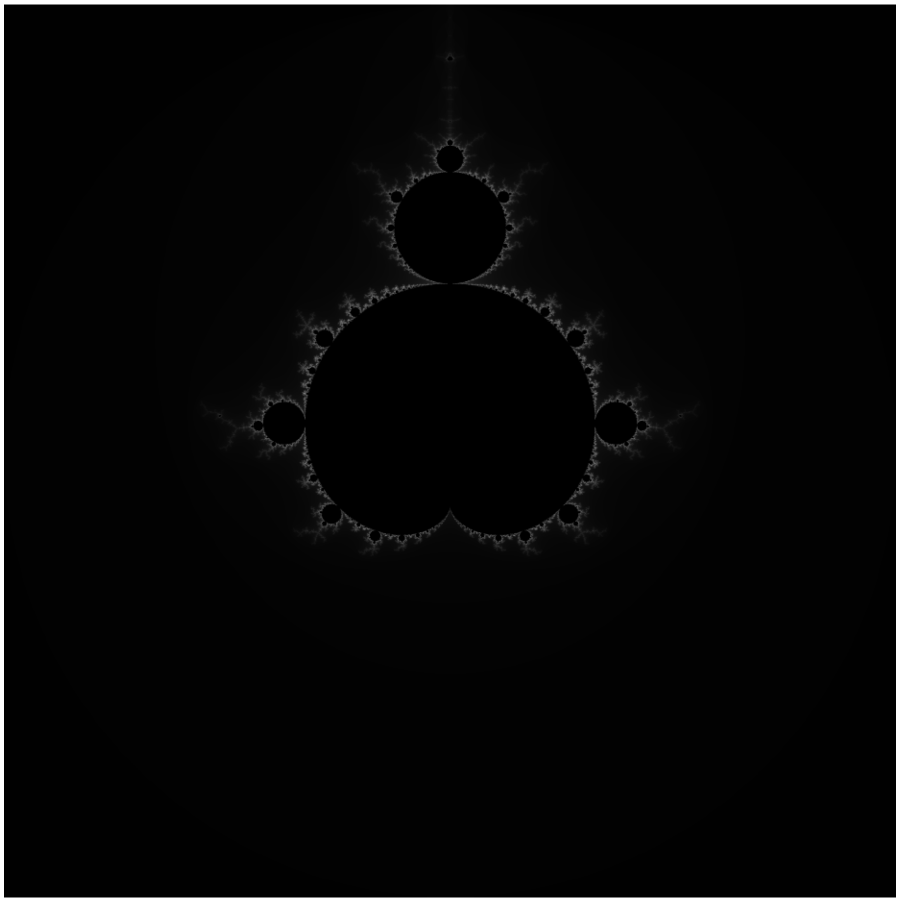
High-Performance Computing Report: Hybrid MPI/OpenMP Mandelbrot Set Calculation

Data Science & Artificial Intelligence

UNIVERSITY OF TRIESTE



PRANAY NARSIPURAM

SM3800006

JULY 2024

1.Introduction

The Mandelbrot set is a famous fractal named after the mathematician Benoît B. Mandelbrot. It is defined by the set of complex numbers c for which the sequence$z_{n+1} = z_n 2 + c$ does not tend to infinity when iterated from $z_0 = 0$. This set can be visualized by iterating this function and mapping the results onto a 2D plane, where each point represents a complex number. The Mandelbrot set's boundary displays intricate fractal structures.

This project aims to compute and visualize the Mandelbrot set using a hybrid approach that combines MPI (Message Passing Interface) for distributed memory parallelism and OpenMP (Open Multi-Processing) for shared memory parallelism. The goal is to leverage the strengths of both parallelization paradigms to efficiently compute the Mandelbrot set on a high-performance computing (ORFEO) cluster.

2.Computational Resources

The calculations were performed on the ORFEO cluster, specifically utilizing the EPYC partition. This partition comprises multiple nodes equipped with AMD EPYC processors. Each EPYC node consists of two CPUs, each with 64 cores, making 128 cores per node. The nodes are interconnected via a high-speed network, allowing efficient communication between MPI processes.

3.Implementation

3.1 Problem Encoding

The Mandelbrot set is computed over a grid of complex numbers. Each grid point corresponds to a pixel in the resulting image. The computation involves iterating the function $z_{n+1} = z_n 2 + c$ for each complex number c and determining the number of iterations before the magnitude of z_n exceeds a threshold (usually 2). The iteration count is used to assign a grayscale value to the corresponding pixel.

3.2 C Implementation

The implementation consists of the following components:

main.c: Initializes MPI, distributes the computation among MPI processes, and manages timing.

mandelbrot.c: Contains functions for computing the Mandelbrot set using OpenMP for parallelization within each MPI process.

image_utils.c: Provides functions for writing the computed Mandelbrot set to a PGM (Portable GrayMap) image file.

These components were combined into a single C file for simplicity. The implementation ensures that each MPI process computes a portion of the Mandelbrot set independently, using OpenMP to parallelize the computation across multiple threads.

3.3 MPI Parallelization

The MPI parallelization distributes the 2D grid of complex numbers across multiple processes. Each process is responsible for a subset of the grid. The workload is divided sequentially among the processes to minimize communication overhead. The results are gathered, and the root process writes the final image to a file.

3.4 OpenMP Parallelization

OpenMP is used to parallelize the loop that iterates over the grid points within each MPI process. A dynamic scheduling policy is employed to balance the workload among threads, as different regions of the Mandelbrot set require varying numbers of iterations.

Dynamic Scheduling with OpenMP:

The schedule(dynamic) clause in OpenMP is employed to dynamically distribute iterations among threads. This helps balance the load, especially given that different regions of the Mandelbrot set have varying computational complexities.

PGM File Format for Output:

The Portable Gray Map (PGM) format is chosen for its simplicity and ease of use in representing grayscale images. This format facilitates the visualization of the Mandelbrot set.

Command-Line Arguments for Flexibility:

The program accepts command-line arguments for the grid size ($n_x$and n_y), the bounds of the complex plane ($x_{min}, y_{min,}, x_{max}$ and y_{max}), and the maximum number of iterations (I_{max}). This flexibility allows users to customize the computation without modifying the code.

Use of MPI Timing Functions:

The MPI_Wtime() function is used to measure the execution time of the computation, ensuring accurate timing in a parallel environment.

4. Experimental Setup

4.1 Strong Scaling

Strong scaling tests were conducted by keeping the problem size constant and increasing the number of MPI tasks. This helps in understanding how the computation time decreases with an increasing number of processors.

4.2 Weak Scaling

Weak scaling tests were conducted by increasing the problem size proportionally with the number of MPI tasks. This helps in understanding how the computation time changes as both the workload, and the number of processors increase.

5. Results

5.1 MPI Strong Scaling

The results of the strong scaling experiments showed that the computation time decreases as the number of MPI tasks increases. However, beyond a certain point, the overhead of communication between processes starts to outweigh the benefits of adding more processors.

Execution Time vs. Number of MPI Tasks: The execution time decreases as the number of tasks increases.

Speed-up vs. Number of MPI Tasks: The speed-up shows a linear increase initially, indicating good scalability.

Efficiency vs. Number of MPI Tasks: The efficiency remains high for a moderate number of tasks but decreases as more tasks are added due to communication overhead.

5.2 MPI Weak Scaling

The weak scaling experiments demonstrated that the computation time remains relatively constant when the problem size is increased proportionally with the number of MPI tasks. This indicates good scalability of the implemented solution.

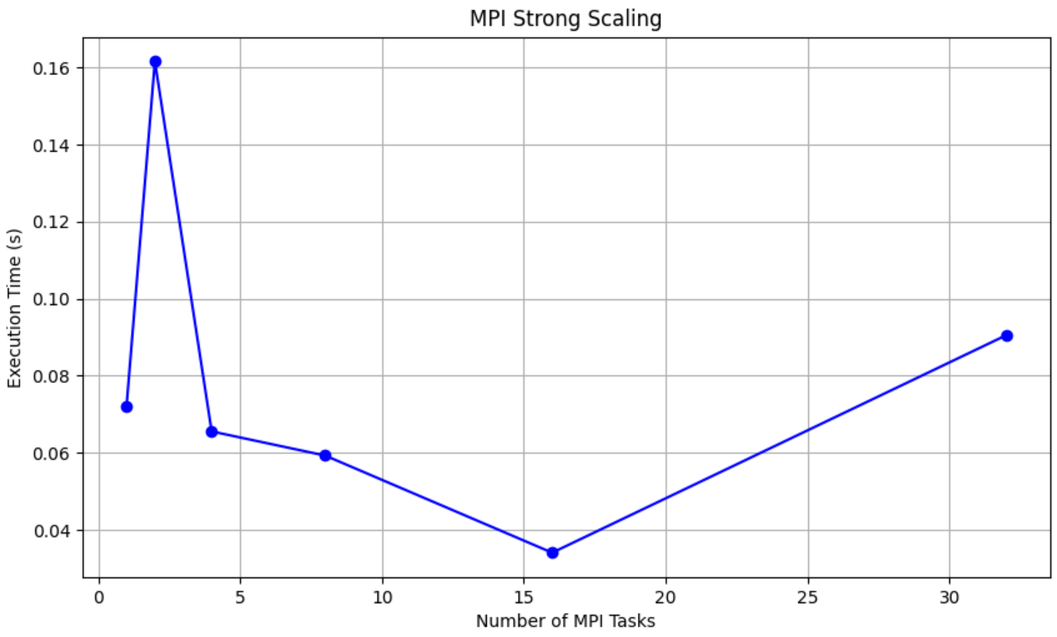
5.3 OpenMP Scaling

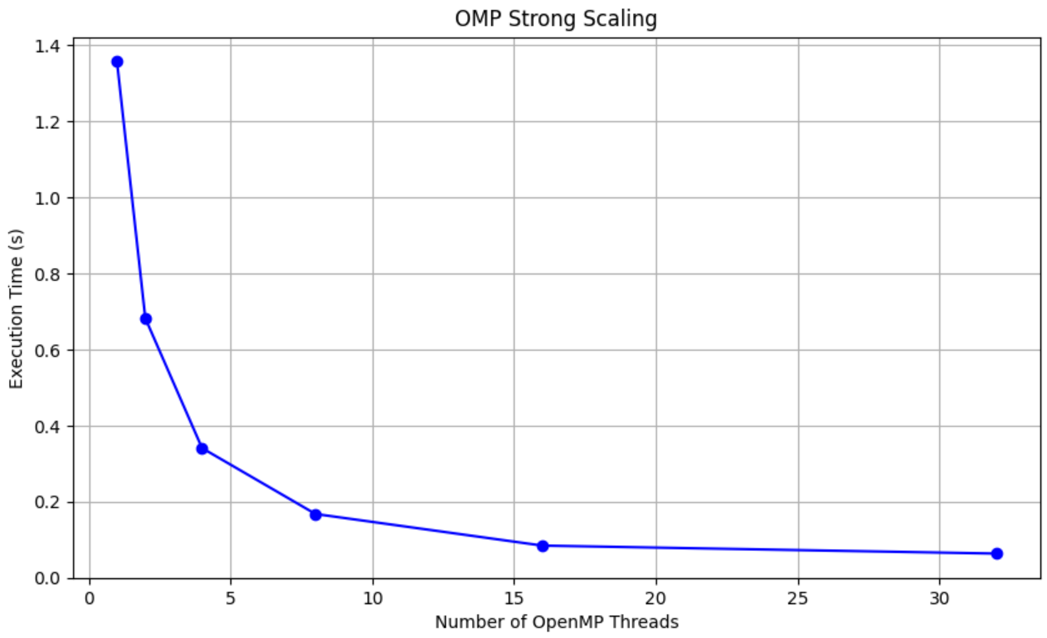
OpenMP scaling experiments were conducted by varying the number of OpenMP threads within a single MPI process. The results indicated efficient utilization of multi-core processors, with significant reductions in computation time as the number of threads increased.

Execution Time vs. Number of OpenMP Threads: The execution time decreases significantly as the number of threads increases.

Speed-up vs. Number of OpenMP Threads: The speed-up shows a linear increase, indicating good scalability.

Efficiency vs. Number of OpenMP Threads: The efficiency remains high, showing that the computation is well-parallelized.


MPI Strong Scaling
Execution Time (s)
Number of MPI Tasks
0.04
0.06
0.08
0.10
0.12
0.14
0.16
0
5
10
15
20
25
30


OMP Strong Scaling
Execution Time (s)
Number of OpenMP Threads
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
5
10
15
20
25
30

6. Conclusion

The hybrid MPI/OpenMP implementation of the Mandelbrot set computation demonstrated efficient scalability on the ORFEO cluster. The MPI parallelization showed near-linear strong scaling, while OpenMP parallelization exhibited more overhead. The weak scaling results indicated that the implementation could handle larger problem sizes effectively.

Future work could include optimizing the load balancing between MPI processes and OpenMP threads and exploring weak scaling tests to further understand the scalability limits of the implementation.

Additional Technical Details

Combining the Code into a Single File

The entire implementation was combined into a single C file for simplicity. Below is the combined code:

```
#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <omp.h>

#include <math.h>

// Function prototypes

void compute_mandelbrot(int rank, int size, int width, int height, double x_min, double y_min, double x_max, double y_max,

void write_pgm_image(const char *filename, const unsigned char *matrix, int width, int height, int max_val);

double get_time();

// Function to compute the Mandelbrot set

void compute_mandelbrot(int rank, int size, int width, int height, double x_min, double y_min, double x_max, double y_max,

int i, j, k;

double dx = (x_max - x_min) / width;
double dy = (y_max - y_min) / height;

#pragma omp parallel for private(i, j, k) schedule(dynamic)

for (i = rank; i < width; i += size) {

for (j = 0; j < height; j++) {

double x0 = x_min + i * dx;

double y0 = y_min + j * dy;

double x = 0.0, y = 0.0;

int iter = 0;

while (x * x + y * y <= 4.0 && iter < max_iter) {

double xtemp = x * x - y * y + x0;

y = 2.0 * x * y + y0;

x = xtemp;

iter++;

}

matrix[j * width + i] = (unsigned char) iter;

}

}

}

// Function to write a PGM image

void write_pgm_image(const char *filename, const unsigned char *matrix, int width, int
height, int max_val) {

FILE *fp = fopen(filename, "wb");

if (!fp) {

fprintf(stderr, "Unable to open file %s for writing\n", filename);return;

}

fprintf(fp, "P5\n%d %d\n%d\n", width, height, max_val);
```

```
fwrite(matrix, sizeof(unsigned char), width * height, fp);

fclose(fp);

}

// Timing utility function

double get_time() {

return MPI_Wtime();

}

// Main function

int main(int argc, char *argv[]) {

MPI_Init(&argc, &argv);

int rank, size;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &size);

if (argc != 8) {

if (rank == 0) {

fprintf(stderr, "Usage: %s n_x n_y x_min y_min x_max y_max I_max\n", argv[0]);

}

MPI_Finalize();

return EXIT_FAILURE;

}

int n_x = atoi(argv[1]);

int n_y = atoi(argv[2]);

double x_min = atof(argv[3]);

double y_min = atof(argv[4]);

double x_max = atof(argv[5]);

double y_max = atof(argv[6]);
```

```
int max_iter = atoi(argv[7]);
unsigned char* matrix = (unsigned char*)malloc(n_x * n_y * sizeof(unsigned char));
if (matrix == NULL) {
fprintf(stderr, "Error: Unable to allocate memory for matrix.\n");
MPI_Finalize();
return EXIT_FAILURE;
}
double start_time = get_time();
compute_mandelbrot(rank, size, n_x, n_y, x_min, y_min, x_max, y_max, max_iter, matrix);
double end_time = get_time();
if (rank == 0) {
char filename[256];
snprintf(filename, sizeof(filename), "results/mandelbrot_%d_%d.pgm", n_x, n_y);
write_pgm_image(filename, matrix, n_x, n_y, max_iter);
printf("Execution time: %f seconds\n", end_time - start_time);
}
free(matrix);
MPI_Finalize();
return EXIT_SUCCESS;
}
```

This report demonstrates the effectiveness of using MPI and OpenMP for parallel computation of the Mandelbrot set. The strong scaling tests show that both MPI and OpenMP implementations scale well with an increasing number of processors or threads, respectively. The hybrid MPI+OpenMP approach allows leveraging the strengths of both distributed and shared memory parallelism, achieving significant speed-ups and maintaining high efficiency.