b UNIVERSITÄT BERN

# FINAL PROJECT REPORT

## MANDELBROT CONCURRENCY


KIAN SAHAFI

JMCS 2025
FACULTY OF COMPUTER SCIENCE
UNIVERSITY OF BERN


PROFESSOR:

PASCAL FELBER

MAY 19

# 1 PROGRAM DEVELOPMENT AND PARALLELIZATION

## 1.1 CORE LOGIC

The computation of the Mandelbrot set involves iterating the quadratic map for each pixel in the output image to determine if the corresponding complex number c belongs to the set or how quickly its orbit diverges.

## 1.2 JAVA IMPLEMENTATION

The application is structured around several key Java classes:

1. **main.java**: The main entry point of the program. It parses command-line arguments to configure parameters such as the number of iterations, scheduling policy, number of threads, chunk size, and the specific chunking method.

2. **Chunking.java**: This class encapsulates the core logic for parallel Mandelbrot set computation. It handles the division of the image into manageable chunks and their concurrent processing.

3. **chunk.java**: Represents a "chunk" of the image, which is essentially a collection of ‘pixel‘ objects that are processed together.

4. **pixel.java**: Defines a pixel with its x and y coordinates in the image and stores its calculated color information.

## 1.3 CONCURRENCY MECHANISMS USED

To achieve parallelism, the application utilizes Java’s built-in concurrency utilities. Specifically, **Chunking.java** employs a **java.util.concurrent.ExecutorService** created using **Executors.newFixedThreadPool(num-threads)**. This creates a thread pool with a user-configurable number of threads. Each chunk of the Mandelbrot set image is processed as a separate task (a **Runnable** that calls the **processChunk** method) and submitted to this **ExecutorService** for concurrent execution. *Future* $<? >$ objects are used to track the completion of these tasks.

## 1.4 WORK DIVISION (CHUNKING)

The image to be generated (1280x720 pixels) is divided into smaller portions called chunks to distribute the workload among the available threads.

1. **Chunking Methods**: Two methods for dividing the work are implemented:
   * **by Row**: The image is divided into chunks of rows.
   * **by Column**: The image is divided into chunks of columns.

2. **Chunk Size**: The **chunkSize** parameter, specified via the command line, determines the number of rows or columns in each chunk.

## 1.5 SCHEDULING POLICIES

The application was benchmarked with four different "scheduling policies" passed as arguments: "Static-block", "Static-cyclic", "Dynamic", and "Guided". In the context of the provided **Chunking.java** code, these policies primarily influence how the initial **chunk-array** is created or ordered before tasks are submitted to the **ExecutorService**. The **ExecutorService** with a **FixedThreadPool** itself uses a shared unbounded queue for tasks and worker threads pick up tasks from this queue.

2

# HOW TO RUN THE PROGRAM

To execute the program:

1. **Compile the Java files**: first compile all Java files using **javac *.java**.
2. **Run the main program:** run the main java file using **java main.java <iterations> <scheduling-policy> <num-threads> <chunk-size> <chunk-method>** (you can just run it as is with the default values)

Or, To execute the program for all combinations of parameters, you must:

1. **edit the route**: go to the **run-benchmarks.sh** file and change the **PROJECT-DIR** variable on top of the code to the location of the project.
2. **Make the script executable**: If not already, **run chmod +x run-benchmarks.sh**.
3. **Run the benchmarks**: Execute **./run-benchmarks.sh**.

The **run-benchmarks.sh** script automates the process of running the Mandelbrot generator (main.java) with various combinations of parameters:

1. **Iterations**: 1000, 10000
2. **Scheduling Policies**: Static-block, Static-cyclic, Dynamic, Guided
3. **Threads**: 1, 2, 4, 8
4. **Chunk Sizes**: 1, 50, 100, 720
5. **Chunking Methods**: "by Row", "by Column"

**Output**:

1. **Images**: The generated images are saved in an **images/** subdirectory.
2. **Benchmark Results**: Performance data (time taken for each configuration) are logged in **Mandelbrot-benchmark-results.csv**.
3. **Log File**: Detailed console output for each test run is saved in **mandelbrot-benchmark.log**.

# 3

# RESULTS AND PERFORMANCE ANALYSIS

The performance of the parallel Mandelbrot generator was analyzed using the data collected by **run-benchmarks.sh** and processed by **performance-analysis.ipynb**. The analysis focuses on speedup, thread scaling, and the impact of different scheduling policies, chunk sizes, and chunking methods.

## 3.1 SPEEDUP

The speedup is calculated as TimeTaken(1 Thread) / TimeTaken(N Threads). Based on the benchmark results from **mandelbrot-benchmark-results.csv**:

1. Example for 1000 Iterations (Static-block, by Row):
   a) 1 Thread (Chunk Size 100): 0.730 seconds
   b) 4 Threads (Chunk Size 100): 0.440 seconds
   c) Speedup = 0.730 / 0.440 ≈ **1.66x**
2. Example for 10000 Iterations(Dynamic, by Row, Chunk Size 50 - from backup.zip results):
   a) 1 Thread: 5.606 seconds
   b) 4 Threads: 1.719 seconds
   c) Speedup = 5.606 / 1.719 ≈ **3.26x**

The speedup varies with parameters. The configuration with 10000 iterations, Dynamic scheduling, "by Row" chunking, and a chunk size of 50 achieved a speedup of approximately **3.26x** on 4 threads compared to 1 thread.

## 3.2 THREAD SCALING

The Python notebook ‘performance-analysis.ipynb‘ generates plots of "Average Execution Time vs. Threads". These plots demonstrate how execution time decreases as the number of threads increases from 1 to 8. Generally, the execution time reduces with more threads, but the speedup is not perfectly linear due to overheads associated with thread management and workload distribution.

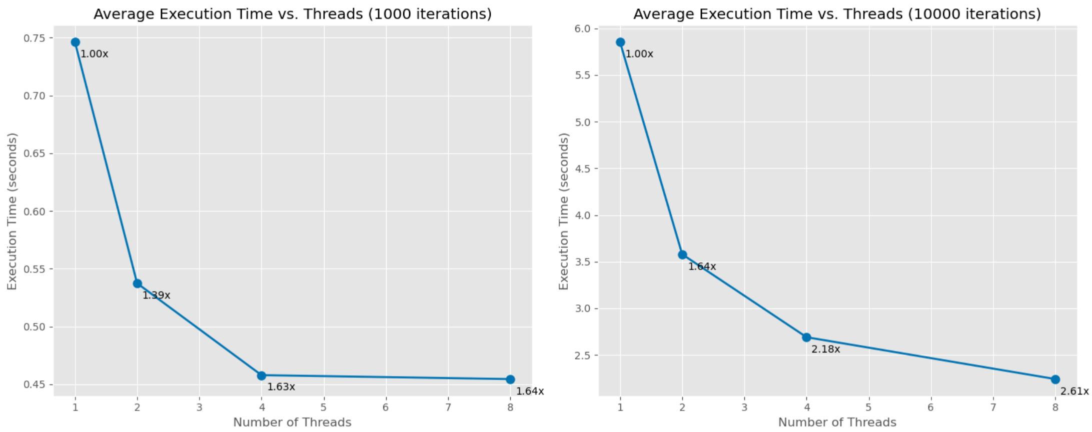


Figure 1: Thread Scaling

## 3.3 SCHEDULING POLICIES

The python notebook allow for comparison of the "Static-block", "Static-cyclic", "Dynamic", and "Guided" scheduling approaches. The plot compares their performance.

1. **For 1000 iterations (by Row, Chunk size 50):**
   a) Static-block (4 threads): 0.441s
   b) Static-cyclic (4 threads): 0.379s
   c) Dynamic (4 threads): 0.356s
   d) Guided (4 threads): 0.387s
2. **For 10000 iterations (by Row, Chunk size 50):**
   a) Static-block (4 threads): 2.710s
   b) Static-cyclic (4 threads): 1.681s
   c) Dynamic (4 threads): 1.719s
   d) Guided (4 threads): 1.704s

From this limited sample, "Dynamic" and "Static-cyclic" policies tend to perform well, especially with 4 threads. The optimal policy can vary depending on the specific iteration count, and chunk parameters.

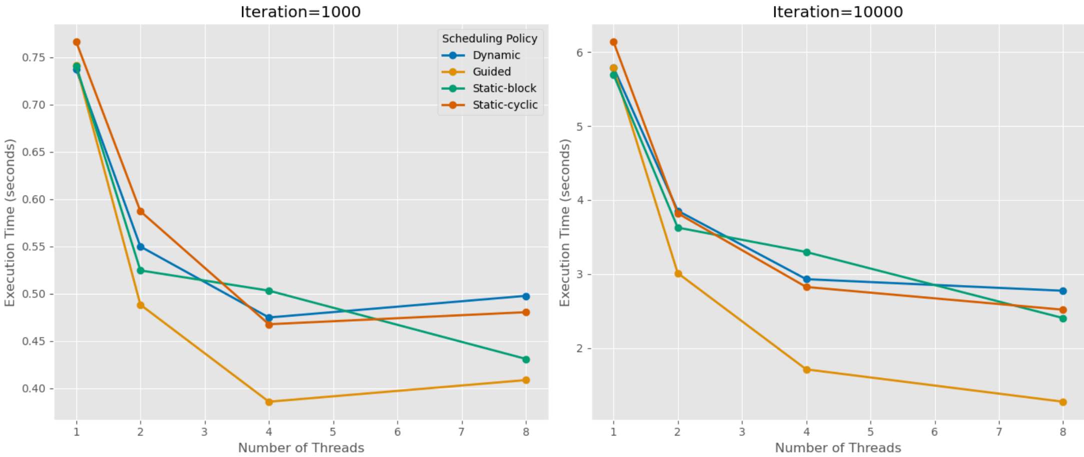


Figure 2: Scheduling Policies

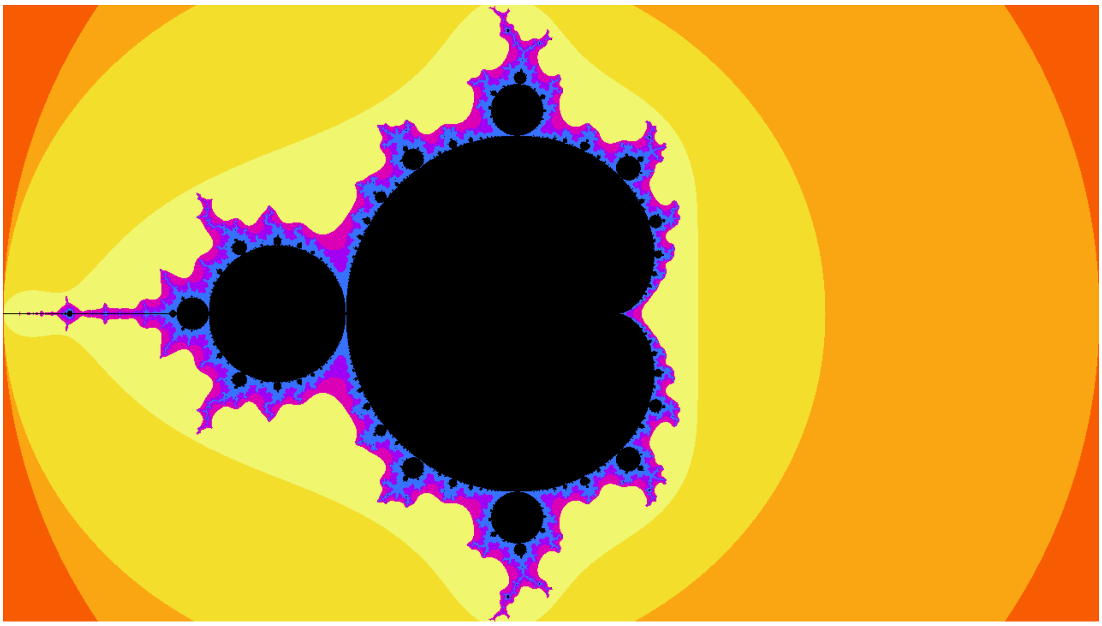

Figure 3: Final Result, 1000 Iterations, Static-cyclic, by Row, Chunk Size 1, by Column, Two threads

# 4 CONCLUSION

This project implemented a parallel Mandelbrot set generator in Java using **ExecutorService**. The benchmarking results demonstrate significant speedup with multiple threads, with configurations like 10000 iterations using the "Dynamic" scheduling policy achieving over 3x speedup on 4 cores.

## 4.1 CITATION


1. java.rubikscube.info
2. jwar663/Java-Mandelbrot