixel requires going through the iterative process described above, making it difficult for load balancing - even neighboring pixels may take completely different amounts of iterations to determine their convergence.

In this project, we explored a variety of load balancing techniques when parallelizing the generation of individual fractal art frames. We also considered

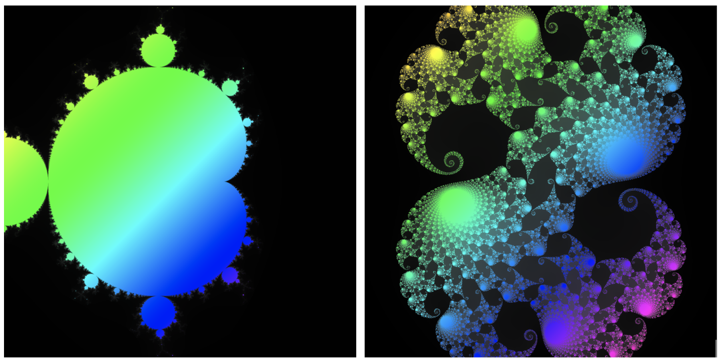

Figure 1: Examples of Mandelbrot set(left) and Julia set(right)

a popular use case - fractal art animation, which requires generating a large number of frames in series (each has a small delta on some parameters); and looked into load-balancing opportunities in that context.

2 Project Context

We worked in a team of 2, consisting of Bo Pang and Zoey Zou. We are both interested in exploring digital art that combines mathematical rigor and human creativity, and we are curious to see how parallel computing can make digital art more accessible via reduced computation time for real-time generation and display of images. This is a standalone project and not part of a larger research effort.

3 Algorithm and implementation

3.1 Fundamental workflow

Our parallelization and load-balancing methods are based on the fundamental workflow shown in Algorithm 1. To generate an animation, Step 1 determines how each frame should look like (for example, decreasing the numerical span creates zoom-in effects, sweeping the variable c changes the shape of Julia set, etc). Steps 2 to 8 generate individual frames.

Algorithm 1 Generate fractal art animation

```
 1: for each frame do
 2:     for each row in current frame do
 3:         for each pixel (column) in current row do
 4:             Run the iterative algorithm (either Julia or Mandelbrot)
 5:             Map convergence data to pixel color in RGB
 6:             Save current pixel to frame buffer
 7:         end for
 8:     end for
 9:     Append frame buffer to file or display
10: end for
```

We only focused on parallelizing the generation of individual frames (Step 2 - 8) rather than across frames (Step 1), because Steps 1 and 9 are naturally serial (e.g., the animation is presented to the user frame by frame) and usually involves non-deterministic change of parameters (e.g., an interactive software that allows user to keep changing the perspective while rendering the frame).

Given that, Step 9 serves as an implicit barrier. Thus, a well-balanced parallel workflow for Step 2 - 8 is important to minimize the overall execution time.

3.2 Naive parallelization (Shared memory)

The “naive” method is straightforward. We simply distributed the rows (in Step 2) of current frame / canvas equally for each thread. Figure 2 (left) illustrates the divide of work among 4 threads.

We expect this method to perform poorly in terms of load balance. For example, in Figure 2 (left), thread 1 and 2 owns more bright areas than thread 0 and 3, thus taking much more iterations, or amount of time, to finish.

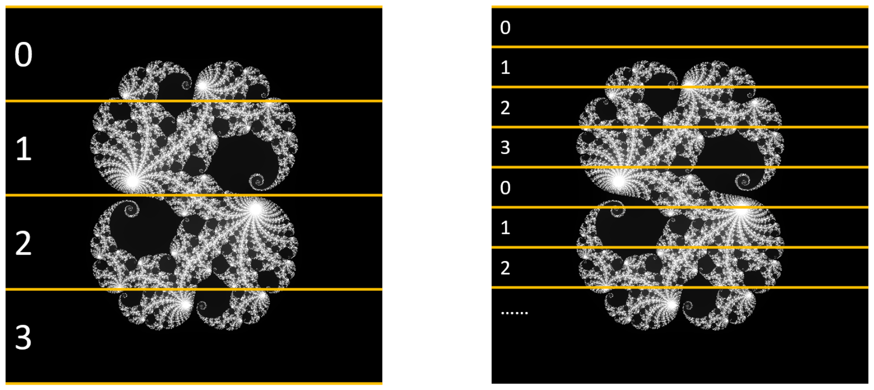

Figure 2: Work allocation by thread for naive parallelization (left) and interleaved parallelization (right)

3.3 Interleaved parallelization (Shared memory)

The method is a small improvement over the naive method. Instead of assigning a continuous large region (rows) to each thread, we assign each thread multiple small regions (rows) across the canvas. Figure 2 (right) illustrates this improvement.

We expect this method to perform much better than the previous one in most cases, given that the bright (less divergent) areas are usually clustered together. The probability that those areas happen to align with our row distribution and overload one or two threads is very low.

3.4 Work Claiming (Shared memory)

In this method, we use a shared variable to track the current progress, and let each thread to ‘claim’ a new work when it finishes its current work via atomic operations. Figure 3 illustrates this approach.

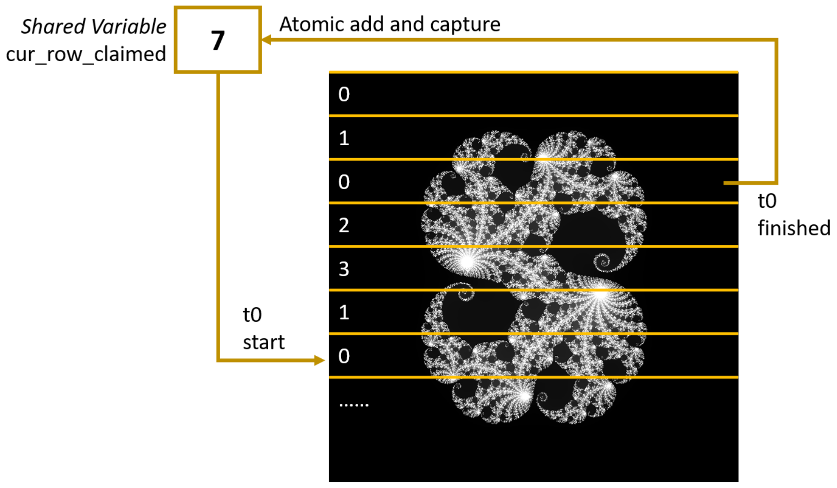

Figure 3: Atomic add and capture for thread 0 as it finishes its current work at row 3 and claims row 7

We expect this method to provide near-perfect work balancement, while acknowledging the existence of overhead due to atomic operations and memory access. The amount of rows each thread can claim in each atomic operation is adjustable. We tuned that for an optimal balance between overhead and flexibility during our experiment on KNL, realized that the atomic operations are so efficient that the overhead is negligible even we let each thread claim one row at a time.

3.5 Inter-frame reference (Shared memory)

This method relies on the information we gathered in the previous frame, to infer a load balancing strategy for the next frame. Our implementation is simple - when generating the previous frame, each thread records the calculation intensity for each row it worked on. After that, each thread iterates through the intensity vector, and assigns itself a chunk of row that will lead to a near-average amount of work. Figure 4 illustrates this workflow. This Animation illustrates the distribution of works when generating a 150-frame Julia set using this technique.

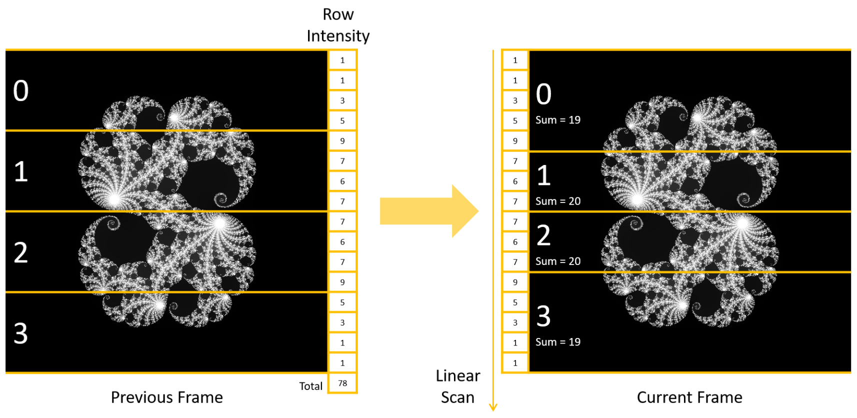

Figure 4: Work reallocation based on the calculation intensity of the previous frame

Currently, we used the total amount of iterations to process each row pixel as the "intensity" value. This is adequate as a proof-of-concept implementation; but we acknowledged that the total amount of iterations is not 100% correlated with execution time - there are other overheads, such as memory writes and color mappings when processing each pixel. Besides, the overhead of scanning that intensity vector is non-negligible.

We expected this method to be more effective when the changes between frames are small (in other words, a high frame rate animation with small amount of motion).

3.6 Dispatcher-Worker (Distributed memory)

We implemented this workflow in MPI. Rank 0 is always the dispatcher, holds the data for the entire frame; Rank 1+ are workers. While Rank 0 receives results from individual workers, it saves the result and allocates new work to that worker. The workflow is bootstrapped by sending each worker a piece of work to do, and is ended by sending a magic number to each worker.

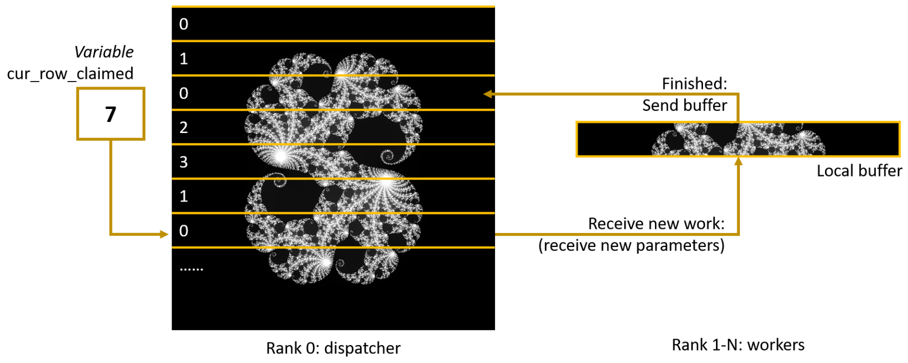

Figure 5: Rank 0 serves as the dispatcher and uses a variable to keep track of current work status; Rank 1 to N serve as workers and receive new work allocation from the dispatcher

Without precisely crafted latency-hiding techniques, we expect this method to be less efficient than "work claiming" using shared memory, but should achieve a similarly well-balanced work distribution. Similar to "work claiming", we also tuned the amount of rows carried by each transaction to find an optimal balance between network overhead and flexibility. The optimal value turned out to be around 15 rows according to our experiments.

3.7 Work claiming with UPC++ (Distributed memory)

We utilized Partitioned Global Address Space (PGAS) memory model of UPC++ and implemented work claiming in distributed memory. By creating a global pointer to keep track of the latest row claimed and using *rput* operations to directly send rgb values after calculations, we were able to free Rank 0 from acting as a dispatcher. The implementation is very similar to work-claiming in shared memory (Section 3.4), except for the creation of global pointers for shared global memory space for *cur_row_claimed* and arrays to hold rgb values.

We expect this method to perform faster than the dispatcher-worker method while keeping the advantage for inter-node scaling. The asynchronous nature of the one-sided communication model also allows better latency hiding.

4 Experiment Results

4.1 Shared Memory

We compare the parallelization performance of different load balancing techniques in shared memory using OpenMP. For benchmarking, we generate 2000 x 2000 Julia animations with 150 frames using 64 threads. We focus on two metrics: 1) total run time; 2) unbalanced and overhead % calculated using

$\frac{t_{max}-t_{avg}}{t_{avg}}$, where t_{max} is the time taken by the slowest thread and t_{avg} is the average run time per thread. Figure 6 compares the performance of static work allocation (naive parallelization and interleaved parallelization) and dynamic load balancing techniques (work claiming and inter-frame reference).

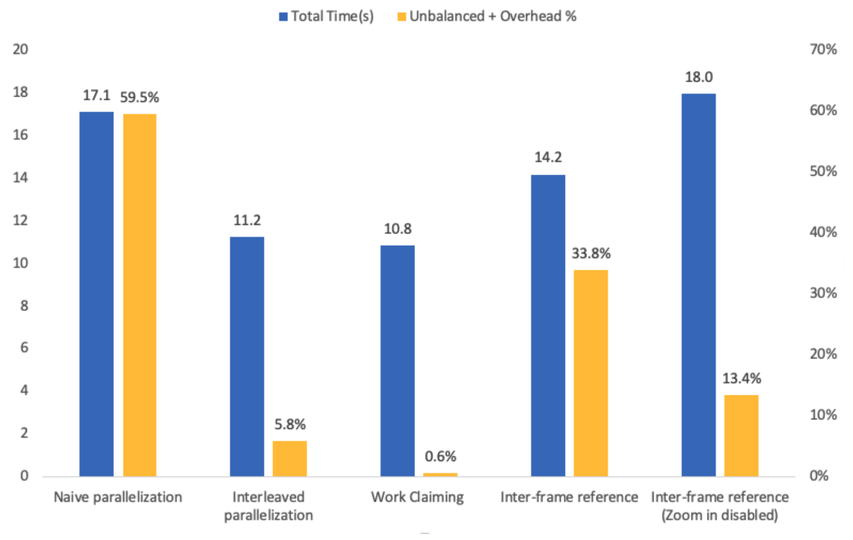

Figure 6: Total execution time (in seconds) and unbalanced and overhead % comparison for different shared memory implementations using 64 threads to generate a 2000 x 2000 Julia animation with 150 frames

Using naive parellelization as the baseline, we can see that work claiming has the best performance with only 0.6% imbalance, followed by interleaved parallelization and inter-frame reference. Their performances largely align with our hypothesis. Nonetheless, we are surprised by how well interleaved parallelization performs, given it does not involve any dynamic load balancing. This is mainly due to the general pattern of Mandelbrot and Julia sets. In addition, our implementation does not require any reads during computation for each thread, eliminating cache misses. We are also surprised that inter-frame reference brings little improvement to the baseline. This suggests that computation intensity per row varies quite a bit between frames, so the "intensity" value calculated using the previous frame is no longer applicable to the current frame. This is especially the case when we have a "zoom in" animation. Hence, when we disable the zoom feature of the animation, load balancing for inter-frame reference improves, as seen in the rightmost bar in Figure 6.

Figures 7 and 8 show the strong scaling performance for the top 2 techniques (work claiming and interleaved parallelization). Here, we generate Mandelbrot plots as load imbalance is more pronounced in Mandelbrot sets. Both methods exhibit close to linear scaling in run time. In terms of load balancing, work claiming performs better, especially with threads larger than 48.

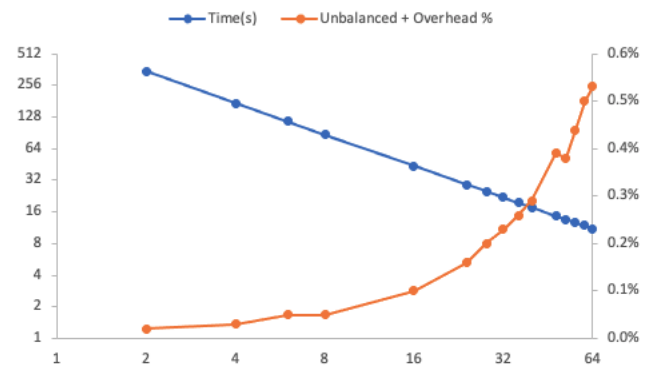

Figure 7: Total run time in seconds and unbalanced % (y-axis) vs. number of threads (x-axis) for **work claiming** to generate a 2000 x 2000 Mandelbrot animation with 150 frames using 2 to 64 threads

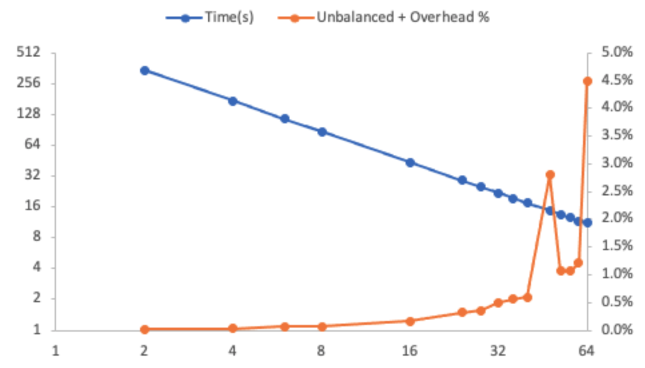

Figure 8: Total run time in seconds and unbalanced % (y-axis) vs. number of threads (x-axis) for **interleaved parallelization** implementation to generate a 2000 x 2000 Mandelbrot animation with 150 frames using 2 to 64 threads

4.2 Distributed Memory

In distributed memory, we implemented the dispatcher-worker method as described in Section 3.6 and a naive static allocation (blocked allocation, no load balancing) as the baseline. Their performance comparison is shown in Figure 9.

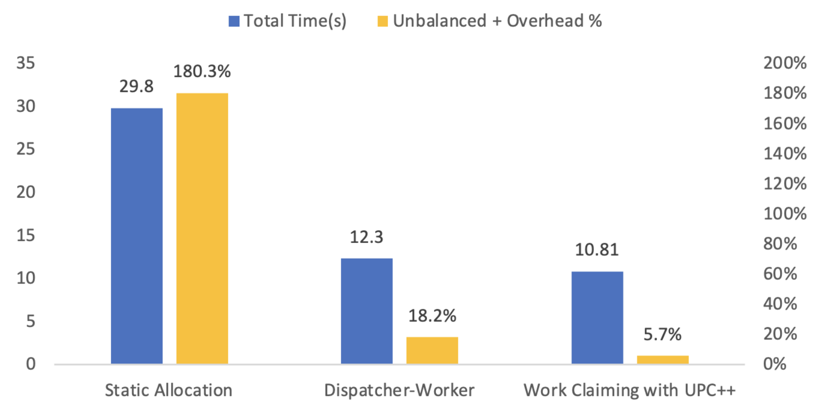

Figure 9: Total execution time (in seconds) and unbalanced and overhead % comparison for different distributed memory implementations using 64 tasks to generate a 2000 x 2000 Julia image with 150 frames

The dispatcher-worker implementation brings significant performance improvements as it reduces the load imbalance from 180.3% to 18.2%. Nonetheless, the communication overhead in MPI is still higher than that of work-claiming in shared memory. As we make the job larger (i.e, generating animation with more frames and higher resolution), we would observe better dispatcher-worker efficiency as the job can be distributed across nodes. Work claiming with UPC++ brings additional performance improvement and the total run time is similar to work-claiming in shared memory.

Figure 10 shows the strong scaling performance for dispatcher-worker and work claiming in UPC++. The first 2 to 64 processes are run on a single node and are used to show intra-node scaling. 128 to 256 processes are run on 2 to 4 nodes with 64 tasks per node and are used to show inter-node scaling.

We observe similar intra-node scaling for these two methods (dispatcher-worker and work claiming in UPC++). However, the inter-node scaling for UPC++ is significantly better.

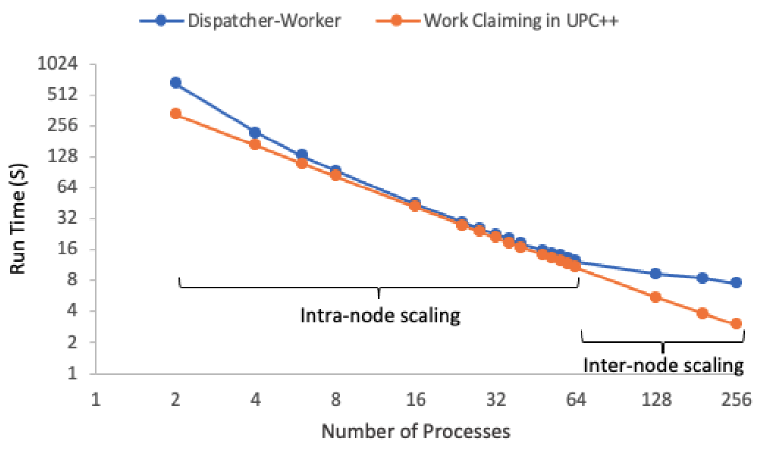

Figure 10: Total run time in seconds (y-axis) vs. number of processes (x-axis) for dispatcher-worker and work claiming implementations to generate a 2000 x 2000 Julia image with 150 frames using 2 to 256 processes

5 Discussion

Overall, we can see that load balancing plays a vital role in fractal art's parallel generation. It is expected that the most efficient two methods (work claiming and interleaved static work assignment) are all based on shared memory. Methods based on distributed memory are relatively less efficient but have the potential to scale up across nodes, as seen in the inter-node scaling performance using UPC++.

Some methods let each thread or worker owns fragmented sections of each frame (i.e., interleaved static allocation and work claiming). Though that is not a problem in our experiment, we might encounter inter-pixel dependencies, for example, when applying an anti-aliasing filter. In that case, methods that let the worker own a continued section of a frame (i.e., inter-frame reference method) may be beneficial.

If time allows, we would like to further enhance our load-balancing methods, via cache-aware programming for shared memory and latency-hiding for distributed memory. There are many other kinds of fractal arts for us to explore, each with different computing characteristics and visual appearance.