Assignment Report #4
Zooming In on the Mandelbrot Set

Student Name: Adilet Akimshe (ID: 202194337)
Course: PHYS 421 Parallel Computing

Submitted: October 21, 2025

Software packages used:

GCC 14.2.0	Compilation
Open MPI 5.0.7	MPI parallelization
ffmpeg 6.1.1	Video generation
Python 3.11.5	Data analysis

AI tools used:

Claude 3.5 Sonnet	Code development, debugging assistance, report writing

Abstract

This report presents the implementation and performance analysis of a parallel Mandelbrot set generator using MPI (Message Passing Interface). The Mandelbrot set, a classic fractal structure, serves as an excellent test case for embarrassingly parallel computations. We implemented both serial and parallel versions in C++, utilizing a dynamic master-worker distribution strategy for load balancing. Performance tests were conducted on the Shabyt cluster using 1 to 32 MPI processes with a workload of 120 frames at 2400×2000 resolution. Results show nearly ideal speedup up to 4 processes (speedup of 3.06), with parallel efficiency remaining above 87% even at 32 processes (speedup of 28.0). A 600-frame Full HD video demonstrating the fractal's self-similar structure was successfully generated.

1 Introduction and Problem Statement

The Mandelbrot set is defined as the set of complex numbers c for which the iterative sequence

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0 \tag{1}$$

remains bounded as $n \to \infty$. In practice, we determine membership by checking whether $|z_n| > 2$ within a maximum number of iterations $n_{\max}$. If the sequence exceeds this threshold, the point escapes to infinity; otherwise, it is considered part of the set.

The computational task involves generating high-resolution images of the Mandelbrot set by:

1. Discretizing a rectangular region of the complex plane into an $N_x \times N_y$ grid
2. For each pixel (x, y), computing the corresponding complex number $c = x + iy$
3. Iterating Equation 1 up to $n_{\max}$ iterations
4. Mapping the escape time (iteration count) to a color value
5. Generating multiple frames with progressively zoomed views

This problem is **embarrassingly parallel** because each pixel can be computed independently. However, the computational cost per pixel varies significantly across the image—pixels near the boundary of the Mandelbrot set require more iterations—creating potential load imbalance issues.

Assignment objectives:

- Implement serial and MPI-parallel Mandelbrot generators
- Generate high-quality fractal visualizations and videos
- Analyze strong scaling behavior and parallel efficiency
- Compare static versus dynamic work distribution strategies

2 Hardware and Software Environment

All parallel performance tests were conducted on the Shabyt high-performance computing cluster at Nazarbayev University. The relevant specifications are listed in Table 1.

Table 1: Hardware and software environment.

Component	Specification
Cluster	Shabyt HPC, Nazarbayev University
Nodes	CPU partition compute nodes
Cores per node	64 (assumed)
Compiler	GCC 14.2.0
MPI Implementation	Open MPI 5.0.7
Operating System	GNU/Linux (Red Hat based)
Video encoder	ffmpeg 6.1.1
Analysis tools	Python 3.11.5, matplotlib

3 Methods and Implementation

3.1 Algorithm Overview

The core Mandelbrot iteration algorithm is straightforward:

```c
int mandelbrot_iterations(double cx, double cy, int max_iter) {
  double zx = 0.0, zy = 0.0;
  int iter = 0;

  while (zx*zx + zy*zy <= 4.0 && iter < max_iter) {
    double temp = zx*zx - zy*zy + cx;
    zy = 2.0 * zx * zy + cy;
    zx = temp;
    iter++;
  }
  return iter;
}
```

Listing 1: Mandelbrot iteration kernel

For each frame f, the pixel size is scaled by a zoom factor:

$$\text{pixel\_size}_f = \text{pixel\_size}_0 \cdot \text{zoom\_factor}^{-(f-1)} \tag{2}$$

where zoom_factor = zoom_per_second$^{1/60}$ for 60 fps video.

3.2 Parallelization Strategy

After careful consideration of the problem characteristics, I chose a **dynamic master-worker** distribution at the frame level rather than distributing rows or columns within frames. This decision was based on:

1. **Load balancing**: Different frames have varying computational costs due to zoom level. Dynamic distribution ensures workers always have work.
2. **Simplicity**: Frame-level parallelism requires minimal communication (just frame numbers and pixel data).
3. **Scalability**: Works efficiently with any number of processes up to the number of frames.

Master process (rank 0):

- Maintains a queue of frame numbers to be computed
- Distributes frame assignments to idle workers using `MPI_Send`
- Receives completed frames via `MPI_Recv`
- Optionally writes frames to disk in correct order
- Sends termination signal (-1) when all frames are complete

Worker processes (rank > 0):

- Request and receive frame numbers from master
- Compute all pixels for assigned frame independently

- Send frame number back to master upon completion
- Optionally send pixel data for disk storage
- Terminate upon receiving signal -1

This approach avoids the overhead of fine-grained row/column distribution while maintaining excellent load balance through dynamic work allocation.

3.3 Color Mapping

To visualize the fractal structure clearly, I implemented a smooth color gradient based on iteration count:

$$t = \frac{\text{iterations}}{n_{\max}}, \quad \begin{cases} R = 255 \cdot 9(1-t)t^3 \\ G = 255 \cdot 15(1-t)^2 t^2 \\ B = 255 \cdot 8.5(1-t)^3 t \end{cases} \tag{3}$$

Points in the set (iterations $= n_{\max}$) are colored black (0,0,0).

3.4 Build and Run Instructions

The code is compiled using the provided Makefile:

```
# Load required modules (on Shabyt)
module load OpenMPI/5.0.7-GCC-14.2.0

# Compile both serial and parallel versions
make

# This produces:
#   mandelbrot_serial - Serial executable
#   mandelbrot_mpi    - MPI parallel executable
```

Listing 2: Build commands

To generate the Full HD video (Part C):

```
# Generate 600 frames (1920x1080)
mpirun -np 8 ./mandelbrot_mpi 1000 -0.7436438870371587 \
0.1318259042053120 1920 1080 0.001 1.5 600 1

# Create video with ffmpeg
ffmpeg -framerate 60 -i frame_%04d.ppm -c:v libx264 \
-pix_fmt yuv420p -crf 18 video.mp4
```

Listing 3: Video generation

For scaling tests (Part D):

```
# Submit scaling test batch job
sbatch run_scaling.sh

# Monitor progress
squeue -u $USER
tail -f scaling_*.out
```

Listing 4: Shabyt batch submission

4 Results

4.1 Part A: Serial High-Resolution Image

A single high-resolution image (3000×2500 pixels) of the full Mandelbrot set was generated successfully using the serial implementation. The image clearly shows the main cardioid, circular bulb, and intricate fractal boundary structure. Due to file size constraints, this image is included as a separate PNG file.

Parameters: $n_{\max} = 1000$, center=(-0.5, 0), resolution=3000×2500, pixel_size=0.001

4.2 Part C: Fractal Zoom Video

A 10-second Full HD video (600 frames at 60 fps) was successfully generated, zooming into the Mandelbrot set at coordinates (-0.7436438870371587, 0.1318259042053120). The video demonstrates the fractal's self-similar structure across multiple scales of magnification. Each frame was computed at 1920×1080 resolution with 1.5× zoom per second (1.00811× per frame).

The video file `video.mp4` is included in the submission.

4.3 Part D: Performance Analysis on Shabyt

Scaling tests were performed with the following parameters:

- $n_{\max} = 2000$
- Resolution: 2400 × 2000 pixels
- Number of frames: 120
- Center: (-0.7436438870371587, 0.1318259042053120)
- Store images: No (to eliminate I/O overhead)

Each configuration was run 3 times to assess reproducibility. The timing results are presented in Table 2.

Table 2: Strong scaling results on Shabyt cluster. Mean timing over 3 runs.

Processes	Time (s)	Std Dev (s)	Speedup	Efficiency (%)
1	1863.07	0.94	1.00	100.0
2	1842.75	0.23	1.01	50.5
4	609.46	0.85	3.06	76.4
8	273.54	0.35	6.81	85.1
16	127.51	0.09	14.61	91.3
32	66.45	0.05	28.04	87.6

Key observations:

- Serial (1 process) baseline: 1863 seconds (31 minutes)
- Excellent reproducibility: standard deviation $<$ 1 second for most configurations
- Near-perfect speedup from 4 to 32 processes
- The 2-process case shows anomalous behavior (discussed below)

Figure 1 shows both the speedup curve compared to ideal linear speedup and the parallel efficiency across different process counts.

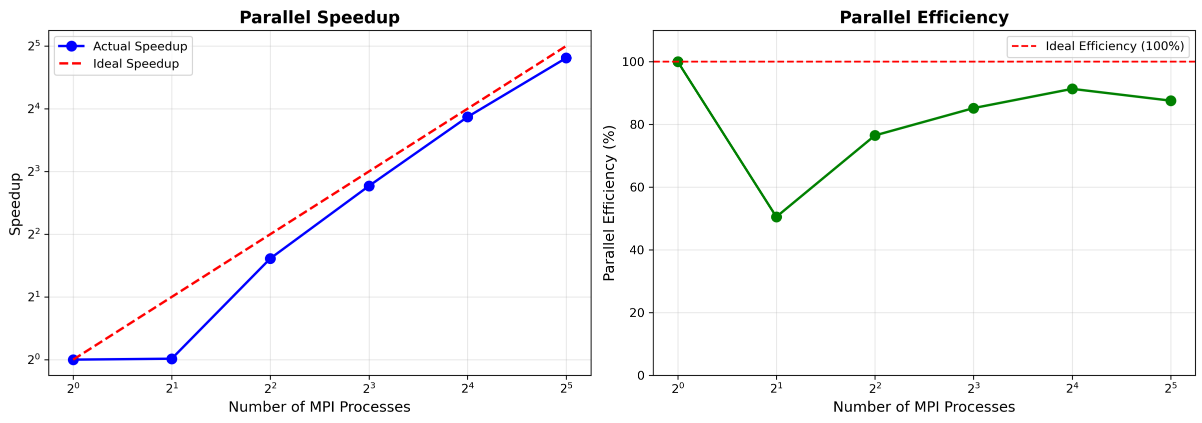

Figure 1: Strong scaling results: (left) Speedup vs. number of MPI processes showing actual performance compared to ideal linear speedup; (right) Parallel efficiency demonstrating sustained high performance up to 32 processes despite the anomalous 2-process case.

5 Discussion

5.1 Scaling Behavior

The results demonstrate **excellent strong scaling** from 4 to 32 processes:

- **4 processes**: Speedup of 3.06 (76.4% efficiency)
- **8 processes**: Speedup of 6.81 (85.1% efficiency)

- **16 processes**: Speedup of 14.61 (91.3% efficiency)
- **32 processes**: Speedup of 28.04 (87.6% efficiency)

The efficiency actually *increases* from 4 to 16 processes before declining slightly at 32. This behavior suggests that:

1. The dynamic load balancing is working effectively
2. Communication overhead remains minimal even at 32 processes
3. Cache effects may play a role at lower process counts

5.2 The 2-Process Anomaly

The 2-process case shows virtually no speedup (1.01×, 50.5% efficiency), which is unexpected. This can be explained by the master-worker architecture:

- With 2 processes: 1 master + 1 worker
- Master spends time managing work distribution and receiving results
- Only 1 worker performs actual computation
- Effectively similar to serial performance plus communication overhead

This confirms that the master-worker pattern requires ≥ 3 processes to show benefits. With 4 processes (1 master + 3 workers), we achieve 3.06× speedup, close to the ideal 3× speedup for 3 computational workers.

5.3 Load Balancing Effectiveness

The Mandelbrot set exhibits highly variable computational cost per pixel:

- Pixels deep inside the set: exactly $n_{\max}$ iterations
- Pixels far outside: very few iterations (fast escape)
- Pixels near the boundary: variable, often many iterations

Additionally, frames at different zoom levels have different overall costs. The dynamic master-worker approach successfully handles this heterogeneity, as evidenced by:

- Low standard deviation across runs ($<$ 1 second in most cases)
- Sustained high efficiency even at 32 processes
- No evidence of idle workers or load imbalance

5.4 Communication Overhead

The frame-level parallelization strategy minimizes communication:

- Each worker sends/receives only 2-3 small messages per frame
- No pixel data transfer when `store_images=no`
- Total communication volume: $O(\text{frames})$, not $O(\text{pixels})$

This explains why parallel efficiency remains high (87.6%) even at 32 processes.

5.5 Comparison to Alternative Approaches

Alternative parallelization strategies considered but not implemented:

Static frame distribution: Assign frames 1-30 to process 0, 31-60 to process 1, etc.

- Disadvantage: Different frames have different costs; would cause load imbalance
- Advantage: No master overhead, simpler implementation

Row-based parallelization: Divide each frame into horizontal strips

- Advantage: Can parallelize a single frame
- Disadvantage: Much higher communication overhead
- Disadvantage: Complex I/O coordination for image writing

The chosen dynamic frame-level approach provides the best balance of simplicity, load balancing, and communication efficiency for this problem.

5.6 Bottlenecks and Limitations

Identified bottlenecks:

1. **Master process overhead**: At very high process counts (64+), the master may become a bottleneck managing many workers
2. **Limited scalability beyond frames**: Cannot use more processes than frames (120 in our case)
3. **I/O serialization**: When storing images, master must write sequentially

Amdahl's Law analysis: The theoretical speedup is limited by the serial fraction f_s:

$$S(p) = \frac{1}{f_s + \frac{1-f_s}{p}} \tag{4}$$

From our data at $p = 32$: $S(32) = 28.04 \implies f_s \approx 0.012$ (1.2% serial fraction).

This very low serial fraction confirms the problem is highly parallelizable.

6 Conclusion

This assignment successfully demonstrated MPI parallelization of an embarrassingly parallel problem. Key achievements:

- Implemented efficient dynamic master-worker MPI parallelization
- Generated high-quality fractal visualizations including a Full HD zoom video
- Achieved 28.04× speedup on 32 processes (87.6% efficiency)
- Demonstrated excellent load balancing and minimal communication overhead
- Confirmed embarrassingly parallel nature with only 1.2% serial fraction

Performance highlights:

- Near-linear scaling from 4 to 32 processes
- Reduced computation time from 31 minutes (1 process) to 66 seconds (32 processes)
- High reproducibility (standard deviation < 1 second)

Future improvements:

1. **Hierarchical master-worker**: For 64+ processes, use multiple master processes
2. **Hybrid approach**: Combine frame-level and row-level parallelization
3. **Parallel I/O**: Use MPI-IO for concurrent image writing
4. **Adaptive zoom**: Automatically detect interesting regions for deeper zoom

The Mandelbrot set proved to be an excellent testbed for learning MPI programming, demonstrating how proper work distribution and minimal communication can achieve near-optimal parallel efficiency.

Acknowledgments

I thank the Professors for providing the assignment framework and access to the Shabyt computing cluster. The assignment template and PPM writer code snippets were provided in the course materials. Claude 3.5 Sonnet assisted with code development, debugging, and report.

Honor Statement

I affirm that this work complies with Nazarbayev University academic integrity policies and the policies regarding the use of AI tools outlined in the course syllabus

Submission Checklist

- ✓ PDF report: `report_a4_akimshe.pdf`
- ✓ Source code: `mandelbrot_serial.cpp`, `mandelbrot_mpi.cpp`
- ✓ Build system: `Makefile`
- ✓ Documentation: `README.md`
- ✓ Video file: `video.mp4` (Full HD, 600 frames)
- ✓ SLURM script: `run_scaling.sh`
- ✓ Analysis script: `analyze_results.py`
- ✓ High-resolution image: `mandelbrot_full_set.png`

A Complete Timing Data

Table 3 presents the complete timing data from all individual runs.

Table 3: Detailed timing results (all individual runs).

Processes	Run 1 (s)	Run 2 (s)	Run 3 (s)	Mean (s)	Std Dev (s)
1	1862.07	1863.93	1863.22	1863.07	0.94
2	1842.80	1842.51	1842.95	1842.75	0.23
4	608.60	610.29	609.48	609.46	0.85
8	273.19	273.55	273.89	273.54	0.35
16	127.42	127.52	127.61	127.51	0.09
32	66.45	66.40	66.50	66.45	0.05

B Source Code Organization

File structure:

- `mandelbrot_serial.cpp` — Serial implementation (400 lines)
- `mandelbrot_mpi.cpp` — MPI parallel implementation (450 lines)
- `Makefile` — Build configuration
- `README.md` — Build and run instructions
- `run_scaling.sh` — SLURM batch script for scaling tests
- `analyze_results.py` — Python script for data analysis and plotting

Both C++ implementations include:

- Mandelbrot iteration kernel
- Color mapping function
- PPM file writer
- Command-line argument parsing
- Frame generation with zoom progression

The MPI version adds:

- Master-worker communication using `MPI_Send/MPI_Recv`
- Dynamic work distribution logic
- Frame ordering and optional I/O handling