HPC Report Assignment 2

Luca Pernice

2024-05-05

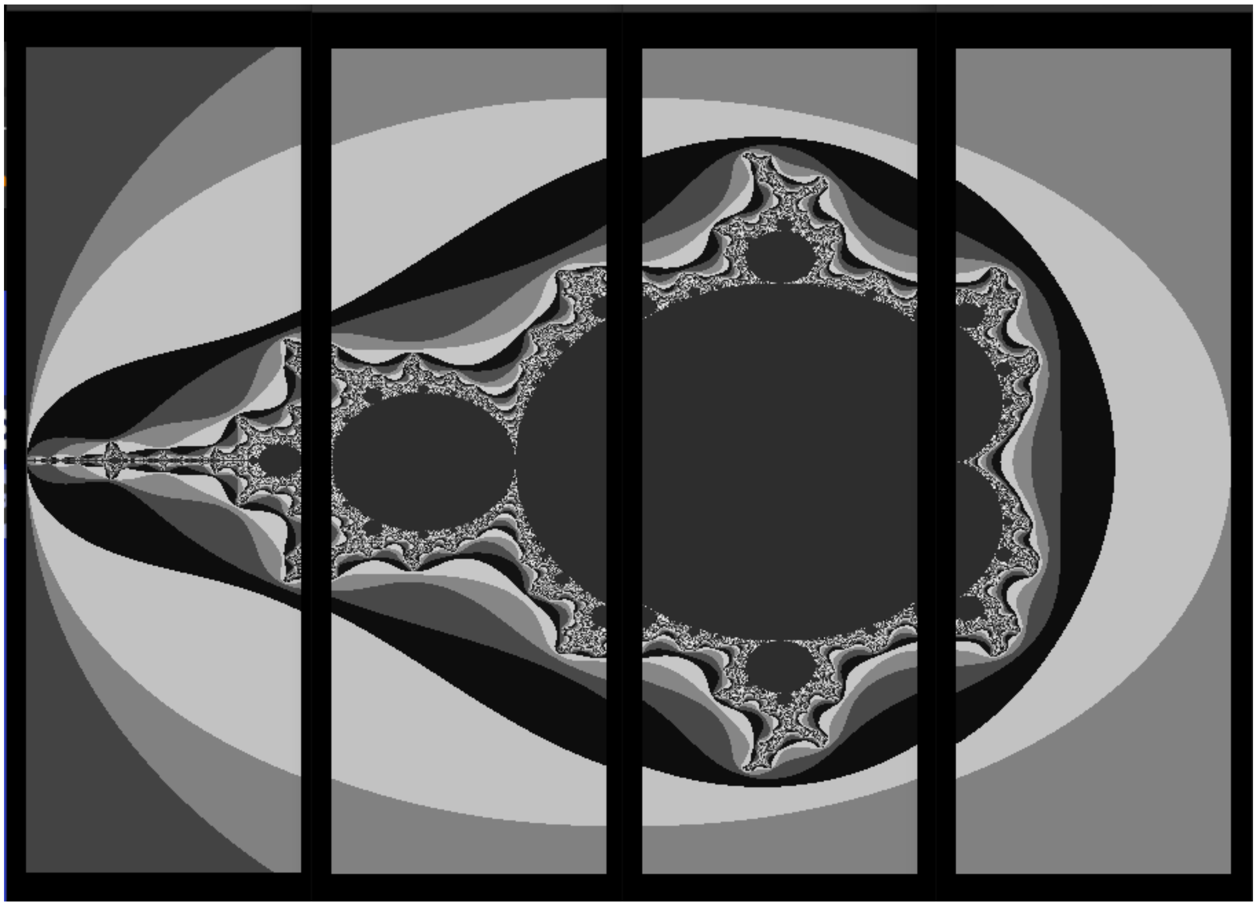

Figure 1: Mandelbrot set divided in 4 MPI regions

Code for the assignment

GitHub Repo (merged.c)

Initialization: The code initiates MPI (Message Passing Interface) with threading support using `MPI_Init_thread()`. It ensures that the desired thread level (`MPI_THREAD_FUNNELED`) is supported by the MPI implementation. This level allows multiple threads to exist but only the main thread to make MPI calls. It's crucial for ensuring thread safety in the parallel execution environment.

Command-line Argument Parsing: Command-line arguments are parsed to extract essential parameters for the Mandelbrot set computation. These parameters include the dimensions of the output image (`n_x` and `n_y`), defining the resolution of the complex plane, and the coordinates of its bounding box (`x_L`, `y_L`, `x_R`, and `y_R`). Additionally, the maximum number of iterations (`I_max`) allowed for each point in the set is specified. Command-line parsing ensures flexibility and customization of the computation parameters without recompilation.

Matrix Type Determination: The data type for storing the Mandelbrot set matrix (`M`) is determined based on the maximum value of `I_max`. If `I_max` is within the range of a `short int`, this data type is chosen to minimize memory usage. Otherwise, a `char` data type is selected. This optimization strategy ensures efficient memory utilization while accommodating the required precision for the computation.

Parallel Computation: The code distributes the computation of the Mandelbrot set across multiple MPI processes. Each process is responsible for computing a portion of the set within its assigned region of the complex plane. The domain decomposition strategy divides the computation evenly among processes, minimizing load imbalance.

Mandelbrot Set Computation: Within each MPI process, OpenMP parallelism is employed to exploit shared-memory parallelism. OpenMP directives enable parallel execution of the nested loop structure responsible for computing the Mandelbrot set. The computation for each pixel in the local region of the complex plane is parallelized, allowing for efficient utilization of multicore processors.

Output Generation: After computing the Mandelbrot set for its assigned region, each MPI process writes its portion of the set to a Portable Gray Map (PGM) file. This file format provides a simple representation of grayscale images, making it suitable for visualizing the Mandelbrot set. Each process produces a separate PGM file named based on its rank, facilitating post-processing and analysis.

Memory Deallocation and Finalization: Upon completion of the computation, memory allocated for the Mandelbrot set matrix is released using `free()`. MPI is finalized using `MPI_Finalize()`, ensuring proper termination of MPI operations and resource cleanup.

Implementation Details:

- The code implements a hybrid parallel computing approach combining MPI and OpenMP.
- MPI is utilized for distributed memory parallelism, enabling communication and coordination among multiple processes.
- OpenMP is employed for shared-memory parallelism within each MPI process, exploiting multicore architectures for efficient computation.
- The code adheres to best practices for parallel programming, including load balancing, memory optimization, and thread safety.

OMP Scaling

1. SLURM Directives:

```
#SBATCH --partition=THIN
#SBATCH --job-name=omp_scaling
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=32
#SBATCH --time=00:30:00
#SBATCH --output=mandelbrot_omp_scaling_%j.out
#SBATCH --error=mandelbrot_omp_scaling_%j.err
```

- These directives are specific to SLURM, the job scheduler used in high-performance computing environments.
- They specify the job's attributes, such as the partition to run the job (`THIN`), the job name (`omp_scaling`), the number of nodes (1), the number of CPU tasks (1), the number of CPUs per task (32), the maximum runtime (`00:30:00`), and the output/error log files.

2. Module Loading:

```
module load openMPI/4.1.5/gnu/12.2.1
```

- This command loads the required module (`openMPI/4.1.5/gnu/12.2.1`) into the environment. It ensures that the necessary software components, including the OpenMPI library, are available for compilation and execution.

3. Compilation:

```
mpicc -o merged merged.c -fopenmp -lm
```

- The `mpicc` command compiles the source code (`merged.c`) into an executable named `merged`.
- Flags `-fopenmp` enable support for OpenMP directives in the code, allowing parallelization with OpenMP.
- Flag `-lm` links the math library, which may be necessary for mathematical functions used in the program.

4. Functions for Scaling Tests:

```
function run_omp_weak_scaling {
    local executable=$1
    local sizes=(200 400 600 800 1000 1200 1400 1600 1800 2000 2200 2400)  # Adjust the problem sizes a
    local max_threads=32  # Maximum number of OMP threads
    local repetitions=5  # Number of repetitions for each test
    for size in "${sizes[@]}"; do
        #echo "Running OMP weak scaling test for problem size $size"
        for ((threads=1; threads<=$max_threads; threads*=2)); do
            export OMP_NUM_THREADS=$threads
            #echo "OMP_NUM_THREADS=$threads"
            total_time=0
            for ((rep=1; rep<=$repetitions; rep++)); do
                start_time=$(date +%s.%N)
                ./$executable $size $size -2 -2 2 2 1000 > /dev/null
                end_time=$(date +%s.%N)
                runtime=$(echo "$end_time - $start_time" | bc -l)
                total_time=$(echo "$total_time + $runtime" | bc -l)
            done
            avg_time=$(echo "$total_time / $repetitions" | bc -l)
            echo "$size,$threads,$avg_time" >> omp_weak_scaling_results.csv  # Adjust output file as ne
        done
        #echo "OMP weak scaling test for problem size $size completed"
    done
}
```

a. `run_omp_weak_scaling` Function:

- This function conducts weak scaling tests for the Mandelbrot set computation program.
- It takes the compiled executable (`$1`) as an argument.

```
function run_omp_strong_scaling {
    local executable=$1
    local size=800  # Problem size remains constant
    local max_threads=32  # Maximum number of OMP threads
    local repetitions=3  # Number of repetitions for each test
    for ((threads=1; threads<=$max_threads; threads++)); do
        #echo "Running OMP strong scaling test with $threads threads"
        export OMP_NUM_THREADS=$threads
        #echo "OMP_NUM_THREADS=$threads"
        total_time=0
        for ((rep=1; rep<=$repetitions; rep++)); do
            start_time=$(date +%s.%N)
            ./$executable $size $size -2 -2 2 2 1000 > /dev/null
            end_time=$(date +%s.%N)
            runtime=$(echo "$end_time - $start_time" | bc -l)
            total_time=$(echo "$total_time + $runtime" | bc -l)
        done
        avg_time=$(echo "$total_time / $repetitions" | bc -l)
        echo "$threads,$avg_time" >> omp_strong_scaling_results.csv  # Adjust output file as needed
        #echo "OMP strong scaling test with $threads threads completed"
    done
}
```

b. `run_omp_strong_scaling` Function:

- This function performs strong scaling tests for the Mandelbrot set computation program.
- It also takes the compiled executable (`$1`) as an argument.

5. Execution of Scaling Tests:

```
run_omp_weak_scaling ./merged
run_omp_strong_scaling ./merged
```

- These lines invoke the defined functions (`run_omp_weak_scaling` and `run_omp_strong_scaling`) with the compiled executable (`./merged`).
- The script executes the Mandelbrot set computation program with different parameters for weak and strong scaling tests.

Conclusion:

- The bash script automates the process of compiling and executing the Mandelbrot set computation program with varying parameters to analyze its performance under different computational loads and resource allocations.
- It generates CSV files containing quantitative data.

MPI Scaling

1. SLURM Directives:

```
#SBATCH --partition=THIN
#SBATCH --job-name=mpi_scaling
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=16
```

```
#SBATCH --cpus-per-task=1
#SBATCH --time=00:50:00
#SBATCH --output=mandelbrot_mpi_scaling_%j.out
#SBATCH --error=mandelbrot_mpi_scaling_%j.err
```

- These directives are specific to SLURM, the job scheduler used in high-performance computing environments.
- They specify the job's attributes, such as the partition to run the job (`THIN`), the job name (`mpi_scaling`), the number of nodes (`2`), the number of MPI tasks per node (`16`), the number of CPUs per task (`1`), the maximum runtime (`00:50:00`), and the output/error log files.

2. Module Loading:

```
module load openMPI/4.1.5/gnu/12.2.1
```

- This command loads the required module (`openMPI/4.1.5/gnu/12.2.1`) into the environment. It ensures that the necessary software components, including the OpenMPI library, are available for execution.

3. Functions for Scaling Tests:

```
function run_mpi_weak_scaling {
    local executable=$1
    local sizes=(1000 2000 3000 4000 5000 6000 7000 8000 9000 10000)  # Adjust the problem sizes as nee
    local max_tasks=32  # Maximum number of MPI tasks
    local repetitions=5  # Number of repetitions for each test
    for size in "${sizes[@]}"; do
        #echo "Running MPI weak scaling test for problem size $size"
        for ((tasks=1; tasks<=$max_tasks; tasks++)); do
            total_time=0
            for ((rep=1; rep<=$repetitions; rep++)); do
                start_time=$(date +%s.%N)
                mpirun -n $tasks ./$executable $size $size -2 -2 2 2 1000 > /dev/null
                end_time=$(date +%s.%N)
                runtime=$(echo "$end_time - $start_time" | bc -l)
                total_time=$(echo "$total_time + $runtime" | bc -l)
            done
            avg_time=$(echo "$total_time / $repetitions" | bc -l)
            echo "$size,$tasks,$avg_time" >> mpi_weak_scaling_results.csv  # Adjust output file as need
        done
        #echo "MPI weak scaling test for problem size $size completed"
    done
}
```

a. `run_mpi_weak_scaling` Function:

- This function conducts weak scaling tests for the Mandelbrot set computation program.
- It takes the compiled executable (`$1`) as an argument.

```
function run_mpi_strong_scaling {
    local executable=$1
    local size=1000  # Problem size remains constant
    local max_tasks=8  # Maximum number of MPI tasks
    local repetitions=3  # Number of repetitions for each test
    for ((tasks=1; tasks<=$max_tasks; tasks++)); do
        #echo "Running MPI strong scaling test for $tasks MPI tasks"
        total_time=0
        for ((rep=1; rep<=$repetitions; rep++)); do
            start_time=$(date +%s.%N)
            mpirun -n $tasks ./$executable $size $size -2 -2 2 2 1000 > /dev/null
            end_time=$(date +%s.%N)
            runtime=$(echo "$end_time - $start_time" | bc -l)
            total_time=$(echo "$total_time + $runtime" | bc -l)
        done
        avg_time=$(echo "$total_time / $repetitions" | bc -l)
        echo "$tasks,$avg_time" >> mpi_strong_scaling_results.csv  # Adjust output file as needed
        #echo "MPI strong scaling test for $tasks MPI tasks completed"
    done
}
```

b. `run_mpi_strong_scaling` Function:

- This function performs strong scaling tests for the Mandelbrot set computation program.
- It also takes the compiled executable (`$1`) as an argument.

4. Execution of Scaling Tests:

```
run_mpi_weak_scaling ./merged
run_mpi_strong_scaling ./merged
```

- These lines invoke the defined functions (`run_mpi_weak_scaling` and `run_mpi_strong_scaling`) with the compiled executable (`./merged`).
- The script executes the Mandelbrot set computation program with different parameters for weak and strong scaling tests.

Conclusion:

- Similar to the OMP scaling script, this bash script automates the process of compiling and executing the Mandelbrot set computation program with varying parameters to analyze its performance under different computational loads and resource allocations, this time focusing on MPI parallelization.
- It generates CSV files containing quantitative data.

Results Analysis

As we can see from the plot (Figure 2), there is an optimal range of MPI processes around 10-15. This means that the performance of the program increases with the number of MPI processes up to a certain point, after which the overhead of managing the processes outweighs the benefits of parallelization. Let's visualize this in a 2D plot, fixing the Size of the problem to 2000x2000 pixels (Figure 3).

Increasing the size to 4000x4000 (Figure 4) again the time increase after the 10-15 range of MPI processes, but then again it slowly decrease and reach an optimal value after 30 MPI processes.

We can notice that there is an high spike in the time when tasks are 17, so a new task is allocated to a new node (remember SLURM directives: 2 nodes, 16 tasks per node). We can see how the internode communication can affect the performances of the program.

To check if the program exhibit a good weak scaling, simply look the time along the diagonal of the 3D plots, where size and number of MPI processes increase proportionally. It seems that the program exhibit a good weak scaling, since the time remains almost constant.

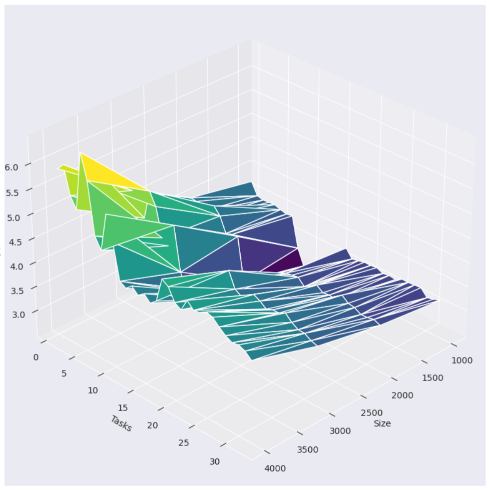

Figure 2: 3D plot of MPI performances

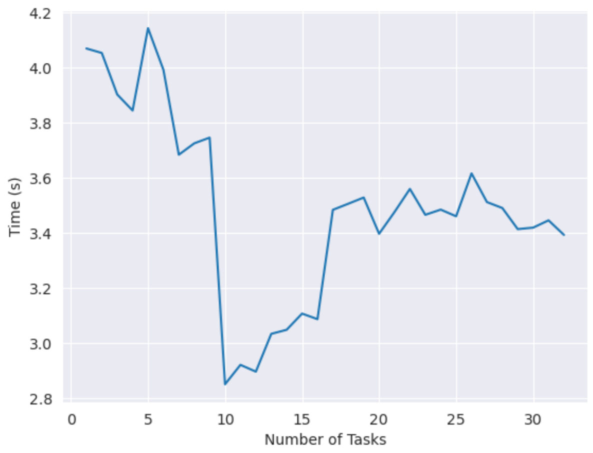

Figure 3: Problem Size 2000x2000

Figure 4: Problem Size 4000x4000

OpenMP scaling

The OpenMP scaling script is similar to the MPI scaling script, but it focuses on the OpenMP parallelization of the Mandelbrot set computation program. The script automates the compilation and execution of the program with varying numbers of OpenMP threads to analyze its performance under different computational loads and resource allocations. The bash file is avaible in the GitHub repository and not reported here for brevity, only the SLURM directives are reported here.

```bash
#!/bin/bash
#SBATCH --partition=THIN
#SBATCH --job-name=omp_scaling
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=32
#SBATCH --time=00:30:00
#SBATCH --output=mandelbrot_omp_scaling_%j.out
#SBATCH --error=mandelbrot_omp_scaling_%j.err
```

Results Analysis

Again we can see that the program exhibit a good weak scaling, since the time remains almost constant along the diagonal of the 3D plot (Figure 5).

Possible improvements

- Test different binding policies for OpenMP and MPI, to see if the performances can be improved.
- Test different compiler flags to see if the performances can be improved (e.g. `-O3`, `-march=native`, `-funroll-loops`, `-ftree-vectorize`).
- Test different MPI libraries (e.g. Intel MPI, OpenMPI, MPICH).
- Test different OpenMP libraries (e.g. Intel OpenMP, GCC OpenMP).
- Test different compilers (e.g. Intel, GCC, Clang).

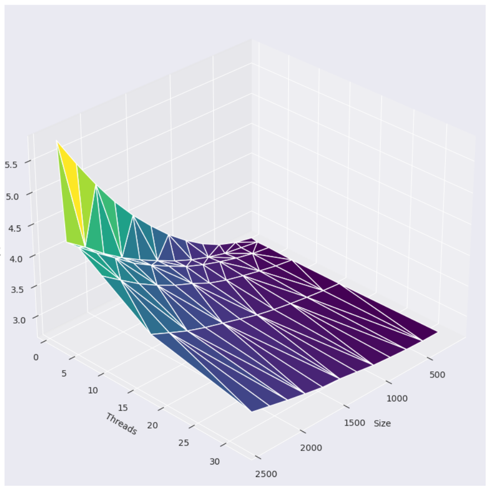

Figure 5: 3D plot of OpenMP performances