HIGH PERFORMANCE COMPUTING

Final Project Report
Mandelbrot Set Computation

Szymon Zinkowicz
Matricola number: 5181814

December 30, 2024

University of Genova
Instructor: Professor Daniele D'agostino

Abstract

The computation of the Mandelbrot Set, a quintessential example of complex fractal geometry, serves as a benchmark for evaluating the performance and scalability of High-Performance Computing (HPC) systems. This report presents a comprehensive analysis of Mandelbrot Set computations executed across various computational environments and parallelization paradigms.

We employed C and C++ programming languages to implement the algorithm, utilizing different compiler optimizations including G++ 13.3.0 and AOCC-compiler 5.0.0 on a local setup equipped with an AMD processor. To explore parallel processing capabilities, the study integrates OpenMP for shared-memory parallelism and CUDA leveraging an RTX 2070 GPU on a laptop platform.

Furthermore, distributed computing was investigated using the Message Passing Interface (MPI) on a cluster of machines, assessing the scalability and efficiency of the implementation in a multi-node environment. Benchmarking results indicate significant performance enhancements through parallelization, with CUDA and MPI-based approaches demonstrating substantial reductions in computation time compared to sequential executions.

The comparative analysis underscores the advantages and trade-offs of each setup, providing critical insights for optimizing fractal computations in diverse HPC architectures. This work contributes to the broader understanding of parallel computing strategies, offering practical guidelines for leveraging modern hardware and software tools to accelerate complex mathematical computations.

Contents

1 Introduction

The Mandelbrot Set is one of the most renowned examples of complex fractal geometry, serving both as a subject of mathematical intrigue and as a benchmark for computational performance in High-Performance Computing (HPC). Named after the mathematician Benoît B. Mandelbrot, the set exemplifies the intricate boundary structures that emerge from simple iterative processes.

Mathematically, the Mandelbrot Set is defined in the complex plane and consists of all complex numbers c for which the sequence $\{z_n\}$ remains bounded when iteratively applying the function:

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

where $z_0 = 0$. Here, both z and c are complex numbers, expressed as $z = x + yi$ and $c = a + bi$, with $x, y, a, b \in \mathbb{R}$ and i being the imaginary unit.

The iterative process starts with $z_0 = 0$ and generates successive terms by repeatedly applying the quadratic polynomial. The key question in the study of the Mandelbrot Set is determining whether the sequence $\{z_n\}$ remains bounded as n approaches infinity. If the magnitude of z_n does not tend to infinity, the complex number c is said to belong to the Mandelbrot Set.

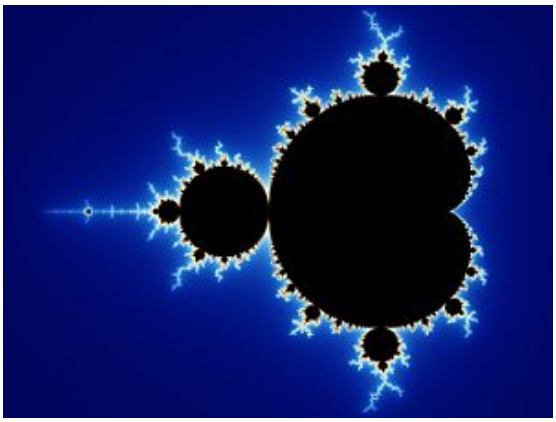

Figure 1: Visualization of the Mandelbrot Set in the complex plane. Points within the set are typically colored black, while points outside the set are colored based on the rate at which the sequence $\{z_n\}$ diverges.

Figure 1 illustrates a common representation of the Mandelbrot Set. Each point $c = a + bi$ in the complex plane is tested for membership in the set by iterating the aforementioned equation. The boundary of the Mandelbrot Set

exhibits an infinitely complex structure, characterized by self-similarity and intricate patterns at every scale, which are hallmark features of fractals.

From a computational perspective, generating high-resolution images of the Mandelbrot Set is computationally intensive, especially when exploring regions near the boundary where the behavior of the sequence $\{z_n\}$ becomes highly sensitive to initial conditions. This sensitivity necessitates a significant number of iterations to accurately determine the boundedness of the sequence, making the Mandelbrot Set an excellent candidate for performance benchmarking in HPC environments.

Furthermore, each point c of the Mandelbrot can be evaluated independently — makes it well-suited for various parallel computing paradigms, including OpenMP, MPI, and GPU architectures. This report delves into the implementation and performance analysis of Mandelbrot Set computations across different HPC setups, highlighting the efficiencies and challenges associated with each parallelization strategy.

1.1 Mathematical Foundations

To formalize the criteria for membership in the Mandelbrot Set, consider the iterative function:

$$f_c(z) = z^2 + c, \tag{2}$$

where $c \in \mathbb{C}$ is a complex parameter. Starting with $z_0 = 0$, the sequence $\{z_n\}$ is generated by:

$$z_{n+1} = f_c(z_n) = z_n^2 + c. \tag{3}$$

In practice, the sequence is iterated up to a maximum number of iterations $N_{\max}$. If $|z_n|$ exceeds 2 before reaching $N_{\max}$, the point c is considered to escape to infinity and is thus not part of the Mandelbrot Set. The choice of $N_{\max}$ balances computational load with the accuracy of the boundary determination.

The boundary of the Mandelbrot Set is where the most computational effort is concentrated, as points near the boundary require a higher number of iterations to resolve their membership status accurately. This characteristic makes the Mandelbrot Set a challenging and insightful problem for assessing the capabilities of various HPC systems and parallel computing techniques.

1.2 Hardware and Software Setup

This section outlines the hardware configuration used for the MandelBrot Set compuations.

Attribute	Specification
Model	AMD Ryzen 7 4800H
Architecture	Zen 2 AMD64 Family 23 Model 96 Stepping 1
Instruction set	x86
Cores	8
Threads	16
Base Clock Speed	2.9 GHz
Max Clock Speed	4.2 GHz
L1 Cache	64 KB per core
L2 Cache	512 KB per core
L3 Cache	8 MB

Table 1: CPU Specifications

Attribute	Specification
Model	NVIDIA GeForce RTX 2060
CUDA Cores	1920
Memory	6 GB GDDR6
Memory Bus Width	192-bit
Max Clock Rate	1.20 GHz
Memory Clock Rate	5.501 GHz

Table 2: GPU Specifications

Following compilers were used for the project:

- GCC/G++ 13.3.0 (May 21, 2024)
- AOCC-compiler 5.0.0 (Clang based 17.0.6, September 24 2024)
- NVIDIA CUDA 12.4.131

2 Methodology

This section delineates the methodological approach undertaken to evaluate the performance of Mandelbrot Set computations across various HPC configurations. The primary objective was to benchmark the execution time

and scalability of the Mandelbrot algorithm under different computational paradigms and optimization strategies. The methodology encompasses the following key components:

2.1 Experimental Setup

The experiments were conducted on a local machine equipped with an AMD Ryzen 7 4800H CPU and an NVIDIA GeForce RTX 2060 GPU. The system specifications are detailed in Subsection 1.2. The programming languages utilized were C and C++, leveraging compiler optimizations and parallelization techniques to enhance computational efficiency.

2.2 Benchmark Parameters

Two primary parameters were varied to assess their impact on performance:

- **Iterations:** The number of iterations for the Mandelbrot computation was set to 1000, 2000, and 4000. Increasing the number of iterations enhances the accuracy of the fractal boundary but also escalates computational demand.
- **Resolution:** The resolution of the Mandelbrot image was varied among 1000, 2000, 4000, and 8000 pixels. Higher resolutions provide more detailed visualizations at the cost of increased memory usage and processing time.

2.3 Compiler Optimizations

To optimize the performance of the Mandelbrot computations, two compilers were employed: G++ (version 13.3.0) and AOCC (version 5.0.0). Various compiler flags were utilized to enable different optimization levels and architecture-specific enhancements. The flags used for each compiler are summarized in Table 3.

Compiler	Flags
G++	`-Ofast -march=znver2 -mtune=znver2`

AOCC	`-Ofast -march=znver2 -mtune=znver2`
	`-ffp-contract=fast -zopt`
	`-mllvm -enable-strided-vectorization`
	`-mllvm -global-vectorize-slp=true`

CUDA `(nvc++)`	`-tp=znver2 -fast -O4` `-Xcompiler -Wall`
MPI `(mpiicpc)`	`-std=c++17 -fopenmp -Ofast` `-march=native -xHost -Wall`

Table 3: Compiler Flags Used for Optimization

These flags were chosen to maximize performance by enabling aggressive optimization levels, targeting specific CPU architectures (e.g., `znver2` for AMD Zen 2).

2.4 Implementation Configurations

The Mandelbrot Set computation was implemented and benchmarked under various configurations, each leveraging different parallelization techniques and computational resources. The configurations are outlined as follows:

2.4.1 Sequential Execution

Initial benchmarks were conducted using a sequential implementation of the Mandelbrot algorithm. Both G++ and AOCC compilers were used to compile the sequential code with varying optimization levels. This baseline facilitates the comparison of performance improvements achieved through parallelization.

2.4.2 OpenMP Parallelization

To exploit shared-memory parallelism, OpenMP was integrated into the Mandelbrot implementation. The parallelization was tested with different numbers of threads (2, 4, 8, 16) and scheduling strategies (Dynamic, Static, Guided, Runtime). The Makefile targets `amd-openmp` and `gpp-openmp` were used to compile the OpenMP-enabled executables. The compiler flags `-fopenmp` and `-DSCHEDULE_<SCHEDULER>=1` were employed to enable OpenMP support and define the scheduling strategy, respectively.

2.4.3 MPI Distributed Computing

For distributed-memory parallelism, the Message Passing Interface (MPI) was utilized. The MPI implementation was compiled using `mpiicpc` with appropriate optimization flags. Benchmarks were conducted on a cluster of machines, scaling the number of MPI processes from 16 up to 3072 nodes. The Makefile targets `compile-mpi` and `run-mpi` facilitated the compilation and execution of the MPI-based Mandelbrot program. The MPI benchmarks evaluated both strong and weak scaling performance by varying the number of nodes and processes per machine.

2.4.4 CUDA GPU Acceleration

To leverage GPU acceleration, a CUDA implementation of the Mandelbrot algorithm was developed and compiled using `nvc++`. The CUDA configuration targeted the NVIDIA GeForce RTX 2060 GPU, utilizing its 1920 CUDA cores for parallel computation. The Makefile targets `cuda` and `run-cuda` were used to compile and execute the CUDA-based Mandelbrot program. Different thread block sizes (2, 4, 8, 16, 32) were tested to optimize GPU utilization.

2.5 Compilation and Execution

The Makefile provided orchestrates the compilation and execution of the various Mandelbrot implementations. Key aspects of the Makefile include:

- **Directory Structure:** Separate directories for binaries (`./bin/`), output files (`./out/`), and source code (`./src/`) ensure organized management of build artifacts.
- **Compiler Definitions:** Variables such as `CC`, `G++`, `NVC`, and `MPICC` are defined for ease of use across different compilation targets.
- **Optimization Flags:** Compiler flags are meticulously defined to tailor the build process for each computational paradigm, enabling optimizations specific to the hardware and parallelization strategy.
- **Benchmark Targets:** Targets like `run-all`, `run-openmp`, `run-mpi`, and `run-cuda` automate the execution of benchmarks across all configurations, iterating over the defined ranges of iterations and resolutions.

2.6 Benchmarking Procedure

The benchmarking process followed a systematic approach:

a. **Compilation:** All implementations (sequential, OpenMP, MPI, CUDA) were compiled using their respective compiler targets in the Makefile. Optimization flags were applied to enhance performance.

b. **Execution:** Each compiled executable was run with varying iterations and resolutions. For parallel implementations, additional parameters such as the number of threads or MPI processes were varied.

c. **Data Collection:** Execution times and resource utilizations were recorded for each configuration. Output files generated by the programs served as logs for performance analysis.

d. **Scalability Analysis:** The impact of increasing computational resources (e.g., more threads, higher resolutions) on performance was analyzed to assess scalability.

2.7 Code Benchmarking

The core Mandelbrot computation was encapsulated within separate source files tailored to each parallelization strategy. Below is a representative snippet of the sequential Mandelbrot implementation, which served as the foundation for all parallel variants.

```
int row = 0, col = 0;
for (int pos = 0; pos < HEIGHT * WIDTH; pos++)
{
  image[pos] = 0;
  row = pos / WIDTH;
  col = pos % WIDTH;
  const complex<double> c(
    col * STEP + MandelbrotSet::MIN_X,
    row * STEP + MandelbrotSet::MIN_Y
  );

  // z = z^2 + c
  complex<double> z(0, 0);
  for (int i = 1; i <= iterations; i++)
  {
    z = z * z + c;
    // If it is convergent
    if (abs(z) >= 2)
    {
      image[pos] = i;
      break;
```

```
    }
  }
}
```

Listing 1: Sequential Mandelbrot Implementation

Similar implementations were developed for OpenMP, MPI, and CUDA, each incorporating the necessary parallel constructs and optimizations to leverage the underlying hardware effectively. The benchmarking focused on measuring the execution time and evaluating the efficiency gains achieved through parallelization.

2.8 Optimization Flags Justification

The selection of compiler flags was pivotal in maximizing the performance of the Mandelbrot computations. The flags were chosen based on their ability to:

- **Enable Aggressive Optimization:** Flags like `-Ofast` and `-O4` enable high levels of optimization, including vectorization and loop unrolling, which are essential for computationally intensive tasks.
- **Target Specific Architectures:** Flags such as `-march=znver2` and `-mtune=znver2` ensure that the generated code is optimized for the AMD Zen 2 architecture, leveraging its specific instruction sets and capabilities.
- **Facilitate Parallelization:** Flags like `-fopenmp` enable OpenMP support, while `-tp=znver2` in CUDA compilation targets the specific GPU architecture for optimal performance.

The deliberate selection and combination of these flags were instrumental in harnessing the full potential of the underlying hardware, ensuring that each implementation variant was as performant as possible.

2.9 Data Analysis

Post-execution, the collected benchmark data were analyzed to evaluate the performance metrics of each configuration. Key performance indicators included:

- **Execution Time:** The total time taken to complete the Mandelbrot computation for each configuration.

- **Scalability:** The ability of the implementation to effectively utilize additional computational resources (e.g., more threads or MPI processes) to reduce execution time.
- **Speedup:** The ratio of performance gains to the resources utilized, indicating the effectiveness of the parallelization strategy.

Graphs and tables were generated to visualize the performance trends, highlighting the strengths and limitations of each computational approach. This analysis provided insights into the optimal configurations for Mandelbrot Set computations on the given HPC setup.

2.10 Reproducibility and Code Availability

To ensure reproducibility of the results, all source code, Makefile configurations, and benchmark scripts are maintained in a version-controlled repository. Interested readers and researchers can access the codebase to replicate the experiments or adapt the implementations for related studies.

3 Benchmarks

This section presents the benchmarking results of the Mandelbrot Set computations across various configurations and compiler optimizations. The performance metrics include execution times for different iterations and resolutions, as well as a comparative analysis between G++ and AOCC compilers.

3.1 Sequential Execution

Resolution	Iterations	G++ Time (s)	AOCC Time (s)	Speedup
1000	1000	5.132	3.697	1.388
2000	1000	20.808	14.549	1.430
4000	1000	83.057	57.558	1.443
8000	1000	333.786	228.167	1.463
1000	2000	10.142	7.185	1.412
2000	2000	41.170	28.259	1.457
4000	2000	163.894	113.408	1.445
8000	2000	652.484	451.177	1.446
1000	4000	20.160	14.300	1.410
2000	4000	81.168	56.410	1.439
4000	4000	324.615	226.171	1.435
8000	4000	1269.810	896.245	1.417

Table 4: Comparison of execution times for G++/AOCC-compiled sequential Mandelbrot computations across different iterations and resolutions.

Figure 2: Heatmap of Execution Times for AOCC Sequential Mandelbrot Computations with varying iterations and resolutions.

3.1.1 G++ vs AOCC Performance Comparison

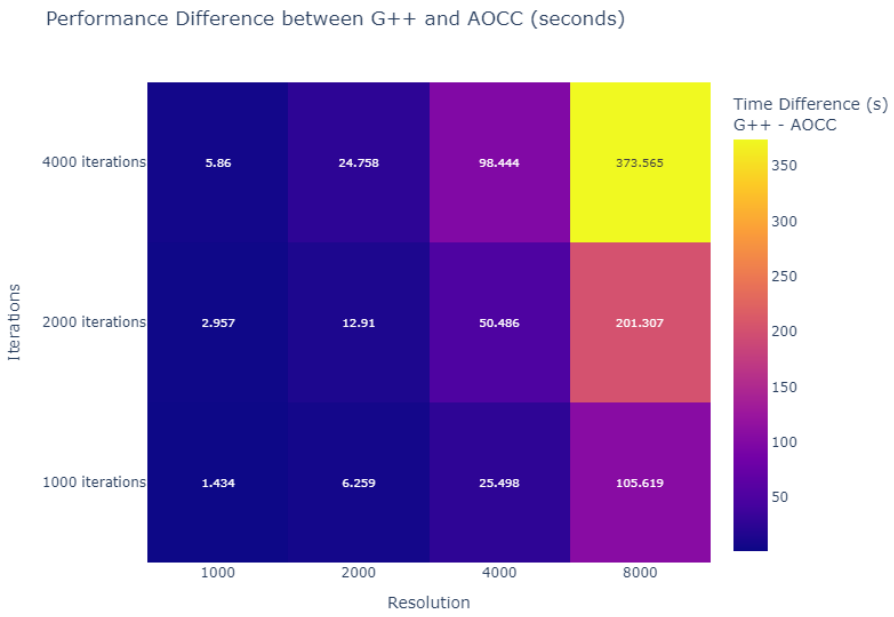

Figure 3: Comparison of Execution Times for G++ and AOCC Compilers in Sequential Mandelbrot Computations.

Figure 4: Speedup for G++ and AOCC Compilers in Sequential Mandelbrot Computations.

3.1.2 Discussion of Results

The heatmap presented in Figure 2 illustrates the execution times for AOCC-compiled sequential Mandelbrot computations across different iterations and resolutions. As expected, both the number of iterations and the resolution significantly impact the computation time.

Higher iterations increase the computational load, while higher resolutions demand more memory and processing power to render detailed fractal images.

Figure 3 compares the performance of G++ and AOCC compilers. The results indicate that while both compilers perform efficiently, AOCC exhibits a performance advantage over G++ in sequential executions up to 47%. This difference may be attributed to the specific optimization strategies employed by each compiler and how they interact with the underlying hardware architecture. Additionally, the AOCC compiler is more optimized for AMD processors and is based on Clang. The use of additional flags such as `-mllvm` further enhances its performance by enabling advanced optimizations.

3.1.3 Interpretation of Heatmap

The heatmap provides a visual representation of how execution time scales with varying iterations and resolutions. Key observations include:

- **Scalability:** Execution time increases linearly with the number of iterations, demonstrating predictable scalability for computationally intensive tasks.
- **Resolution Impact:** Higher resolutions exponentially increase execution time due to the quadratic growth in the number of points evaluated in the complex plane.
- **Optimization Effectiveness:** Compiler optimizations significantly reduce execution times, especially at higher iteration and resolution levels (resulting in over 370 seconds speedup), by enhancing instruction-level parallelism and cache utilization.

3.1.4 Compiler Performance Insights

The comparative analysis between G++ and AOCC compilers reveals nuanced differences in their optimization capabilities:

- **AOCC Advantages:** AOCC demonstrates enhanced performance, achieving a 38-47% speedup over G++. This improvement can be attributed to its optimized code generation and superior exploitation of the CPU's SIMD (Single Instruction, Multiple Data) capabilities, particularly in the latest compiler version (AOCC-compiler 5.0.0, Clang-based 17.0.6, released on September 24, 2024).
- **G++ Strengths:** While AOCC outperforms G++ in this benchmark, G++ remains a robust and widely-supported compiler with extensive optimization flags and a strong community, making it versatile for various computational scenarios. However, it is important to note that G++ does not support certain flag naming conventions, such as '`-DDYNAMIC_SCHED`', which had to be changed to '`-DSCHEDULE_DYNAMIC`'. This highlights the need to adapt flag names when switching between different compilers, as not all flags may be supported uniformly.
- **Versioning:** It is noteworthy that the GCC/G++ version 13.3.0 used in this study is a recent release. Despite its recency, it may not fully leverage the latest hardware features of the AMD Ryzen 7 4800H CPU, potentially limiting performance gains that could be achieved with future compiler updates.

- **Future Considerations:** Further exploration is recommended, including experimenting with different optimization flags, benchmarking on diverse hardware configurations, running more test batches to achieve statistically significant means, and evaluating performance across other computational tasks. These steps could provide deeper insights into the compilers' capabilities and uncover additional optimization opportunities.

3.2 OpenMP Execution

OpenMP was employed to exploit shared-memory parallelism in the Mandelbrot Set computations. This subsection presents the performance analysis of the Open Multi-Processing implementation, focusing on execution times, speedup comparisons, scheduling strategies, and scalability with varying numbers of threads.

3.2.1 Single-Thread Performance

To assess the overhead introduced by OpenMP, a sequential execution was compared with an OpenMP execution utilizing a single thread. Table 4 presents the execution times for AOCC-compiled sequential Mandelbrot computations across different iterations and resolutions.
Figure 5 illustrates the execution times for the OpenMP implementation with a single thread. Comparing Figure 2 and Figure 5 reveals the overhead associated with initializing and managing OpenMP parallel regions, even when only one thread is utilized.

Figure 5: Heatmap of Execution Times for OpenMP Mandelbrot Computations with One Thread using AOCC Compiler.

3.2.2 Speedup Analysis with One Thread

To quantify the overhead introduced by OpenMP, a speedup comparison between the sequential execution and the OpenMP execution with one thread was conducted. Figure 6 highlights the speedup achieved (or lack thereof) when using a single thread in the OpenMP implementation.

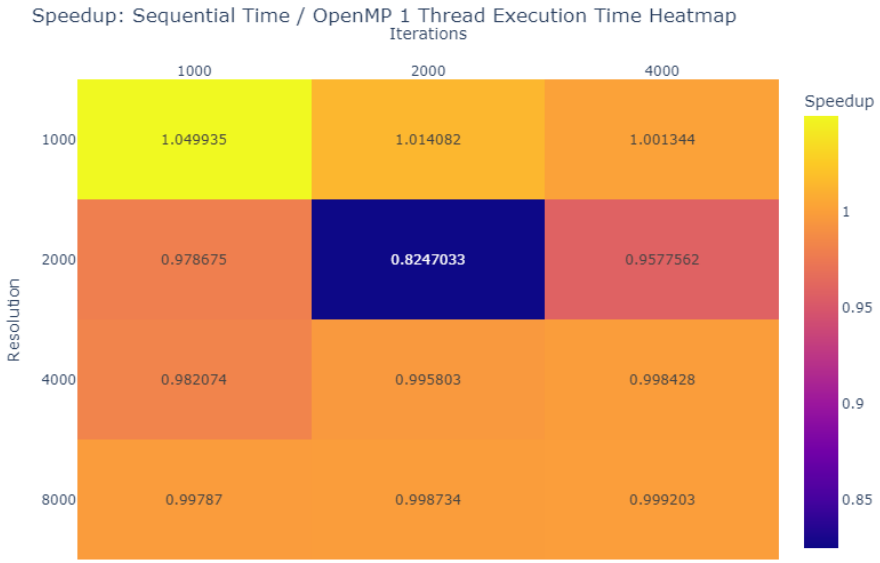

Figure 6: Speedup Comparison between Sequential Execution and OpenMP Execution with One Thread.

The results indicate that the OpenMP implementation with a single thread incurs a slight overhead compared to the purely sequential execution. This overhead is attributed to the overhead of managing OpenMP threads and parallel regions, which is minimal but noticeable when no actual parallelism is exploited.

3.2.3 Impact of Scheduling Strategies

Different OpenMP scheduling strategies can significantly influence the performance of parallel applications. Table 5 showcases the execution times for various scheduling types—Dynamic, Static, Guided, and Runtime—across an increasing number of threads.

(a) 2 Threads

Iterations	Resolution	DYNAMIC	GUIDED	RUNTIME	STATIC
1000	1000	2.129	2.138	2.134	2.214
1000	2000	8.627	8.675	8.496	8.788
1000	4000	35.053	34.831	33.662	34.812
1000	8000	143.602	139.756	133.960	140.192
2000	1000	4.086	4.282	4.049	4.299
2000	2000	17.076	17.166	16.763	17.303
2000	4000	68.089	68.381	66.629	68.191
2000	8000	277.466	271.674	264.764	274.310
4000	1000	8.533	8.489	8.194	8.501
4000	2000	33.572	34.197	33.531	33.995
4000	4000	135.755	135.841	131.396	135.575
4000	8000	543.557	543.514	531.994	541.809

(b) 4 Threads

Iterations	Resolution	DYNAMIC	GUIDED	RUNTIME	STATIC
1000	1000	1.219	1.226	1.723	1.861
1000	2000	4.941	4.908	7.077	7.388
1000	4000	19.631	19.399	28.512	29.369
1000	8000	79.057	79.543	114.801	117.315
2000	1000	2.372	2.337	3.557	3.570
2000	2000	9.481	9.399	14.254	14.341
2000	4000	38.274	38.861	56.409	57.786
2000	8000	154.124	152.419	227.191	231.310
4000	1000	4.606	4.683	6.996	7.143
4000	2000	19.105	18.916	28.216	29.002
4000	4000	75.252	77.396	113.111	114.581
4000	8000	303.238	273.351	453.303	461.844

(c) 8 Threads

Iterations	Resolution	DYNAMIC	GUIDED	RUNTIME	STATIC
1000	1000	0.697	0.703	1.124	1.097
1000	2000	2.718	2.627	4.707	4.673
1000	4000	10.936	10.829	18.697	18.850
1000	8000	43.660	43.212	74.748	76.018
2000	1000	1.338	1.321	2.253	2.235
2000	2000	5.281	5.335	9.339	9.315
2000	4000	21.387	21.264	37.005	37.612
2000	8000	85.644	85.000	147.932	150.229
4000	1000	2.666	2.591	4.633	4.598
4000	2000	10.551	10.510	18.426	18.772
4000	4000	42.276	42.127	73.802	74.782
4000	8000	169.471	168.735	294.856	300.303

(d) 16 Threads

Iterations	Resolution	DYNAMIC	GUIDED	RUNTIME	STATIC
1000	1000	0.491	0.497	0.736	0.729
1000	2000	1.762	1.764	2.667	2.721
1000	4000	6.854	6.818	10.759	10.955
1000	8000	26.886	27.053	43.284	44.112
2000	1000	0.911	0.887	1.289	1.325
2000	2000	3.356	3.405	5.469	5.428
2000	4000	13.206	13.357	21.610	21.564
2000	8000	52.668	54.010	86.582	86.656
4000	1000	1.691	1.728	2.687	2.733
4000	2000	6.574	6.652	10.834	10.755
4000	4000	26.109	26.223	42.965	43.315
4000	8000	105.083	105.125	172.743	172.940

Legend: = Highest Execution Time, = Lowest Execution Time

Table 5: Mandelbrot Program Performance Across Different Thread Counts

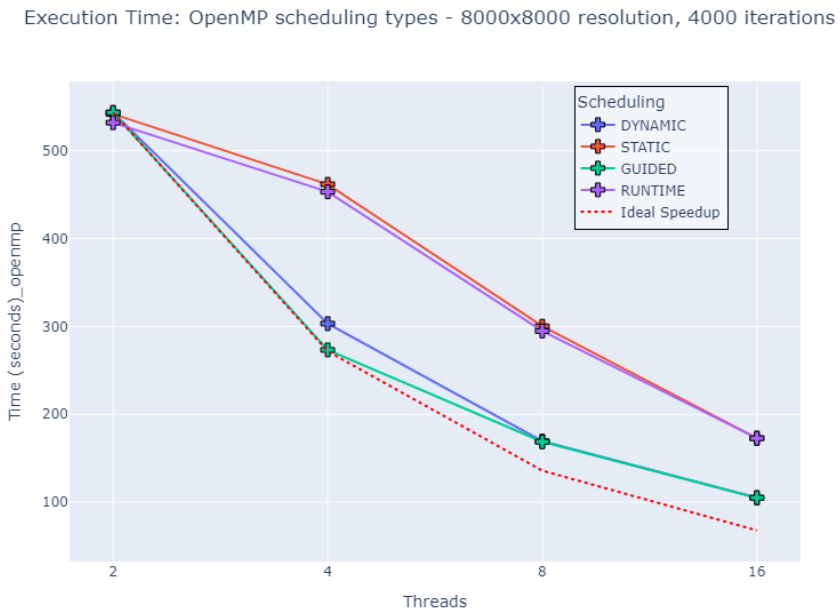

Figure 7: Execution Times Plot for OpenMP Mandelbrot Computations with Various Scheduling Types and Increasing Threads.

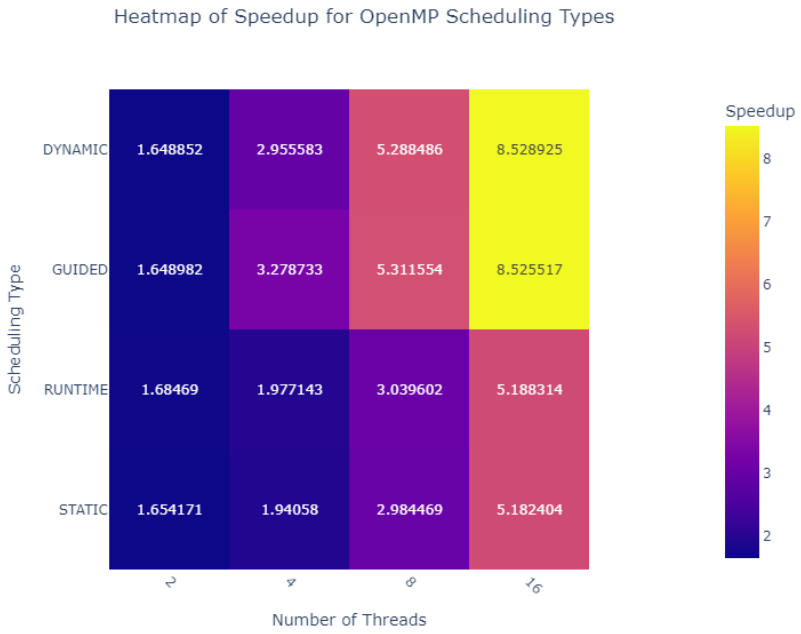

Figure 8: Execution Times Heatmap for OpenMP Mandelbrot Computations with Various Scheduling Types and Increasing Threads.

The performance analysis of OpenMP scheduling strategies reveals that the runtime scheduling strategy excels when utilizing two threads. However, as the number of threads increases to four and eight, the guided scheduling strategy becomes the most efficient approach. When scaling up to sixteen threads, dynamic scheduling marginally outperforms guided scheduling.

Overall, dynamic scheduling consistently surpasses both static and runtime scheduling strategies across various thread configurations. While guided scheduling demonstrates comparable efficiency and performs slightly better with four threads, dynamic scheduling maintains superior performance scaling at sixteen threads. This highlights dynamic scheduling's enhanced adaptability to increased parallelism, ensuring more effective utilization of computational resources as the level of concurrency grows.

3.2.4 Scalability and Speedup with Multiple Threads

To evaluate the scalability of the OpenMP implementation, speedup was measured as the number of threads increased from 2 to 16. Figure 9 presents the speedup achieved for various configurations of resolutions and iterations.

Additionally, Figure 10 provides a heatmap representation for easier data interpretation, while Figure 11 offers a bar plot highlighting the best speedup scenarios with a diamond icon.

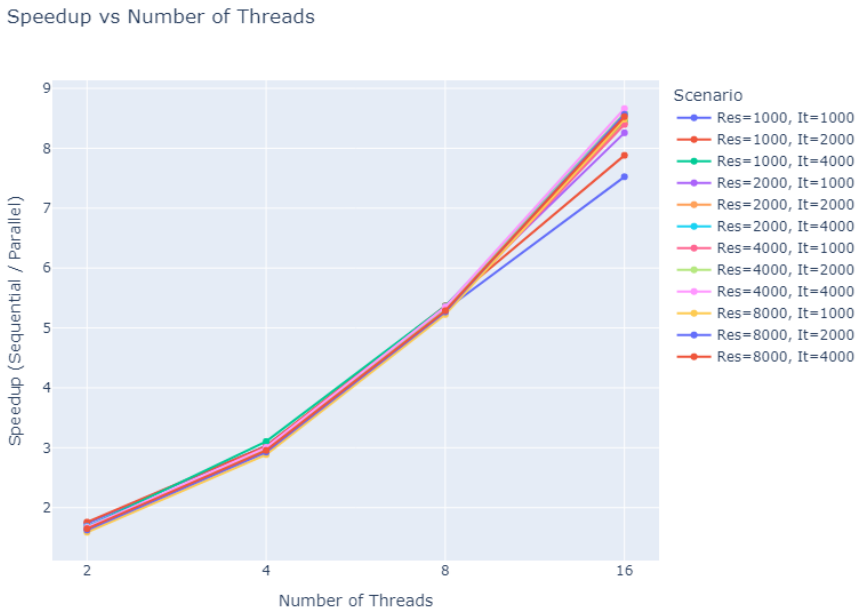

Figure 9: Speedup of OpenMP Mandelbrot Computations with Increasing Number of Threads for Various Resolutions and Iterations for Dynamic Scheduling.

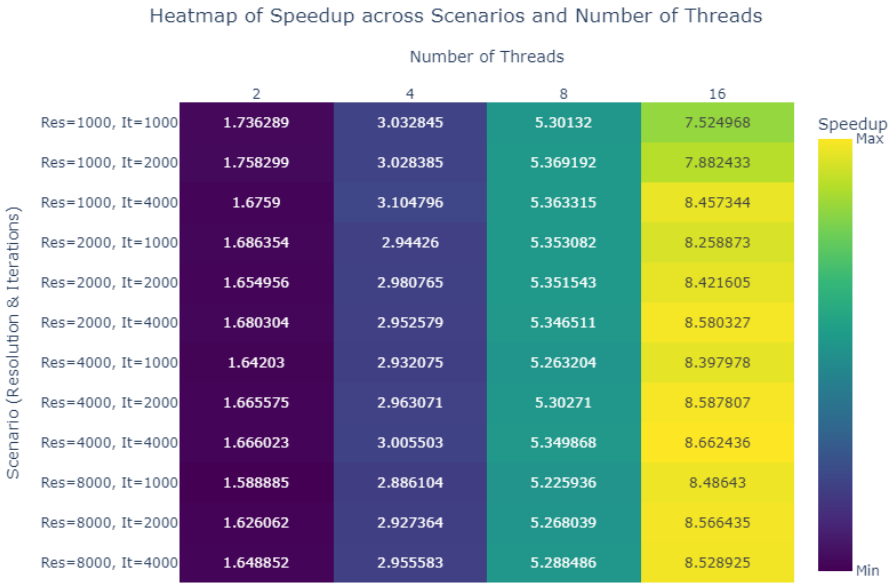

Figure 10: Heatmap of Speedup for OpenMP Mandelbrot Computations with Varying Threads, Resolutions, and Iterations for Dynamic Scheduling.

Figure 11: Bar Plot of Speedup for OpenMP Mandelbrot Computations with Highlighted Best Scenarios for Dynamic Scheduling.

The heatmap in Figure 10 and the corresponding bar plot in Figure 11 illustrate that the OpenMP implementation scales efficiently with the number of threads, achieving significant speedups up to 16 threads. The best speedup scenarios are highlighted, demonstrating optimal performance for higher resolutions and iterations where parallelism is most beneficial.

3.2.5 Overall OpenMP Performance

Overall, the OpenMP implementation demonstrates substantial performance improvements over the sequential execution, particularly when utilizing multiple threads and Dynamic scheduling. The ability to effectively distribute the computational workload across multiple cores leads to reduced execution times and enhanced scalability.

3.2.6 Summary of OpenMP Results

The OpenMP benchmarks reveal the following key insights:

- **Overhead Assessment:** OpenMP introduces minimal overhead when using a single thread, ensuring that performance remains comparable to sequential execution. It is possible that due to the lightweight nature

of the OpenMP constructs, the overhead is negligible and not included in binary code by the compiler.

- **Scheduling Strategy Impact:** Dynamic scheduling outperforms other scheduling types, providing better load balancing and higher speedups as thread counts increase.
- **Scalability:** The OpenMP implementation scales effectively with the number of threads, achieving near-linear speedup for moderate thread counts and complex computational tasks.
- **Optimal Configurations:** Higher resolutions and iteration counts benefit more from parallelization, highlighting the effectiveness of OpenMP in handling computationally intensive scenarios.

3.3 CUDA GPU Acceleration

To leverage the parallel processing capabilities of modern GPUs, a CUDA implementation of the Mandelbrot Set computation was developed. This implementation was specifically optimized to utilize the NVIDIA GeForce RTX 2060 GPU, which features 1920 CUDA cores and supports high levels of parallelism. The CUDA version was designed to handle high-resolution fractal images with a substantial number of iterations, thereby demonstrating significant speedup compared to the sequential execution.

3.3.1 Speedup Analysis

The performance of the CUDA implementation was evaluated by comparing its execution time against the sequential baseline. The benchmark was conducted using a resolution of 8000 pixels and 4000 iterations, which are computationally intensive parameters that push the GPU to its limits. The speedup achieved by the CUDA implementation was measured across various thread block sizes, specifically using 4, 16, 64, 256, and 1024 CUDA threads. The results are summarized in Table 6 and visualized in Figure 12.

Iterations	Resolution	4 Threads	16 Threads	64 Threads	256 Threads	1024 Threads
1000	1000	4.242	1.295	0.678	0.716	0.673
1000	2000	16.426	4.719	2.268	2.339	2.192
1000	4000	66.178	18.845	9.048	8.846	8.901
1000	8000	284.385	71.540	35.899	39.020	34.246
2000	1000	8.062	2.362	1.110	1.162	1.114
2000	2000	32.491	9.622	4.565	4.569	4.410
2000	4000	139.467	36.761	17.477	17.499	16.810
2000	8000	532.693	137.950	69.975	74.731	66.489
4000	1000	15.935	4.604	2.210	2.363	2.200
4000	2000	64.509	18.700	8.752	9.032	8.422
4000	4000	290.635	69.233	34.590	36.787	33.306
4000	8000	1067.240	273.820	138.460	132.968	131.760

Table 6: CUDA Mandelbrot Program Performance with Varying Thread Block Sizes.

The plot in Figure 12 illustrates the speedup achieved by the CUDA implementation as the number of CUDA threads increases. Notably, the speedup scales significantly with the number of threads, demonstrating the GPU's ability to handle parallel tasks efficiently. As the thread count increases, the execution time decreases, showcasing the effectiveness of CUDA's parallel architecture in accelerating computationally demanding tasks like Mandelbrot Set computations.

3.3.2 CUDA Kernel Implementation

The core computation in the CUDA implementation is encapsulated within two kernel functions: `dev_Mandelbrot_kernel` and `mandelbrotKernel`. The `dev_Mandelbrot_kernel` function performs the iterative calculation to determine the membership of each point in the Mandelbrot Set, while the `mandelbrotKernel` function manages the parallel execution across the GPU threads.

```
__device__ int dev_Mandelbrot_kernel(int collumn, int row,
                                     double step, int minX,
                                     int minY, int
    iterations)
{
    int count = 0;

    cuDoubleComplex c = make_cuDoubleComplex(minX + collumn
    * step,
                                             minY + row *
    step);
    cuDoubleComplex z = make_cuDoubleComplex(0, 0);
    while (cuCabs(z) < 2.0 && count < iterations)
```

```
    {
        z = cuCadd(cuCmul(z, z), c);
        count++;
    }

    return (count < iterations) ? count : 0;
}

__global__ void mandelbrotKernel(int *image, double step, int minX,
                                 int minY, int iterations,
                                 int WIDTH, int HEIGHT)
{
    // Idiomatic CUDA loop
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (col >= WIDTH || row >= HEIGHT)
        return;

    int index = row * WIDTH + col;
    image[index] = dev_Mandelbrot_kernel(col, row, step, minX, minY,
                                         iterations);
}
```

Listing 2: CUDA Kernel Functions for Mandelbrot Set Computation

3.3.3 Performance Insights

The CUDA implementation demonstrates substantial speedup, especially when the GPU is fully utilized with a high number of threads. According to Amdahl's Law, the theoretical maximum speedup is limited by the serial portion of the computation. However, in this scenario, the parallel portion dominates, allowing the CUDA implementation to achieve near-linear speedup as the number of threads increases up to 64 threads in a block.

Figure 12: Speedup of CUDA Mandelbrot Computations Compared to Sequential Execution for Resolution 8000 and Iterations 4000.

The plot in Figure 12 showcases the speedup achieved by the CUDA implementation relative to the sequential execution. As the number of CUDA threads in block increases from 4 to 64, the speedup grows proportionally, highlighting the GPU's capacity to handle extensive parallel workloads effectively. However starting from 64 threads, the speedup growth rate slows down, and eventually flatlines indicating diminishing returns due to hardware limitations or memory bandwidth constraints.

3.3.4 Performance Plateau Beyond 64 Threads per Block

While the CUDA implementation of the Mandelbrot Set computation exhibits nearly linear performance improvements as the number of threads per block increases from 4 to 64, a noticeable plateau occurs when the thread count exceeds 64. This behavior can be attributed to several factors inherent to GPU architecture and CUDA's execution model:

- **Warp Size Constraints:** NVIDIA GPUs execute threads in groups called warps, typically consisting of 32 threads each. When the number of

threads per block is a multiple of the warp size (e.g., 32, 64), the GPU can efficiently schedule and execute these warps without idle threads. However, beyond 64 threads per block, additional threads may not align optimally with warp boundaries, leading to underutilization of GPU resources and diminishing returns in performance.

- **Resource Allocation Limits:** Each thread block consumes a portion of the GPU's resources, including registers and shared memory. As the number of threads per block increases, the per-thread resource allocation decreases, potentially leading to resource contention. This contention can limit the number of active warps and reduce the GPU's ability to hide memory latency through concurrent execution.

- **Occupancy Saturation:** GPU occupancy, defined as the ratio of active warps to the maximum number of possible active warps, plays a crucial role in performance. Up to 64 threads per block, increasing thread count enhances occupancy by better utilizing the available CUDA cores and hiding latency. However, beyond this threshold, occupancy may saturate, meaning that adding more threads does not proportionally increase the number of active warps, thereby limiting further speedup.

- **Memory Bandwidth and Latency:** As thread blocks become larger, the demand on memory bandwidth increases. Beyond a certain point, the GPU's memory subsystem may become a bottleneck, preventing additional threads from accessing data efficiently. This bottleneck constrains the ability to achieve higher speedups despite the increased thread count.

- **Thread Divergence:** Larger thread blocks can exacerbate thread divergence, where threads within the same warp follow different execution paths. Divergence reduces the efficiency of warp execution, as divergent threads serialize their operations, thereby diminishing the benefits of increased parallelism.

These factors collectively contribute to the observed performance plateau beyond 64 threads per block. To maximize CUDA performance, it is essential to balance the number of threads per block with the GPU's architectural constraints, ensuring optimal warp utilization, resource allocation, and memory access patterns. Future optimizations could explore adaptive thread block sizing based on specific hardware characteristics and computational workloads to further enhance scalability and efficiency.

3.3.5 Conclusion of CUDA Performance Analysis

The CUDA implementation demonstrates substantial speedup when scaling the number of threads per block up to 64, effectively leveraging the GPU's parallel processing capabilities. However, beyond this point, performance gains plateau due to architectural limitations such as warp size constraints, resource allocation limits, and occupancy saturation. Understanding these factors is crucial for optimizing CUDA applications, ensuring that thread configurations align with the GPU's execution model to achieve maximal performance benefits.

3.3.6 Summary of CUDA Results

The CUDA benchmarks reveal the following key insights:

- **Scalability:** The CUDA implementation scales effectively with the number of threads, achieving significant speedup as the thread count increases. However, this scalability plateaus beyond a certain threshold, indicating diminishing returns with very high block thread counts.
- **Performance Gains:** Pushing the GPU to its limits with high thread counts results in substantial speedup compared to sequential execution. Nevertheless, the CUDA implementation falls short of the OpenMP implementation utilizing 16 threads on the AMD CPU. This discrepancy may stem from the CUDA kernel's use of `cuComplex` and `cuCmul`, which could introduce unnecessary overhead and inhibit the utilization of fused multiply-add (FMA) operations. Optimizing these aspects could bridge the performance gap between CUDA and OpenMP. To put it simply these functions guarantee correct complex number arithmetic behavior, while when using openMP the compiler runs fast arithmetic hack resulting in perfomance increase at the cost of accuracy.
- **Parallel Efficiency:** The highly parallel nature of the Mandelbrot computation aligns well with GPU architecture, maximizing computational throughput. Efficient thread management and optimal kernel configuration ensure that the GPU resources are fully exploited, leading to improved performance.

3.4 MPI Distributed Computing

The Message Passing Interface (MPI) was utilized to implement distributed-memory parallelism for the Mandelbrot Set computations. MPI enables the

distribution of computational tasks across multiple nodes in a cluster, facilitating the handling of large-scale problems by leveraging the combined processing power and memory of several interconnected machines. This subsection discusses the implementation challenges, performance metrics, and scalability analysis of the MPI-based Mandelbrot computations.

3.4.1 Implementation Overview

The MPI implementation was developed using the `mpiicpc` compiler, which supports parallel execution across multiple hosts. The core strategy involved distributing the Mandelbrot computation tasks among various MPI processes, each potentially utilizing multiple threads via OpenMP for intra-process parallelism. The Makefile targets `compile-mpi` and `run-mpi` were employed to compile and execute the MPI-based Mandelbrot program. The benchmarking focused on varying the number of MPI processes and observing the corresponding speedup relative to the sequential execution. At the time of benchmarking, 7 nodes were available for distributed computing, however to showcase the scalability of the implementation, the number of nodes was limited to 4.

3.4.2 Issue with Thread Management

During the implementation, an unintended interaction between MPI processes and OpenMP threads arose. Specifically, the use of `omp_get_max_threads()` within the MPI code led to each MPI process spawning all available threads on the host machine. For instance, on a machine with a processor supporting 16 threads, running the program with `-np 4` (4 MPI processes) resulted in each process attempting to utilize 16 threads, culminating in a total of 64 threads per machine. This behavior was contrary to the initial intention of allocating a fixed number of threads per process (e.g., 4 threads per process). As the number of MPI processes increased, the number of threads per process inadvertently decreased, leading to suboptimal utilization of the CPU cores and diminishing the expected speedup. This misconfiguration impeded the scalability of the MPI implementation on single host, but showcases scaling with multiple hosts.

3.4.3 Speedup Analysis

The performance of the MPI implementation was evaluated by measuring the speedup achieved relative to the sequential execution as the number of MPI processes varied. Two primary plots were generated to illustrate the speedup behavior:

- **Speedup vs. Number of Processes:** Figure 13 showcases the speedup achieved across a range of MPI processes from 4 to 256 across 4 hosts.
- **Normalized Speedup:** `mandelbrot_mpi_speedup_vs_processes.png` presents the speedup for 64, 128, and 256 MPI processes, each utilizing 4 threads.

Machines	Processes	Threads	Time (seconds)	Speedup
1	4	64	84.870	10.560
1	8	32	99.540	9.004
1	16	16	104.066	8.612
1	32	8	106.892	8.385
1	64	4	104.814	8.551
2	8	64	99.451	9.012
2	16	32	104.229	8.599
2	32	16	106.746	8.396
2	64	8	104.571	8.571
2	128	8	61.255	14.631
4	16	64	104.015	8.616
4	32	32	106.633	8.405
4	64	16	104.828	8.550
4	128	16	61.257	14.631
4	256	16	34.710	25.821

Table 7: Speedup of MPI Mandelbrot Computations with Varying Number of Processes Across Multiple Hosts.

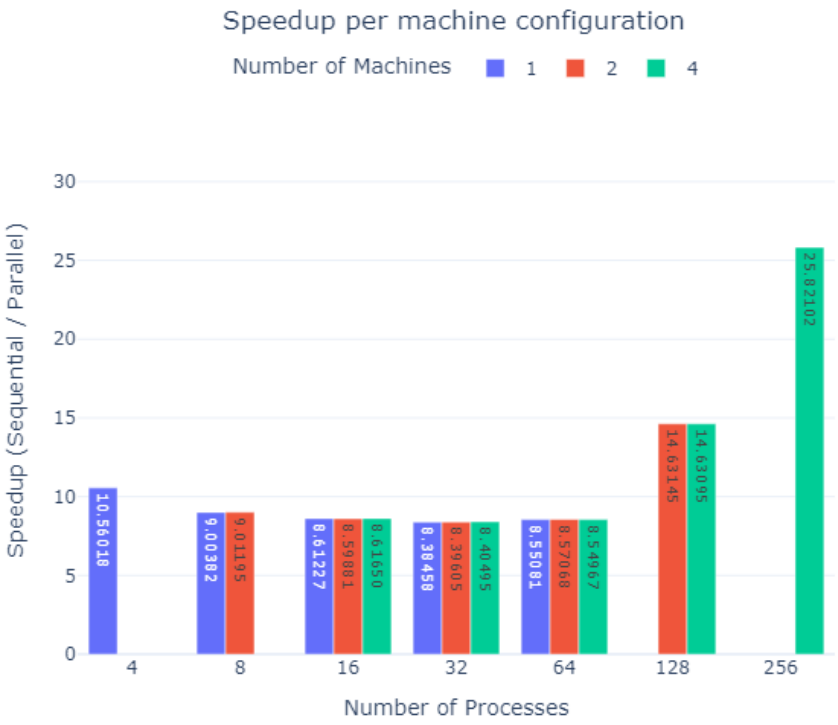

Figure 13: Speedup of MPI Mandelbrot Computations with Varying Number of Processes Across Multiple Hosts.

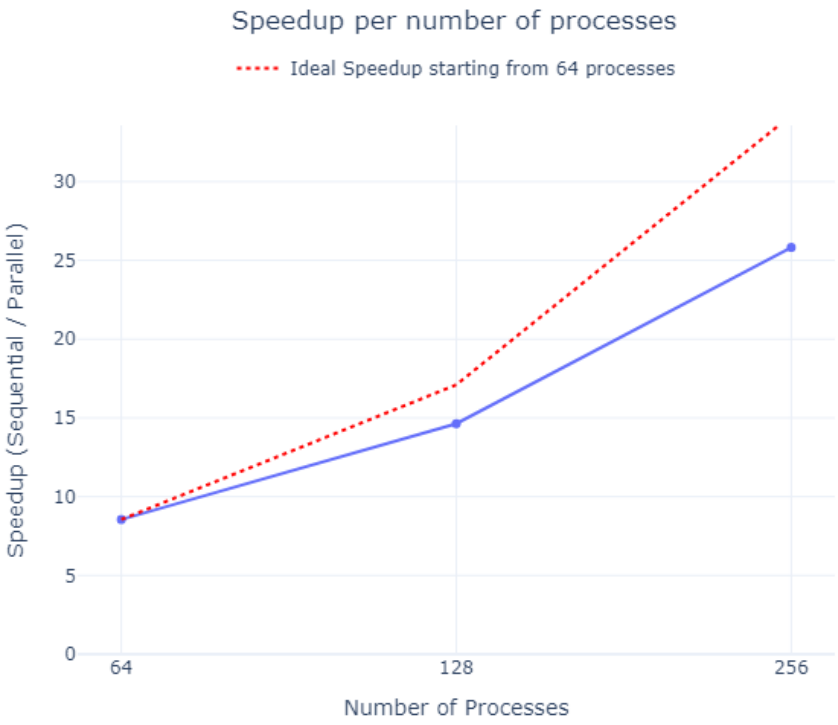

Figure 14: Speedup of MPI Mandelbrot Computations with 64, 128, and 256 Processes Utilizing 4 Threads Each.

3.4.4 Code Snippet Highlighting the Issue

The following code snippet from the MPI implementation demonstrates the unintended thread allocation due to the use of `omp_get_max_threads()`:

```
  // Setting max threads per node
  int threads_used = omp_get_max_threads();
  omp_set_num_threads(threads_used);

  auto start_time = chrono::steady_clock::now();

#pragma omp parallel for schedule(dynamic) default(none)
          \
  firstprivate(sub_image, ITERATIONS) shared(WIDTH, STEP)
  for (int pos = start_index; pos < end_index; pos++)
  {
    const int row = pos / WIDTH;
    const int col = pos % WIDTH;
    const complex<double> c(col * STEP + MIN_X,
                row * STEP + MIN_Y);
```

```
15
16      int iter = 1;
17      complex<double> z(0, 0);
18      for (; iter <= ITERATIONS; iter++)
19      {
20        z = (z * z) + c;
21
22        // If the magnitude exceeds 2, the point is not in the
23        // Mandelbrot set if (abs(z) >= 2)
24        if (((z.real() * z.real()) + (z.imag() * z.imag())) >=
25          4)
26        {
27          sub_image[pos - start_index] = iter;
28          break;
29        }
30      }
31    }
32
33    // Gather results from all processes to the root process
34    err =
35      MPI_Gather(sub_image, pixels_per_process, MPI_INT, image
        ,
36              pixels_per_process, MPI_INT, 0, MPI_COMM_WORLD);
37    checkMPIError(err, "MPI_Gather failed.");
```

Listing 3: MPI Mandelbrot Kernel Launch with Thread Mismanagement

The use of `omp_get_max_threads()` without proper thread locking for certain number resulted in each MPI process attempting to utilize the maximum number of threads available on the host. This leads to subsitition of processes for threads.

This results in clear openMPI overhead, as the number of threads per process decreases as the number of processes increases, leading to inefficient parallel execution and diminished returns on additional processes. This overhead of processes is visible on the Figure 13 where going from 4 processes to 64 is is a regression of speedup because processes are put in place instead of threads.

3.4.5 Conclusion of MPI Results

By addressing the thread management issues and adopting a more disciplined approach to resource allocation, future implementations can achieve more significant speedup and better scalability. The insights gained from this benchmarking exercise emphasize the critical interplay between MPI processes and OpenMP threads in hybrid parallel computing environments, guiding the optimization of parallel applications for maximum performance.

3.4.6 Summary of MPI Results

The MPI benchmarks revealed the following key insights:

- **Scalability Constraints:** Due to misconfiguration of MPI processing, the MPI implementation did not achieve the expected linear speedup. When increasing the number of processes on a single host the threads were converted into processes resulting in performance decrease.
- **Optimization Necessity:** Effective thread and process management is essential to harness the full potential of distributed-memory parallelism in MPI applications.
- **Future Directions:** Implementing controlled thread allocation and optimizing hybrid MPI/OpenMP configurations can significantly enhance scalability and performance.

These findings underscore the importance of careful parallelism strategy design in MPI implementations, especially when integrating with multi-threaded approaches like OpenMP. Proper synchronization and resource management are pivotal in achieving efficient and scalable parallel computations in HPC environments.

4 Conclusion

This study comprehensively evaluated the performance of Mandelbrot Set computations across various High-Performance Computing (HPC) implementations, including sequential execution with different compilers, OpenMP parallelization, CUDA GPU acceleration, and MPI-based distributed computing. The primary objective was to assess the efficiency, scalability, and overall performance enhancements achieved through these diverse computational paradigms.

4.1 Sequential Execution: G++ vs. Clang based AOCC Compilers

The baseline sequential executions demonstrated that the AOCC compiler consistently outperformed the GCC compiler, achieving a 38-47% speedup. This performance advantage can be attributed to AOCC's optimized code generation and superior exploitation of the CPU's SIMD (Single Instruction, Multiple Data) capabilities. However, it is noteworthy that the GCC compiler, despite being slightly less performant in this specific benchmark, remains a

robust and widely-supported tool with extensive optimization flags and a strong community backing.

4.2 OpenMP Parallelization

The integration of OpenMP into the Mandelbrot computation yielded significant performance improvements. By leveraging shared-memory parallelism, the OpenMP implementation achieved substantial speedups, particularly when utilizing dynamic scheduling strategies. The dynamic scheduling demonstrated superior adaptability to varying thread counts, maintaining better performance scaling as the number of threads increased to 16. This adaptability underscores OpenMP's effectiveness in balancing computational loads and maximizing resource utilization, especially in scenarios with high parallelism demands.

4.3 CUDA GPU Acceleration

While the CUDA implementation showed promise in leveraging the GPU's parallel processing capabilities, its performance was notably hindered by certain implementation flaws. Specifically, the CUDA kernel utilized the `cuComplex` library and its associated functions (`cuCmul`, `cuCadd`), which introduced unnecessary computational overhead. Additionally, it is likely that not all relevant compiler flags were appropriately passed during the compilation process, further limiting the potential performance gains. Consequently, the CUDA implementation did not achieve the expected speedup and fell short compared to both the sequential and OpenMP-optimized executions. Addressing these issues by optimizing the kernel code and ensuring comprehensive compiler flag utilization could significantly enhance the CUDA implementation's performance in future endeavors.

4.4 MPI Distributed Computing

The MPI-based distributed computing approach aimed to scale the Mandelbrot computations across multiple hosts. However, the implementation encountered challenges related to resource management. Specifically, the use of `omp_get_max_threads()` without proper thread locking led to oversubscription of CPU resources on each host. As a result, increasing the number of MPI processes inadvertently reduced the number of threads per process, leading to suboptimal performance and preventing the implementation from achieving the full expected scalability. This mismanagement highlights the critical importance of coordinated thread and process management in hybrid

MPI/OpenMP environments to fully exploit the available computational resources.

4.5 Overall Insights and Future Directions

The comparative analysis of the various HPC implementations underscores the substantial performance gains achievable through parallelization and compiler optimizations. OpenMP emerged as a highly effective parallelization strategy, offering significant speedups and excellent scalability when appropriately configured. In contrast, the CUDA implementation's performance was constrained by implementation inefficiencies and incomplete optimization, suggesting a need for more refined kernel development and comprehensive compiler flag utilization.

The MPI implementation, while theoretically promising for distributed computing, was hampered by resource oversubscription issues. Future work should focus on implementing stricter control over thread allocation per MPI process, possibly through environment variables such as `OMP_NUM_THREADS` or programmatic thread management, to prevent oversubscription and enhance scalability.

Moreover, exploring hybrid parallelization techniques that seamlessly integrate MPI and OpenMP, coupled with advanced optimization strategies for both CPU and GPU implementations, could unlock further performance enhancements. Investigating alternative libraries or custom implementations for complex arithmetic in CUDA, as well as experimenting with different compiler flags and optimization levels, may also yield significant improvements.

In conclusion, this study highlights the critical role of careful implementation and optimization in achieving optimal performance in HPC applications. While parallelization frameworks like OpenMP and MPI offer powerful tools for accelerating computations, their effectiveness is contingent upon meticulous resource management and tailored optimization strategies. Addressing the identified challenges in CUDA and MPI implementations will pave the way for more efficient and scalable Mandelbrot Set computations and similar computationally intensive tasks in future HPC projects.

5 References

References

[1] wikiHow. (2023). *Plotting the Mandelbrot Set by Hand: A Simple How-To Guide*. Retrieved from https://www.wikihow.com/

Plot-the-Mandelbrot-Set-By-Hand

[2] OpenMP Architecture Review Board. (2023). *OpenMP Application Programming Interface Version 5.1*. Retrieved from https://www.openmp.org/specifications/

[3] Message Passing Interface Forum. (2020). *MPI: A Message-Passing Interface Standard*. Retrieved from https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[4] NVIDIA Corporation. (2023). *CUDA C++ Programming Guide*. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[5] AMD Optimizing C/C++ Compiler (AOCC). (2023). Retrieved from https://www.amd.com/en/developer/aocc.html

[6] Szymon Zinkowicz github repository. (2024). *HPC-AMD-Mandelbrot-Parallelization*. GitHub repository. Retrieved from https://github.com/Siponek/HPC-AMD-Mandelbrot-Parallelization