SW401 - Fall 2025

Parallel & Distributed Computing

– Phase 1 Report –

Team Members:

Salma Ayman 202201191
Malak Ahmed 202200812
Elhusseain Aboulfetouh 202202239
Mohaned Atef 202100383

Problem Statement and Baseline Design

The project's main objective is to compute and render the Mandelbrot/Julia set which is a well-known mathematical algorithm defined by iterating the recurrence:

$$z(n + 1) = z(n)^2 + c$$

Where $c = x + iy$ is a point in the complex plane that's associated with each pixel of the output image.

For every pixel, we repeat the recurrence until either of the following conditions are met:

- $|z(n)| > 2$ which means that the sequence diverges
- A fixed maximum iteration count is reached

Coloring the pixel is done by the number of iterations before the divergence, which produces the characteristic fractal image. This computation is labeled as "embarrassingly parallel" in many textbooks. That's because each pixel is independent and doesn't interact with other pixels. However, in the baseline (sequential) version, the entire image is computed on a single loop that's executed by 1 CPU core.

Baseline Algorithm

1. Map each pixel (px, py) to a complex coordinate (x0, y0)
2. Iterate the recurrence until escape or max iterations
3. Store the iteration count into an image buffer
4. Write the buffer to a .ppm image file

The time complexity for this algorithm is $O(width\ x\ height\ x\ number\ of\ iterations)$ because the loop over the pixel grid is what dominates the runtime.

Baseline Implementation Characteristics

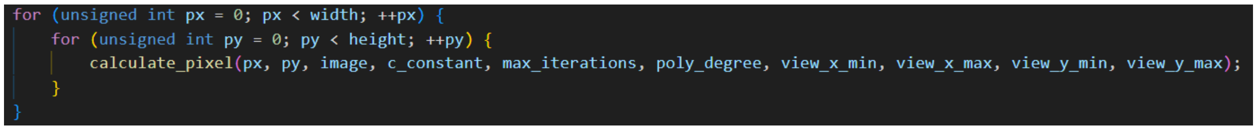
1. Language → C++
2. Major computation → Nested loop over pixels
3. Output format → ppm image file

Parallelization approach and OpenMP directives

Since each pixel can be computed independently, the Mandelbrot program is a standard parallelization program. The outer loop over the image rows (or columns) is a main parallelization point because of the following reasons:-

- No shared writable variables inside the loop
- Each iteration writes to a unique index in the output buffer
- No synchronization is needed between iterations

Our parallel implementation extends the sequential version by distributing pixel computations across multiple CPU cores using OpenMP. In the baseline version, the program renders the fractal by iterating over each pixel in 2 nested loops.

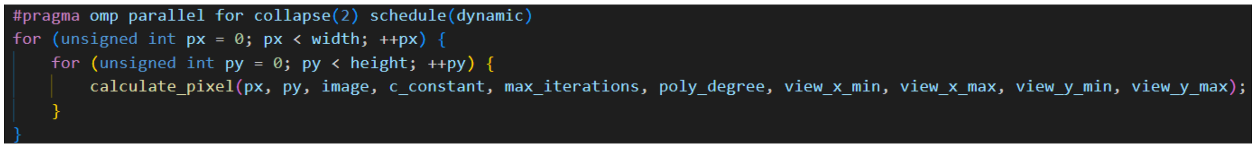


```
for (unsigned int px = 0; px < width; ++px) {
    for (unsigned int py = 0; py < height; ++py) {
        calculate_pixel(px, py, image, c_constant, max_iterations, poly_degree, view_x_min, view_x_max, view_y_min, view_y_max);
    }
}
```

Each call to calculate_pixel() performs the fractal iterations, maps the escape count to a color, and writes the result into the output image buffer (sf::Image). Pixels don't share state or data, which means that the loop can be parallelized safely with OpenMP.

OpenMP Directives

Parallelization is implemented inside the ParallelCalculator::calculate_polynomial() function by using the following pattern:



```
#pragma omp parallel for collapse(2) schedule(dynamic)
for (unsigned int px = 0; px < width; ++px) {
    for (unsigned int py = 0; py < height; ++py) {
        calculate_pixel(px, py, image, c_constant, max_iterations, poly_degree, view_x_min, view_x_max, view_y_min, view_y_max);
    }
}
```

- parallel for: spawns worker threads and distributes loop iterations across these threads
- collapse(2): flattens the 2 nested loops into a single parallel loop which improves the workload balance and allows OpenMP to schedule work over the 2D domain
- schedule(type) allows runtime selection of scheduling method (whether static, dynamic or guided)
- The number of threads is explicitly set using omp_set_num_threads(numThreads) which then allows for more experimentation while running the SFML Window Drawer

Scheduling Policies

Our implementation supports 3 scheduling modes, allowing for runtime selection:

1. static: even block distribution and lowest overhead. It's considered the best strategy when the workload per pixel is fixed
2. dynamic: threads request work in chunks. It's considered a good choice when the pixels vary in iteration counts (uneven work)
3. guided: chunk size starts big and shrinks over time. It's a hybrid approach the mixes between static and dynamic scheduling for better handling of tradeoffs

Thread Safety Considerations

- Each thread writes to a unique pixel location so there’s no risk of race conditions
- calculate_pixel() only does arithmetic and returns a color, so there’s no modification on shared variables
- No OpenMP reduction or synchronization strategies are needed because there’s no inter-pixel communication in the algorithm

Additionally, we used omp_get_wtime() to measure the parallel runtime.

Performance Table

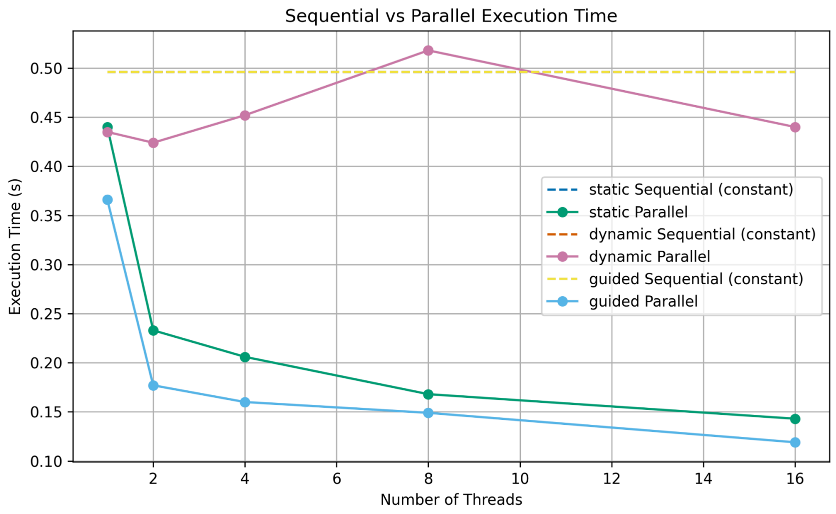
Sequential Time: 0.496 seconds

```
Calculation took 0.496 seconds
```

Schedule	Threads	Time	Speedup	Efficiency (%)
Static	1	0.44	1.12727	112.727
Static	2	0.233	2.12875	106.438
Static	4	0.206	2.40777	60.1941
Static	8	0.168	2.95238	36.9048
Static	16	0.143	3.46853	21.6783
Dynamic	1	0.435	1.14023	114.023
Dynamic	2	0.424	1.16981	58.4906
Dynamic	4	0.452	1.09734	27.4336
Dynamic	8	0.518	0.957529	11.9691
Dynamic	16	0.44	1.12727	7.04545
Guided	1	0.366	1.35519	135.519
Guided	2	0.177	2.08226	140.113
Guided	4	0.16	3.1	77.4999
Guided	8	0.149	3.32886	41.6107
Guided	16	0.119	4.16807	26.0504

Plots

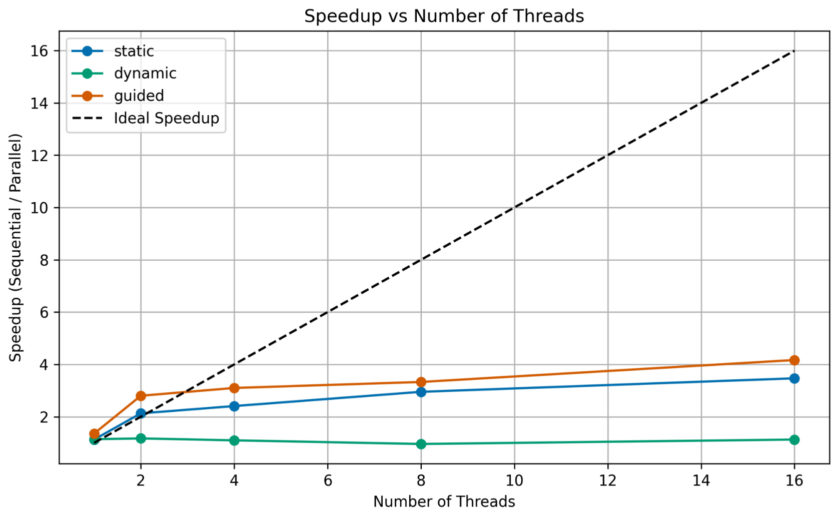
Time Comparison



Sequential Time: 0.496 seconds
Highest Time: 0.518 seconds (Schedule: Dynamic, Threads: 8)
Lowest Time: 0.119 seconds (Schedule: Guided, Threads: 16)

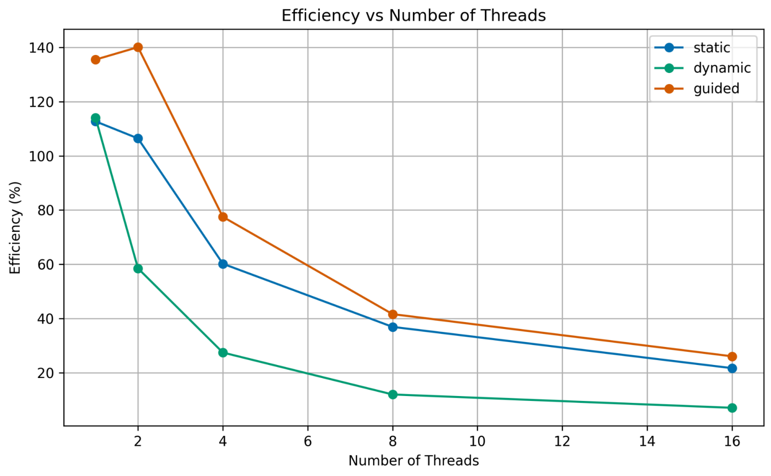
Interpretation: The parallel execution time decreases with increasing the number of threads, static and guided scheduling showed the best results, while the dynamic execution time fluctuates slightly even taking more time than the sequential. This is due to the overhead that was introduced when managing small workloads.
Conclusion: Guided scheduling achieved the best balance between load balancing and communication.

Speedup



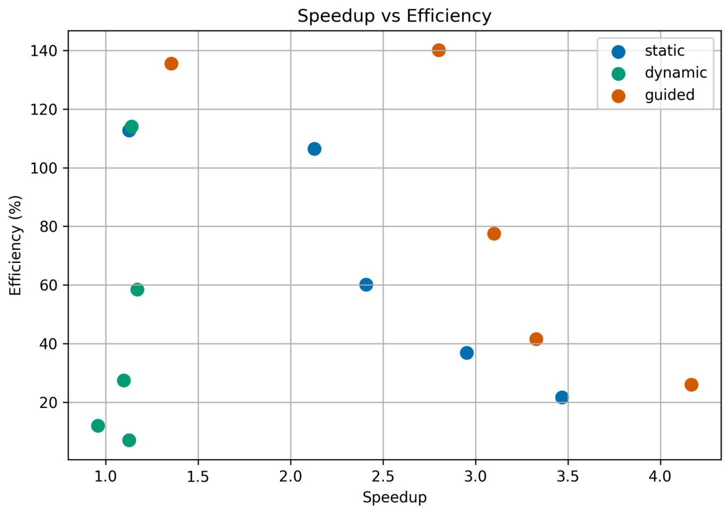
Highest Speedup: 4.16807 (Schedule: Guided, Threads: 16)
Lowest Speedup: 0.957529 (Schedule: Dynamic, Threads: 8)
Interpretation: Dynamic scheduling achieved the worst speedup (almost flat), this is because every pixel's calculation approximately takes the same time; therefore, distributing small tasks and assigning them at a time will cause thread management overhead.
Conclusion: Guided scheduling achieved the best speedup.

Efficiency



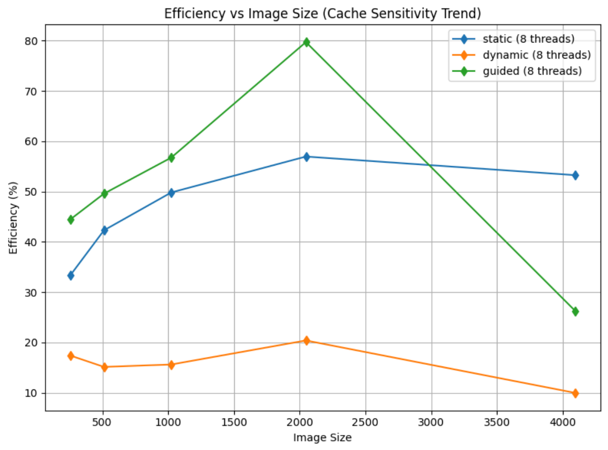
Best Efficiency: Guided
Worst Efficiency: Dynamic
Interpretation: Peak efficiency is at 140% (Guided, t = 2) as it achieved a superlinear speedup (S > p). Static scheduling achieved the second best 120% (Static, t = 2). Dynamic scheduling achieved the worst performance due to the high overhead. After (t = 4), there is a drop in the efficiency due to the overhead and imbalance (sublinear speedup, S < p).
Conclusion: Guided scheduling achieved the best performance and highest efficiency.

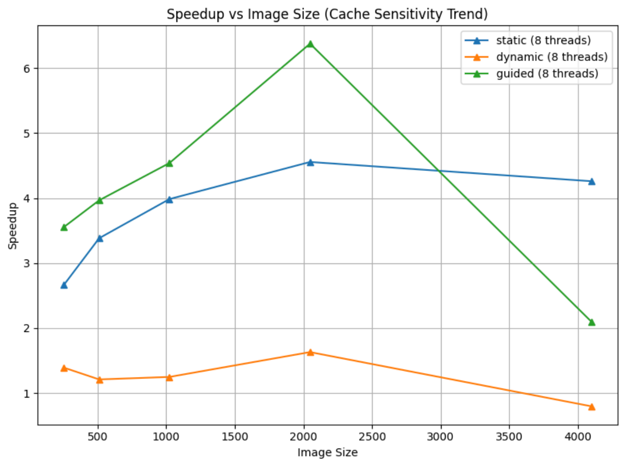
Speedup vs. Efficiency



Cache Sensitivity Test

This measures how the performance changes when increasing the problem size (the image size). Small image sizes means more pixels fitting into the cache line (i.e. increase cache hits) and high performance. As size increases, threads will have to fetch from the main memory.





Interpretation: After size: 2000, there is an evident drop in the speedup and efficiency, this indicates that the pixel sizes had to be fetched from the main memory and this increased the waiting time and overhead.

Discussion of Amdahl's and Gustafson's Analysis

Amdahl's Law

This law is used to give the theoretical upper bound on speedup based on the serial fraction of the code under a fixed workload, where a fraction f of the program is strictly serial and the remaining $(1 - f)$ is parallelizable. In the fractal computation, the dominant computation occurs inside the pixel loop. This means that the serial fraction is very small and the expected speedup should approach linear scaling.

Observed Results

Threads	Static Time (s)	Dynamic Time (s)	Guided Time (s)
1	0.496	0.496	0.496
2	~0.22	~0.45	~0.32

4	~0.18	~0.37	~0.20
8	~0.15	~0.518	~0.14
16	~0.14	~0.44	~0.119

Interpretation

- Static and guided schedules follow Amdahl's law. As thread count increases, execution time decreases, approaching the theoretical speedup curve.
- The dynamic schedule deviates significantly from Amdahl's ideal scenario. At 8 threads, the runtime is slower than the sequential version which indicated that synchronization overhead dominates the cost
- The guided schedule has the lowest execution time which means that it found the best balance between exploiting parallelism and minimizing load imbalance.

Observation

From the static and guided schedules, the effective serial fraction is very small which confirms our interpretation of the Mandelbrot/Julia problem to be highly parallelizable. However, there are some limitations for scaling such as scheduling overhead and load imbalance.

Gustafson's Law

This law is considering an opposite assumption to Amdahl's. It states that instead of fixing the workload, we scale the problem size with the core count. We evaluated this principle by increasing the image size while keeping the number of threads constant (8 threads).

Observed Results

- Speedup and efficiency increase as image size grows from 500 to 2000 pixels. This range almost reaches a size where cache can still hold most data.
- Beyond 2000 pixels, both speedup and efficiency drop drastically, especially for the guided schedule.
- This drop is because the working set no longer fits the cache which causes frequent main memory fetches and cache evictions

Interpretation

- For small image sizes, the workload is too small to utilize the 8 cores fully which limits the parallel efficiency.

- The parallel portion dominates as the problem size increases. The efficiency approaches the ideal value (static ~57%, guided ~80% at size 2000)
- After a threshold, scaling breaks down because memory hierarchy becomes the bottleneck rather than computation

Observation

This experimentation confirms that Gustafson's law only holds up to the point where problem size exceeds cache capacity. However, in real-world applications, there are still some bottlenecks that aren't covered in the simulation.

Reflections: Remaining Bottlenecks

Overall, the parallel version of the Julia Set reduced the execution time compared to the sequential version. Results show that the guided scheduling achieved the best performance as it balanced between communication and load balancing. On the other hand, dynamic scheduling underperformed due to the high overhead from frequent tasks redistribution. As the number of threads increased, the speedup increased until reaching a certain point and plateaued. This was caused by memory bandwidth limit and synchronization overhead.

- Memory and cache: Large image resolutions exceed the cache line capacity, resulting in frequent cache misses. This slows down performance as threads have to access the main memory.
- Thread management overhead: For small image sizes, the overhead of creating threads and managing them becomes significant and greater than the computation time. In such a case, sequential would be more efficient.
- Memory bandwidth saturation: Efficiency drops after 8 threads due to bandwidth saturation. Adding more threads will not improve the performance as they all compete to access the main memory.