Hign Performance Computing 2023/2024

Assignment 2:

Exercise 2b with MPI only
Exercise 2c with OpenMP only

Malasi Denis
IN2000213

Parallel Implementation of the Quicksort Algorithm Using MPI: Performance Analysis and Scalability

1.Introduction

Sorting large datasets efficiently is a fundamental problem in computer science, with applications ranging from database management to scientific computing. As the size of data grows, traditional serial algorithms like Quicksort, although efficient, can become a bottleneck due to their computational limits. Parallel computing offers a way to overcome these limitations by distributing the computational workload across multiple processors.

In this report, we focus on implementing a parallel version of the Quicksort algorithm using MPI (Message Passing Interface). The goal is to distribute the sorting task across multiple processes, allowing the algorithm to scale and reduce execution time for large datasets. MPI is well-suited for this problem as it enables communication between processes running on distributed memory systems, ensuring scalability across different computing architectures.

The implementation tackles the challenge of dividing and sorting the data in parallel, ensuring that the workload is evenly distributed across processes. We will evaluate the performance of the algorithm by analyzing its strong and weak scalability.

2.Execution Environment

The parallel implementation of the Quicksort algorithm was executed on the Orfeo system, utilizing a THIN node with 24 processes. The node used for this task has the capability to run multiple processes in parallel, making it ideal for analyzing the performance of parallel algorithms like Quicksort.
To compile the program, we used the mpicc compiler from the MPI library, which is based on GCC version 14.2.1 (Red Hat 14.2.1-1).
The multi-core architecture of the Orfeo node enabled efficient parallelization, allowing us to test the scalability of the algorithm in both strong and weak scaling scenarios.

3.Strategy for Parallel Data Distribution and Sorting

The strategy for this parallel implementation of Quicksort using MPI revolves around efficient data distribution and workload management across multiple processes. The key goal is to divide the dataset among MPI processes in a balanced way, allowing each process to independently sort its portion of the data, followed by a final merge step. This approach leverages parallel processing to reduce the overall time required to sort large datasets.

Data Distribution and Chunking

The first phase involves distributing the dataset from the root process to all other MPI processes. The root process is responsible for generating the full dataset, an array of double-precision floating-point numbers. The dataset is divided into chunks using *MPI_Scatterv*, which allows for a flexible distribution of data, ensuring that each process receives a balanced share, even when the dataset size is not evenly divisible by the number of processes.
To manage this, the arrays *recvcounts* and *displs* are employed. The *recvcounts* array defines how many elements each process receives, while *displs* specifies the offset in the global array from where each process's data starts. This ensures that each process gets a fair distribution of the workload, allowing the algorithm to handle datasets of any size and effectively distribute them among the processes.

Local Quicksort Execution

Once the data is distributed, each process works independently to sort its allocated chunk of the array using the Quicksort algorithm. Quicksort is chosen because of its average-case time complexity of *O(nlogn),* making it highly efficient for large datasets. Since the sorting is done independently by each process, no inter-process communication is required during this phase, maximizing parallel efficiency. The Quicksort algorithm used in each process follows a classic divide-and-conquer approach, where the array is recursively partitioned and sorted. Since each process handles only a subset of the original data, the time spent sorting locally is significantly reduced compared to a serial implementation.

Gathering and Final Merge

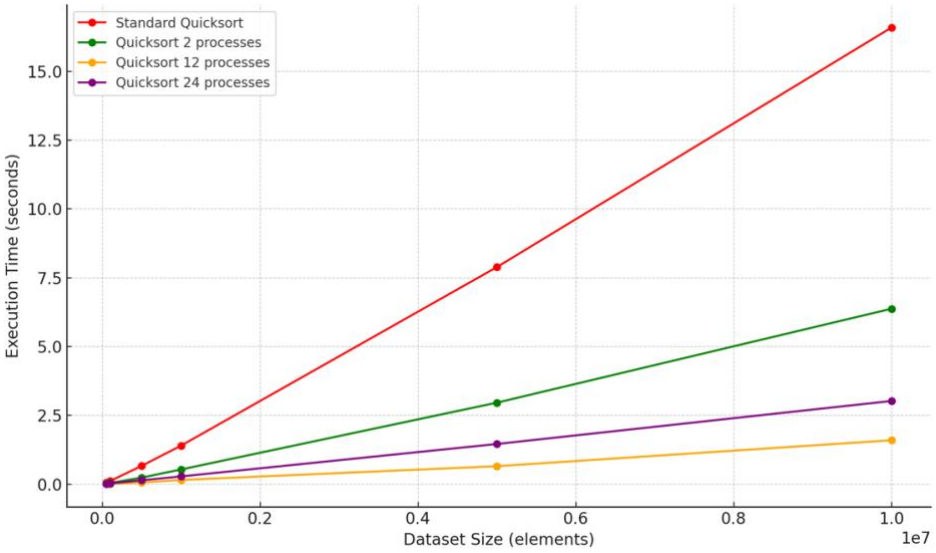
After local sorting is completed, the sorted subarrays from each process must be combined into a globally sorted array. This is done using *MPI_Gatherv,* which collects the sorted chunks from all processes and sends them to the root process.
The root process then performs a final merge to combine the locally sorted arrays. In the current implementation, this merge is performed serially by iterating over the sorted chunks and merging them element by element.
However, the current serial merge step at the root process presents a limitation in terms of scalability, especially for large datasets and higher process counts. Addressing this bottleneck by implementing a parallel merge would further enhance the scalability of the algorithm.

4. Results and Performance Analysis

In this section, we present the results of the parallel Quicksort implementation using MPI, focusing on both strong scaling and weak scaling performance. The analysis was performed on the Orfeo system, using a thin node with 24 processes.

Comparison Between MPI Quicksort and Standard Quicksort

Before delving into the strong and weak scaling analysis, it is important to compare the performance of the MPI-based Quicksort implementation with a standard serial Quicksort. This comparison helps demonstrate the benefits of parallelization and sets a baseline for evaluating scalability.

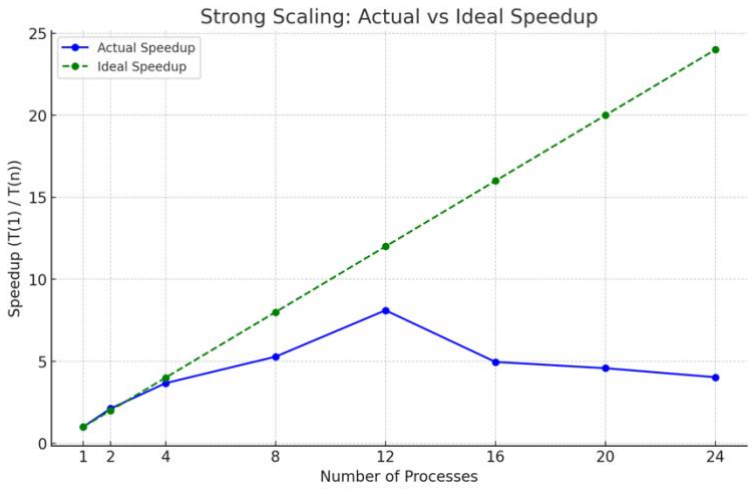


The graph compares the performance of the standard Quicksort (in red) with the MPI-based Quicksort using 2, 12, and 24 processes. As the dataset size increases, the standard Quicksort takes significantly longer. The MPI version shows clear improvements, with execution time decreasing as the number of processes increases.

Strong Scaling Results

Strong scaling measures how the execution time changes as we increase the number of MPI processes while keeping the total dataset size constant. The goal is to observe how well the parallel implementation can reduce the execution time as more resources (processes) are added.
For this analysis, we fixed the dataset size at *N=1,000,000* elements and varied the number of MPI processes from 1 to 24. The results are shown in the table and graph below.

PROCESSES	TIME (S)
1	1.093297
2	0.516297
4	0.298099
8	0.207023
12	0.1348245
16	0.220418
20	0.238780
24	0.271339



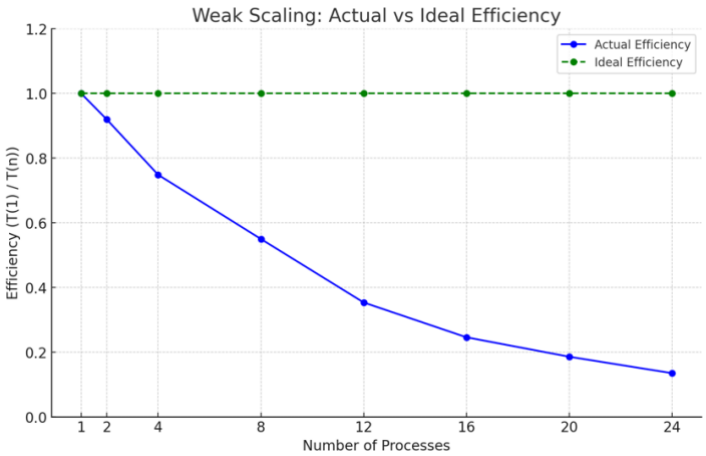
From the graph, we can observe that the speedup increases as more processes are added, with significant improvements up to about 12 processes. However, beyond this point, the actual speedup starts to decline, deviating from the ideal speedup (shown by the green dashed line).

This behavior suggests that while the algorithm scales well initially, adding more processes introduces overhead, such as inter-process communication and synchronization costs. Additionally, the merge step, which is handled serially at the root process, likely becomes a bottleneck as the number of processes increases, limiting further improvements in execution time.
These results highlight the effectiveness of the parallel implementation for moderate numbers of processes but also point to potential inefficiencies that could be addressed to improve scalability, particularly for larger numbers of processes.

Weak Scaling Results

Weak scaling measures how the execution time changes as both the dataset size and the number of MPI processes increase proportionally. The goal is to observe how well the algorithm maintains efficiency when the workload per process remains constant.
For this analysis, we increased the dataset size proportionally with the number of processes, starting from *N =1,000,000* elements for 1 process and going up to *N=24,000,00* elements for 24 processes. The results are presented in the table and graph below.

PROCESSES	ELEMENTS	TIME (S)
1	1.000.000	1.022230
2	2.000.000	1.111749
4	4.000.000	1.365972
8	8.000.000	1.858919
12	12.000.000	2.889953
16	16.000.000	4.153120
20	20.000.000	5.496289
24	24.000.000	7.560878



The graph shows that, as expected, the efficiency decreases as the number of processes and dataset size grow. Initially, with a small number of processes, the efficiency remains close to the ideal value of However, as the number of processes increases, the overhead of communication, synchronization, and data merging becomes more significant, resulting in lower efficiency.
These results indicate that the parallel implementation handles weak scaling well up to a certain point, but further optimizations might be necessary to maintain efficiency with larger process counts. The serial merging step at the root process likely contributes to the observed drop in efficiency.

Overall, the results demonstrate that the MPI Quicksort implementation is effective for parallelizing sorting operations, particularly for moderate process counts. However, further improvements in the merging phase and communication overhead are necessary to achieve better scalability, especially as the number of processes and the dataset size increase.

5.Future Improvements

Although the MPI-based Quicksort implementation shows solid performance, there are a couple of areas where it could be enhanced for better scalability and efficiency.
One key aspect is the merge phase, which is currently performed serially at the root process. As the number of processes increases, this step becomes a bottleneck, limiting the potential speedup. Implementing a parallel merge, where the merging process is distributed across multiple processes, could significantly reduce this bottleneck and improve overall efficiency, particularly when working with larger datasets.

Another important area for improvement is the selection of the pivot in the Quicksort algorithm. At present, the pivot is chosen as the last element of the array, which can lead to inefficient partitioning in some cases. A more robust pivot selection strategy, such as using the "median of three" or a randomized pivot, would lead to better partitioning, particularly in worst-case scenarios, ultimately improving the overall sorting performance.
By addressing these two areas, the MPI Quicksort implementation could scale more effectively and maintain higher efficiency, especially when working with larger datasets and a greater number of processes.

6.Conclusion

The parallel implementation of the Quicksort algorithm using MPI demonstrates significant improvements in performance over the standard serial version, particularly when applied to large datasets. Through the strong and weak scaling analyses, we observed that the algorithm scales well with a moderate number of processes, achieving substantial speedup and maintaining reasonable efficiency. However, as the number of processes increases, the limitations of the current implementation become more apparent, particularly due to the serial merge phase and the overhead introduced by inter-process communication.

Despite these challenges, the results show that parallelization is an effective strategy for reducing execution time in sorting algorithms like Quicksort. The potential improvements, such as implementing a parallel merge and refining the pivot selection strategy, offer clear paths to enhancing scalability and performance further.

In conclusion, this MPI-based Quicksort implementation provides a strong foundation for parallel sorting and serves as a useful demonstration of how distributed computing can be leveraged to solve large-scale problems efficiently. With further optimizations, the algorithm could perform even better on larger datasets and with more processes, solidifying its utility in high-performance computing environments.

Parallel Implementation of Mandelbrot Set Generation using OpenMP: Performance Analysis and Scalability

1.Introdution

The Mandelbrot set is a famous example of a complex fractal, where each point in a two-dimensional complex plane undergoes iterative computations to determine whether it belongs to the set. Due to the iterative nature of these computations, the process of generating the Mandelbrot set can become computationally expensive, especially for high-resolution images and large iteration limits.

The objective of this project is to implement a parallelized solution to generate the Mandelbrot set using OpenMP. OpenMP, a widely adopted parallel programming model, allows for efficient parallelization on shared-memory systems by distributing computational tasks across multiple processor cores. By exploiting OpenMP's parallel capabilities, the goal is to reduce the execution time for generating the Mandelbrot set without sacrificing accuracy.

In this report, we detail the algorithmic approach used for parallelization, the specific optimizations applied, and an analysis of the implementation's performance. Both strong and weak scaling tests were conducted to evaluate how well the implementation scales with an increasing number of threads. It is important to note that the execution times recorded for the scaling tests exclude the time spent generating the output .pgm file, focusing solely on the computation of the fractal itself.

2.Execution Environment

The Mandelbrot set computation was performed on a EPYC node of the Orfeo high-performance computing cluster. A single node with 128 CPU cores was allocated for each job, utilizing OpenMP for parallel execution.
The code was compiled using GCC version 14.2.1, which supports OpenMP version 5.1. This enabled efficient parallel processing across the 128 cores.
This setup provided an ideal environment for the high-performance computation required by the Mandelbrot set generation, fully leveraging the available parallel resources.

3. Parallelization Approach and Implementation Details

Parallelization Strategy

The problem at hand involves computing the Mandelbrot set, which is inherently parallelizable because each point (or pixel) on the complex plane can be computed independently. This allows for efficient parallel execution using OpenMP, a shared-memory parallel programming model.

The main strategy for parallelizing the Mandelbrot set generator is to divide the computation across multiple threads, where each thread calculates a subset of the pixels in the output image. Given that

there are no data dependencies between pixels, this is an example of an **embarrassingly parallel problem**, ideal for OpenMP.

OpenMP was chosen because it provides a simple and flexible way to distribute the workload across multiple CPU cores on a shared-memory system. The *#pragma omp parallel* for directive was used to parallelize the double loop responsible for iterating over the image pixels and performing the Mandelbrot set calculations for each.

Dynamic Scheduling

One of the challenges in generating the Mandelbrot set is that certain regions of the complex plane take more computational effort than others, particularly points near the boundary of the Mandelbrot set. To address this, dynamic scheduling was applied in the OpenMP parallel loop, which allows threads to request new tasks as soon as they complete their current work. This dynamic approach helps balance the workload among threads, preventing some threads from being idle while others are stuck with more complex calculations.

The directive used in the code is:

$$\#pragma\ omp\ parallel\ for\ collapse(2)\ schedule(dynamic)$$

The *collapse(2)* clause was used to merge the two loops (for rows and columns of the image), ensuring that the scheduling is applied to the combined loop iteration space.

Thread Management and Memory Considerations

Each thread writes the results of its calculations to a shared matrix M, where each element of the matrix corresponds to the number of iterations needed for the corresponding point in the complex plane to escape the set. Because each thread works on a different portion of the image, there is no need for explicit synchronization mechanisms (e.g., locks), as no two threads write to the same memory location. This makes the parallelization straightforward and efficient, as thread synchronization overhead is minimized.

File Output

The output of the program is written in the Portable Graymap (.pgm) format, which represents the generated fractal as a grayscale image. The matrix M, containing the iteration counts, is written to the .pgm file after all the computations are complete.

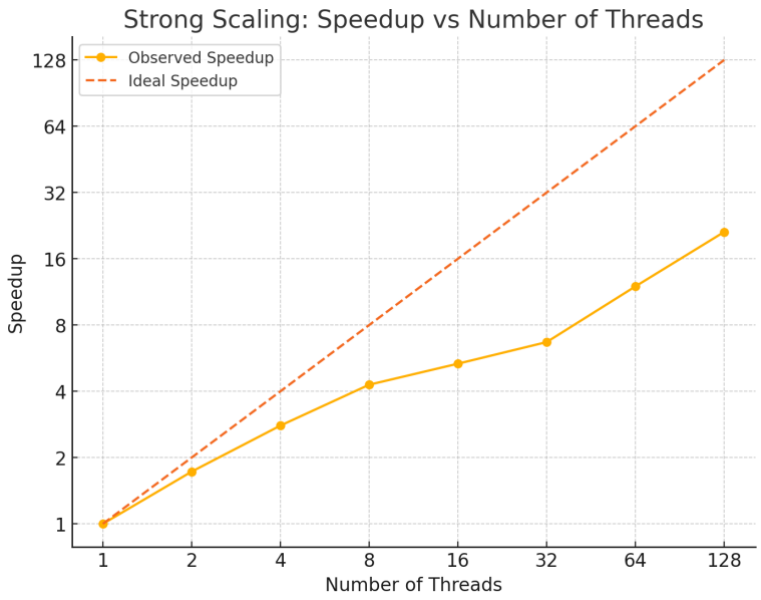
4. Results and Performance Analysis

The performance of the Mandelbrot set generator was assessed using **strong scaling** and **weak scaling**. Strong scaling measures speedup with a fixed problem size as the number of threads increases, while weak scaling tests efficiency as both the problem size and thread count grow proportionally.

Strong Scaling Results

Strong scaling was tested by fixing the problem size and progressively increasing the number of threads to observe how the computational time decreases. Ideally, the speedup should be proportional to the number of threads, resulting in a linear relationship between threads and speedup. The results of the strong scaling test are summarized in the table and graph below.

THREADS	TIME (S)
1	21.911611
2	12.691843
4	7.842132
8	5.114257
16	4.104780
32	3.274480
64	1.828183
128	1.034187



The strong scaling results indicate a clear improvement in performance as the number of threads increases. The computational time drops from 21.91 seconds with 1 thread to just 1.03 seconds with 128 threads. This shows that the parallelization is effective, but the speedup is not perfectly linear.

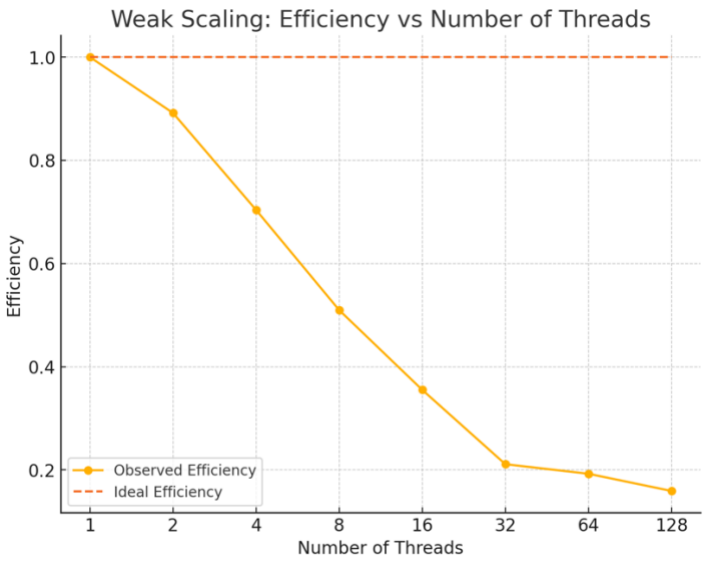
Up to 16 threads, the speedup follows the ideal curve quite closely, suggesting efficient parallel execution. However, as more threads are added, the performance gain starts to diminish. This is likely due to overhead associated with thread management, scheduling, and memory contention, where multiple threads compete for shared resources.

Despite these limitations, the results demonstrate a substantial improvement in performance, making the parallel implementation highly effective for this problem size. The strong scaling efficiency could likely be further improved by optimizing memory access patterns or reducing thread synchronization overhead.

Weak Scaling Results

The weak scaling test was conducted by increasing both the problem size and the number of threads proportionally, aiming to maintain a constant workload per thread. Ideally, the computational time should remain stable as the problem scales, indicating high efficiency. The following table and graph present the results of the weak scaling test.

THREADS	PROBLEM SIZE	TIME (S)
1	1000x1000	1.378521
2	1414x1414	1.545798
4	2000x2000	1.958566
8	2828x2828	2.704784
16	4000x4000	3.882996
32	5656x5656	6.535210
64	8000x8000	7.163590
128	11313x11313	8.664932



In the weak scaling test, the problem size increases proportionally with the number of threads. Rather than doubling the dimensions of the problem directly, they grow by the square root of 2, ensuring that the overall area (or number of pixels) doubles. This method maintains a constant workload per thread as the number of threads increases.

The results show a gradual increase in computational time as more threads are added, starting from 1.37 seconds with 1 thread and rising to 8.66 seconds with 128 threads. As expected, the efficiency decreases as the thread count grows, and this trend is clearly visible in the graph. Initially, with a smaller number of threads (from 2 to 8), the efficiency remains relatively high, but beyond 16 threads, the efficiency drops more sharply.

This decline can be attributed to several factors that affect performance in larger parallel computations. As the problem size and the number of threads increase, the demand for memory access grows, which can slow down the computation due to limitations in memory bandwidth. Additionally, managing a large number of threads introduces more overhead, such as scheduling and synchronization, which becomes more noticeable as the thread count increases. With more threads in play, there is also greater contention for shared resources, like memory and cache, which further reduces the overall efficiency.

Despite the drop in efficiency, the implementation is still able to handle larger problem sizes with reasonable computational times, demonstrating that it can effectively scale for larger workloads, though with diminishing returns as the system reaches its resource limits.

The results of both strong and weak scaling tests show that the OpenMP parallelization of the Mandelbrot set generator is highly effective, achieving substantial speedup and scalability. While strong scaling demonstrates significant performance gains, particularly up to 16 threads, weak scaling

highlights the challenges of maintaining efficiency as the problem size and thread count grow. Despite these limitations, the implementation proves capable of handling large-scale computations efficiently.

5.Future Improvements

While the current OpenMP implementation of the Mandelbrot set generator has achieved impressive performance gains, there are several ways in which the program could be further optimized and improved.

One potential direction is to adopt a hybrid parallelization approach that combines both OpenMP and MPI. OpenMP works well for parallelizing tasks within a single node, but integrating MPI would enable the program to scale across multiple nodes. This could be particularly beneficial for larger problem sizes, allowing the computation to be distributed across several machines, taking advantage of the additional cores in a distributed memory system.

More sophisticated dynamic load balancing techniques could also be implemented. While dynamic scheduling is already used in the current implementation, certain regions of the Mandelbrot set may still require significantly more computation than others. Implementing adaptive load balancing or work-stealing algorithms could ensure that no threads remain idle while waiting for others to complete, leading to a more even distribution of the workload.

Finally, one of the most promising areas for future improvement could be exploring GPU acceleration. By offloading the computation to a GPU using frameworks such as CUDA or OpenCL, the Mandelbrot set generator could take advantage of the massive parallelism that modern GPUs offer. This would allow for even faster computation, particularly for very large problem sizes that require extensive processing power.

6.Conclusion

The OpenMP implementation of the Mandelbrot set generator successfully achieved significant performance improvements, especially in strong scaling, where computation time decreased as more threads were used. This demonstrates the effectiveness of OpenMP in handling highly parallelizable problems.

However, in weak scaling, the efficiency dropped as the problem size and thread count increased, primarily due to memory contention and overhead from thread management. Despite these challenges, the implementation handled larger problem sizes within reasonable times, showing its scalability.

Future improvements, such as integrating MPI and exploring GPU acceleration, could further enhance the program's performance and scalability. Overall, this project demonstrates the benefits of parallel computing, while also highlighting some of the challenges in achieving optimal scalability.