High Performance Computing assignment Exercise 2C

Piero Zappi

May 2024

The Mandelbrot set

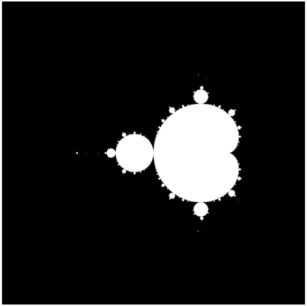
- Generated on the complex plane by iterating the complex function $f_c(z) = z^2 + c$.
- Composed of all the complex numbers c such that the sequence $z_0 = 0, z_1 = f_c(z_0), z_2 = f_c(z_1), \ldots,$ is bounded.
- Condition: $|z_n = f_c^n(0)| < 2 \quad \text{or} \quad n > I_{max}$
- Represented as a fractal shape, which exhibits self similarity.
- Each point c can be calculated independently of each other $\rightarrow$ problem is well suited for being computed efficiently in parallel.

Objective

Develop a C hybrid code to compute the Mandelbrot set using both *MPI* and *OMP*; determine the strong and weak scalings of the code.

Implementation - Algorithm

- Compute the set and produce a .pgm image:
  1. Iterate the function $f_c(z)$ for each point c.
  2. Each point c corresponds to a pixel of the image.
  3. Assign the correct value to each pixel depending on the behaviour of the sequence.
  4. The entries of a matrix of integers store the pixels' values.
  5. Convert the matrix to a .pgm file.



Implementation - MPI

- To compute the set, distribute the rows of the matrix among the *MPI* processes.
- Rows are assigned to the tasks in a round robin fashion, to reduce the load imbalance.
- Each process allocates only the strictly required memory.
- Results are gathered to the master process using MPI_Gatherv().
- The master reorders correctly the rows and produces the image.

Implementation - OMP

- Further subdivide the computational work assigned to each *MPI* task among *OMP* threads.
- Each thread is tasked with computing one row at a time.
- Parallel region introduced with the #pragma omp parallel for directive.
- Use of dynamic scheduling in order to minimize the load imbalance.
- The code is designed to reduce cache misses and to minimize the remote accesses.

```

// Compute the local part of the matrix M on each process
#pragma omp parallel for schedule(dynamic)
for (int j = 0; j < local_rows; ++j)
{
  const double y = y_L + (start_row + j * size) * dy;
  const int index = j * n_x;
  for (int i = 0; i < n_x; ++i)
  {
    double complex c = x_L + i * dx + y * I;
    local_M[index + i] = mandelbrot(c, I_max);
  }
}
```

Implementation - mandelbrot() function

Keywords to provide the compiler with hints for optimization:

- static
- inline

```

static inline int mandelbrot(const double complex c, const int max_iter)
{
  double complex z = 0.0;
  int k = 0;
  while (creal(z)*creal(z) + cimag(z)*cimag(z) < 4.0 && k < max_iter)
  {
    z = z*z + c;
    k++;
  }
  return k;
}
```

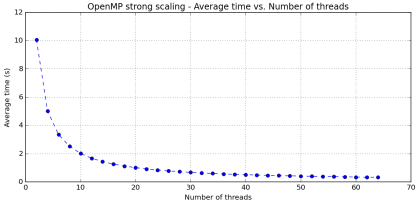
Experimental setup

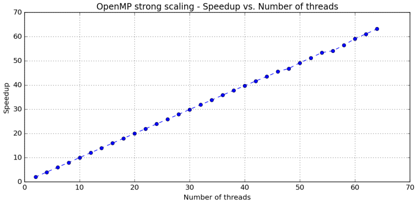
- ORFEO cluster:
  - 2 **EPYC** nodes
  - 128 cores per node $\rightarrow$ 256 cores
- Program compiled employing the highest optimization level -O3 along with the -march=native flag.
- Each time measurement collected six times $\rightarrow$ mean and standard deviation.
- **bash** scripts to automate the data collection process.

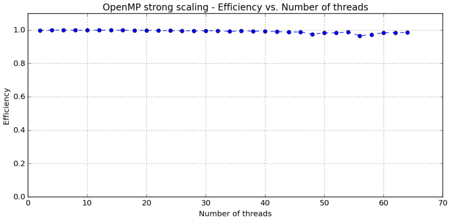
Strong scaling - OMP

Analysis:

Run the code with a single *MPI* task and gradually increase the number of *OMP* threads from 2 to 64, while keeping the total workload constant.



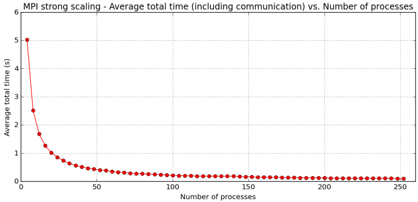


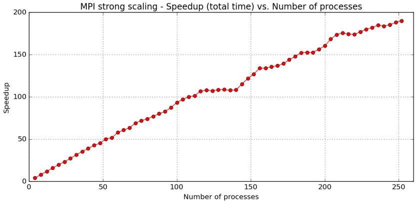


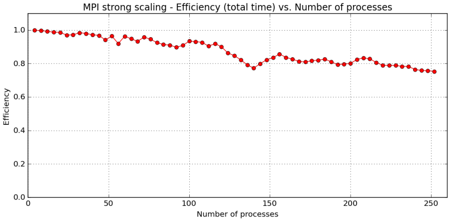
Strong scaling - MPI

Analysis:

Run the code with a single *OMP* thread per *MPI* task and gradually increase the number of *MPI* tasks from 4 to 254, while keeping the total workload constant.



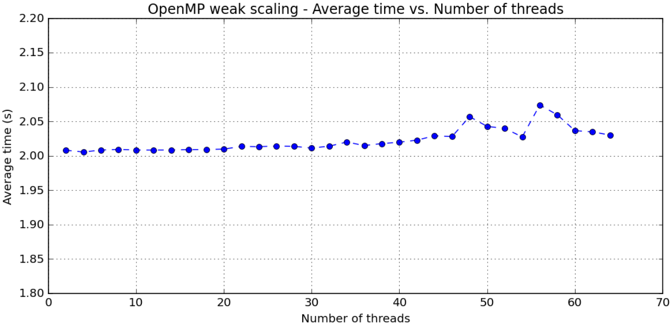




Weak scaling - OMP

Analysis:

Run the code using a single *MPI* process and gradually increase the number of threads from 2 to 64, while maintaining fixed the workload assigned to each thread (100 $\times$ 1000 pixels).



Weak scaling - MPI

Analysis:

Run the code using a single *OMP* thread per *MPI* task and gradually increase the number of *MPI* tasks from 4 to 254, while maintaining fixed the workload assigned to each process (100×1000 pixels).