High Performance Computing - Assignment 2c

Computing the Mandelbrot set in a parallel and distributed environment

Gabriele Pintus
gabrielegavino.pintus@studenti.units.it

ABSTRACT

In this report we present an implementation in C of a program which computes the Mandelbrot set in a parallel and distributed environment. To achieve this, we use the OpenMP and MPI libraries respectively. We perform multiple tests to evaluate the performance of our implementation, varying the amount of resources used and the number of processes. All the tests are performed on the ORFEO cluster of the Area Science Park. Finally we compare the results obtained with the theoretical expectations.

KEYWORDS

HPC, Mandelbrot set, Distributed Computing, MPI, OpenMP

1 INTRODUCTION TO THE PROBLEM

The Mandelbrot set is the set of all complex numbers c for which the sequence defined by $z_{n+1} = z_n^2 + c$ does not diverge. In this report we implement a C code which computes the Mandelbrot set point by point and then save the result as a PGM image. The computation of each point is independent of the others, so we can easily parallelize the computation and distribute it among multiple processes to speed up the computation. We use the OpenMP and MPI libraries to achieve this. In this report we improperly make a distinction between the two libraries, referring to OpenMP as a parallel computing library and MPI as a distributed computing library. In reality, MPI can also be used to parallelize the computation on a single machine. However, in our hybrid implementation presented few sections below, we use MPI to distribute the computation among multiple nodes of the cluster and OpenMP to parallelize the computation on each node.

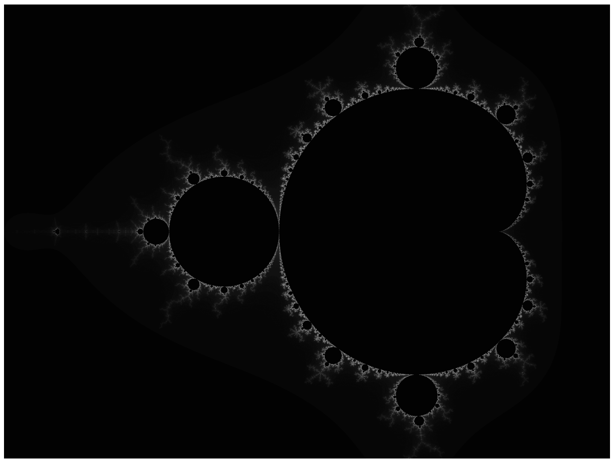


Figure 1: Rendering of the Mandelbrot set

1.1 Arithmetic Intesity

The arithmetic intensity of a program is defined as the ratio of the number of arithmetic operations to the number of memory operations. Considering only the function which computes the value of a single point of the set, we have:

- 2 mult and 1 add: `z.x*z.x+z.y*z.y`
- 2 mult, 1 sub and 1 add: `temp=z.x*z.x-z.y*z.y+c.x`
- 2 mult and 1 add: `z.y=2*z.x*z.y+c.y`
- 1 add: `++n`

for a total of 11 arithmetic operations. The memory operations are:

- 1 write: `int n = 0`
- 2 writes: `Complex z = {0,0}`
- 2 reads: `z.x*z.x+z.y*z.y`
- 3 reads and 1 write: `temp=z.x*z.x-z.y*z.y+c.x`
- 3 reads and 1 write: `z.y=2*z.x*z.y+c.y`
- 1 write: `z.x=temp`
- 1 write: `++n`

for a total of 13 memory operations. The arithmetic intensity is then

$$\frac{11 \cdot N}{10 \cdot N + 3} = \frac{11}{10 + \frac{3}{N}} \xrightarrow{N \to \infty} \frac{11}{10} = 1.1 \text{ flops/byte}$$

This value is slightly greater than 1, which means that the computation is compute-bound when the number of iterations is large, memory-bound otherwise. Notice that we did not consider the memory operations required to store the partials results when more than one arithmetic operation is involved in the same line of code. Anyway we also did not consider the major effect of the cache, which significantly reduces the time to access the memory. A more accurate analysis would probably increase the value of the arithmetic intensity, making the computation even more compute-bound.

2 IMPLEMENTATION

We begin by showing how the basic functions of the program work. We then show how we parallelize the computation using OpenMP and MPI.
The value of a point in the Mandelbrot set is computed by iterating the aforementioned sequence, this is achieved with the `mandelbrot` function. This function is invoked for each point in the image by the `mandelbrot_set` function. The image array is considered a 1-dimension array allocated contiguously in memory with the `calloc` function. This also initialize each element to 0. Finally, the result is saved as a PGM image by the `save_image` function.
The parameters one can pass to the program are the following:

```
./mandelbrot <width> <height>
             <x_min> <y_min>
             <x_max> <y_max>
             <max_iter>
```

2.1 Hardware

All the test have been performed on the ORFEO cluster of the Area Science Park, more specifically on the `THIN` partition. This is formed by 10 nodes, each with 2 Intel Xeon Gold 6126 processors and 768 GB of RAM. The nodes are interconnected with a 100 Gbit HDR Infiniband network, this allows a fast communication between the nodes with low latency.

2.2 Parallel Implementation

To parallelize the computation of the Mandelbrot with OpenMP we have to make some modifications to the code. First we define the number of threads to be used with the environment variable `OMP_NUM_THREADS`. The part of the code we want to parallelize is the loop in the `mandelbrot_set` function which iterate over all the points in the image. We add the directive `#pragma omp parallel for schedule (dynamic, OMP_CHUNK_SIZE)` before the loop to parallelize it. The `schedule` clause is used to specify how the scheduler should distribute the iterations among the threads. The `dynamic` option tells the scheduler to assign chunks of iterations to the threads as they finish their work. This is useful when the time to compute each piece of work is not constant, as is the case with the Mandelbrot set in which few points require a high number of iterations to be computed. By defining the `OMP_CHUNK_SIZE` constant we can control the size of the chunks of iterations assigned to each thread. By default this value is set to 1. By increasing it to 4 we slightly reduce the overhead of the scheduler without risking load imbalance among the threads.

2.3 Distributed Implementation

To distribute the computation of the Mandelbrot set among multiple processes we use the Message Passing Interface (MPI) library. Unlike OpenMP, MPI require a more complex setup. Here we have to explicitly define how the communication between the processes works. The main idea is to divide the image into multiple chunks, each of size `MPI_CHUNK_SIZE`, build a work queue with the chunks and then distribute the work among the processes. The queue structure is provided by the library `sys/queue.h`, each element of the queue is a `struct` which contains the starting point of the chunk, the ending point and a pointer to the next element. After having built the queue, the master process sends a chunk to each worker process with the function `MPI_Send`. We keep track of which process is working with the `processes_status` array: a value of 0 means the process is idle, a value of 1 means the process is working. We also keep track of the overall number of chunks that have been completed with the `completed_chunks` variable and the number of available chunks with the `available_chunks` variable. The available chunks are the ones that are not already assigned to a process. If the number of available chunks is greater than 0, the master process assign a work chunk to itself, without using any MPI function obviously. As a consequence, the master process has two responsibilities: to compute the mandelbrot set (as any other process) and to distribute the work among the processes. When a worker process has finished its computation, it sends back the result to the master process with the function `MPI_Send`. Eventually, when the work queue has been emptied, the master process sends a stop message to the idle processes.
By dynamically assigning the chunks to the processes we aim to reduce the load imbalance. Basically we reproduce the same mechanism OpenMP adopts to distribute the iterations among the threads when the `dynamic` schedule is used.

2.4 Hybrid Implementation

We can combine the OpenMP and MPI libraries to achieve a hybrid implementation of the Mandelbrot set computation. The idea is to use MPI to distribute the work among the processes and then use OpenMP to parallelize the computation of each chunk. Recall that the implementation of both OpenMP and MPI is scheduling the work dynamically, so we expect a good load balance among the threads and the processes. The size of the chunks assigned to each MPI process is not trivial to determine, and a further step of optimization could be to adaptively change the size of the chunks depending on the resolution of the image and the number of processes. This is left as a future work.

3 SCALABILITY

We evaluate the performance of our implementation by assessing its scalability in a strong and a weak sense. For both cases we scale once the number of OMP threads and once the number of MPI processes. Again, for each case we compute the image of the Mandelbrot set with two different resolutions: 512×512 and 4096×4096. We then compare the results with the theoretical expectations.

3.1 Amdahl's Law

The theoretical speedup determined by Amdahl's law is given by

$$S_p = \frac{T_1}{T_p} = \frac{1}{(1-P) + \frac{P}{N}}$$

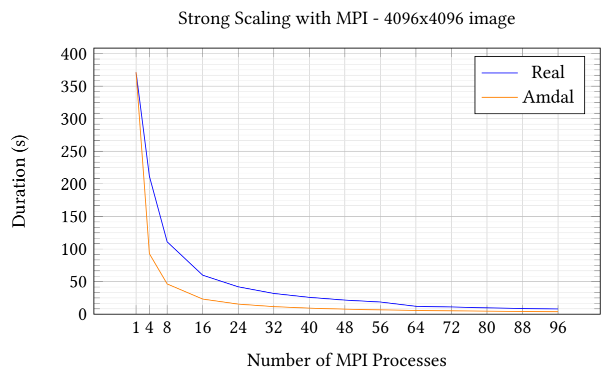
where T_1 is the time to solve the problem with one process, T_p is the time to solve the problem with P processes, N is the number of processors and P is the fraction of the program that can be parallelized. In our case, we will estimate the proportion of parallelizable code by measuring the time to solve the problem with two processors and then solve $P = 2\,(1 - T_2/T_1)$. We will then asses if the theoretical speedup is consistent with the experimental results for all the number of processors.

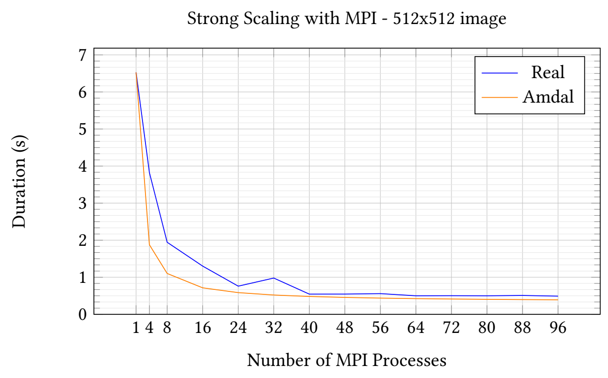
3.2 Strong Scaling

By strong scaling we mean the ability of the program to solve a problem of fixed size in less time by using more computational resources. In our case, the problem size is the resolution of the image, while the computational resources are the number of threads or processes.

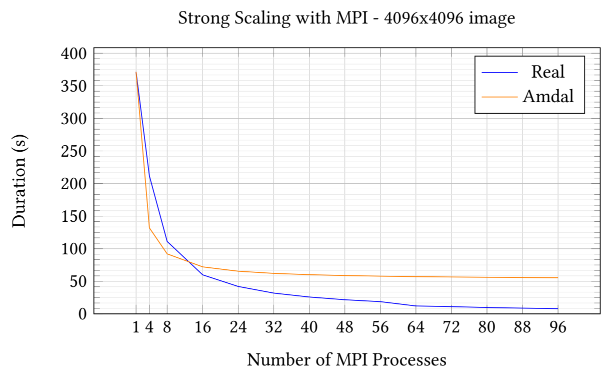
3.2.1 *MPI.*

We begin by showing the results of the strong scaling with MPI compared with the theoretical expectations according to Amdahl's law with $P = 1$. The benchmark has been performed using 4 nodes of the cluster with 24 cores each. The MPI processes have been distributed among the nodes in such a way that each node has the same number of processes. This is done by using the `–map-by ppr:"processes per node":node` option of the `mpiexec` command. The results are shown in the following figures.





As we can see, the experimental results do not match the lower bound given by Amdahl's law. This is due to the fact that the assumption that all the program is parallelizable ($P = 1$) is not true. In fact, estimating the fraction of parallelizable code with the formula $P = 2\,(1 - T_2/T_1)$ we obtain $P_{512} = 0.83$ and $P_{4096} = 0.86$. Considering the estimated P_{4096}, we can compute the theoretical speedup for the 4096×4096 image and again compare it with the experimental results.



This time we observe a different behavior. In the first part of the curve the experimental results lies above the theoretical expectations, while in the second part the experimental results lies below. This discrepancy can be explained by the fact that the Amdahl's law assume that the parameter P is constant and independent from N. Remebering that the estimate has been made with the first two instancies, it is possible that in these first two cases the load imbalance was greater than in the latter ones. This would explain why the experimental results are above the theoretical expectations in the first part of the curve and below in the second part. However, the average computation times per process, shown in the table below, contradict this hypothesis.

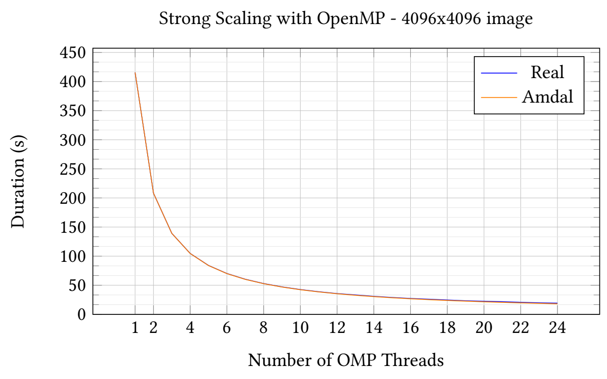
Process	0	1	2	3
Time	211.745397	211.746975	211.745469	211.744659

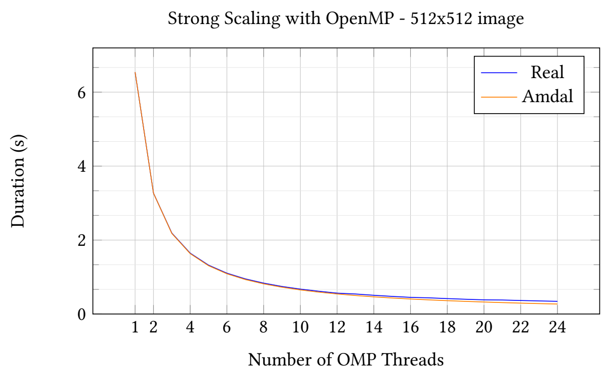
Table 1: Average computation time per process ($N = 4$)

The case of the 512×512 image is analogous.
Another possible explanation could rely in the overhead caused by the initial serial part of the program and the communication between the processes. This overhead could be greater than the time saved by the parallel computation, leading to a worse performance when the number of processes is increased.

3.2.2 *OpenMP.*

We now show the results of the strong scaling with OpenMP compared with the theoretical expectations according to Amdahl's law with $P = 1$.





This time the experimental results are consistent with the theoretical expectations. The estimated P is this time equal to 1 in both cases. After the 12-th core, we can observe a slight increase in the computation time, likely due to the fact that we are now using the cpu of the second socket and this cause a slight increase in the latency of the communication between the two sockets. Despite the slight discrepancy in the second part, the OpenMP implementation scales significantly better than the MPI one. This is due to the fact that the overhead caused by the MPI communication and the initial setup is lower and almost absent.

3.3 Weak Scaling

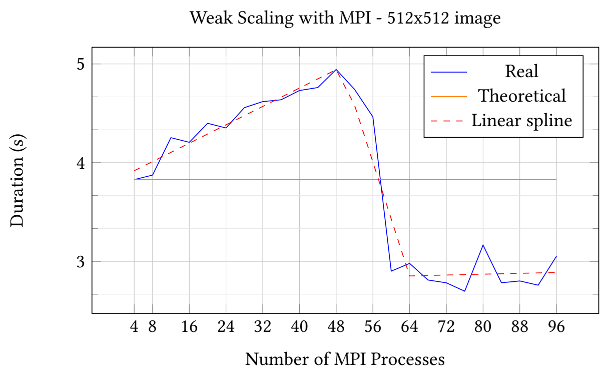
By weak scaling we mean the ability of the program to solve a problem in the same amount of time by using more computational resources and a proportionally larger problem size. The resolution of the image is proportionally increased with the number of threads or processes, the height and the width of the image are multiplied by the square root of the increment of computational resources.

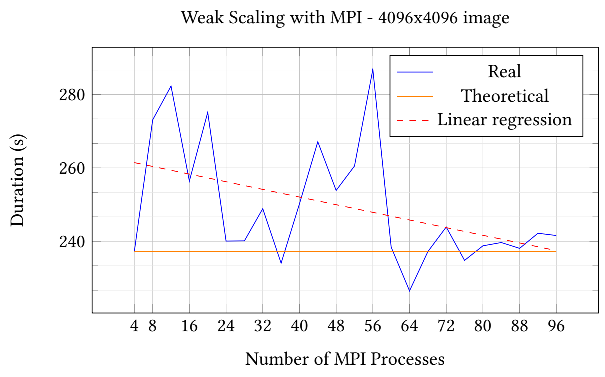
$$W_{t+1} = W_t \cdot \sqrt{\frac{N_{t+1}}{N_t}}$$

$$H_{t+1} = H_t \cdot \sqrt{\frac{N_{t+1}}{N_t}}$$

$$\text{size}_{t+1} = W_{t+1} \cdot W_{t+1} = W_t \cdot H_t \cdot \frac{N_{t+1}}{N_t}$$

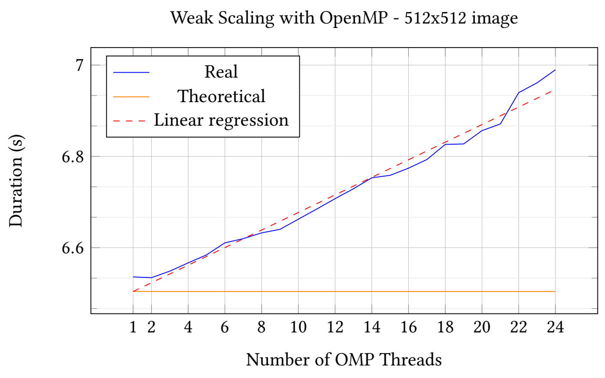
3.3.1 MPI. With the same setup as before, we show the results of the weak scaling with MPI. The results are shown in the following figures.

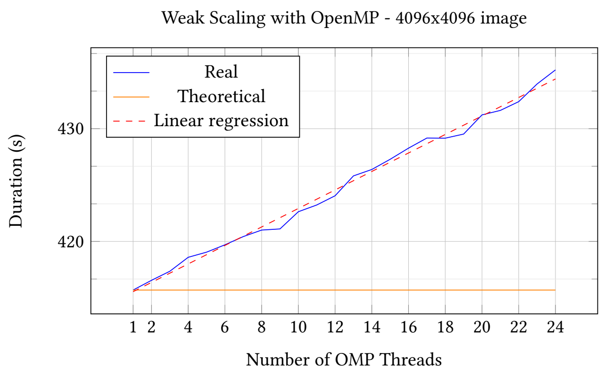




Starting from the second image (4096 × 4096), the real results show a high volatility when compared to the theoretical model. There is no clear trend in the results, despite this the linear regression tells us that the slope (−0.26) is negative, meaning that the computation time is decreasing with the number of processes.
The first image (512 × 512) is more delicate to analyze. The experimental results shows three different behaviors. In the first part the computation time is increasing with slope 0.023, which is consistent enough with the theoretical model. In the second there is a sudden drop in the computation time, with a slope of −0.16, happenging when the total number of processes is 24, therefore every first socket of the each node is filled.Finally, in the last part the computation time follows almost perfectly the theoretical model with a slope of 0.001.

3.3.2 OpenMP. We now show the results of the weak scaling with OpenMP. The results are shown in the following figures.





Theoretically, the experimental data should be consistent with the straight horizontal line indicating that the computation time is constant. In the first case the results are consistent enough with the theoretical model, with a linear increasing trend with coefficient of 0.019. In the second case, the coefficient raises to 0.82, indicating that the computation time is increasing significantly with the number of processors and therefore that the program is not scaling well.

4 CONCLUSIONS

In this work we have implemented the computation of the Mandelbrot set with OpenMP and MPI and we have then evaluated its performance by assessing its scalability in a both a strong and a weak sense. When working with a single node the OpenMP implementation obviously outperforms the MPI one. This is due to the fact that the overhead caused by the MPI communication and the initial setup is absent. Nonetheless, the MPI implementation allows to distribute the computation among multiple nodes, allowing to solve problems of larger size. To get the best of both worlds, we have implemented a hybrid version of the program, which combines the two approaches. The perfect setup would be to use the MPI library to distribute the work among the nodes and then use the OpenMP library to parallelize the computation on each of them.