Hybrid OpenMP + MPI parallel implementation of a Mandelbrot set generator algorithm

Author's Loletti Barbara (mat. SM3800010) High Performance Computing final exam, a.a. 2023/24
University of Trieste, master degree in Data Science and Artificial Intelligence (DSAI)

Abstract: This document reports the procedure used and the results obtained for the **Exercise 2c** of the final exam of High Performance Computing. The assigned task consists in implementing the Mandelbrot set using a hybrid parallel approach by exploiting OpenMP for shared memory computation and MPI scaling for distributed memory computation. In this implementation, the matrix has been divided in rows, distributed among MPI processes in a round-robin fashion. Within each row, the points (represented by pixels in the final image) are then computed in a parallel OpenMP region, which is dynamically allocated. The resulting output is a pgm image. The tests have been performed using 2 AMD Epyc Rome nodes of the Orfeo cluster and show best performance in the solution hereby identified as optimal, which uses MPI for distributed multi-process parallelization among different nodes, and OpenMP for distributed multi-core parallelization within nodes.

1 Introduction

During the development of the algorithm of interest, the first step consists in understanding the problem and devising a serial algorithm in order to provide a first solution. What follows is a betterment of such implementation, creating a distributed and parallelized context, meaning being able to use multi-processing and multi-threading in a hybrid configuration. The resulting code can be run either serially, with multi-processes with MPI run and multi-thread with OpenMP

1.1 The problem

The Mandelbrot set is on its core a mathematical computation in the complex plane which has to be performed on a two-dimensional matrix M. The goal here is to iterate the complex function

$$f_c(z) = z^2 + c \tag{1}$$

for a complex point $c = x + iy$ and starting form the complex value of $z = 0$. The result will be a series of values $z_i = f_c(z_{i-1})$. In this sense, the Mandelbrot set is defined as the set of complex points c for which the function $f_c(z)$ does not diverge to infinity. The points are all contained into a region that can be specified as a square through the coordinates passed as input parameters x_R, y_R, x_L, y_L). In order to highlight the fractal nature in a 2D space of this problem, each point can be mapped to the iteration count needed to establish that the function diverges for each point c. The criterion used to carry out such discrimination is

$$|z| \geq 2.0 \tag{2}$$

The total iteration count for each point is bounded, for the sake of a computer-run implementation, to a precise I_{max} parameter, which represents the maximum value that can be stored using the datatype chosen to represent the output matrix. If the recursive function for a given point c surpasses this maximum number of iterations without satisfying the divergence criterion, such point is assumed to belong to the Mandelbrot set with a value of 0 being associated to c in the output matrix. On the other hand, if at some point the criterion for divergence is met during the recursion, the value n of the current iteration is associated to c in the output matrix. The output matrix therefore will contain values in the $[0; I_{max}]$ range, which can be converted into pixels in a straightforward way by mapping each value into a grayscale range. A simple image format, such as the portable pixmap format or PPM, is then used to encapsulate the so obtained data in order for it to be represented as a grayscale image. The resolution of the image, i.e. the size of the computed matrix, is given by the n_x and n_y parameters.

1.2 Hybrid parallelization

The solution proposed to this problem relies on hybrid parallelization. The simplest approach, in the context of operations on a matrix, is to choose a strategy in order to distribute the rows and columns computation to the various available computational resources. In this sense, a sensible approach is to parallelize across multiple processes, i.e. in distributed memory, in case we're working with physically detached hardware, as in the case of different nodes or memory regions with lower affinity. On the other hand, we can also use multiple threads if we're dealing with affine computational resources, as in the case of cores inside the same processor.

2 Parallelization

In order to parallelize the computational load, a choice has been made to distribute the computation of different rows across multiple processor sockets and parallelize the calculation within each row by splitting it across cores in each socket. What happens in practice is that an MPI process is assigned to each processor socket, within which the calculation is performed by splitting it within all the available cores with OpenMP threads. The decision to choose sockets over nodes is dictated by the will to better investigate the scaling of MPI with a larger number of available resources.

2.1 Hardware

The nodes of the Orfeo computing cluster that have been chosen in order to test the algorithm in different conditions are 2 AMD Epyc Rome nodes.

2.2 MPI parallelization

Computation of the Mandelbrot set, as already stated, is intrinsically a parallel task, as independent calculations are carried out for each element of the considered matrix. The solution proposed is a C implementation based on a recursive function which computes all the values for a given point and is applied to all the points of the considered matrix. In this case, as said before, multi-processing is obtained by spawning an MPI process for each and every available CPU socket. In order for the computation of different rows to be distributed, a round-robin approach is used.

Slaves-only parallelization The first approach to have been tested consists in having a process (the *master process*) which serves as a coordinator to all the others processes (*slave processes*). The *master* awaits for the availability of the *slaves* in order to send them a row to compute, being sent back the computed data, store them and repeat the process till all the rows have been computed. The master itself doesn't perform any sort of computational work.

```
while (recvd_rows < ny) {
  // wait for any rank to be ready to receive
       a new task (row to be computed)
  MPI_Recv(row, nx, MPI_UNSIGNED_SHORT,
      MPI_ANY_SOURCE, TAG_TASK_ROW_RESULT,
      MPI_COMM_WORLD, &status);

  nrow = assigned_rows[status.MPI_SOURCE];
  if (nrow != -1) {
    memcpy(M + nrow * nx, row, nx * sizeof(
        mb_t));
    #ifdef VIZ
    viz_render(M, nx, ny, Imax);
    #endif
    recvd_rows++;
  }

  // send the ready rank some work to do, i.e
      . the next available row to be computed
  if (next_row < ny) {
    llog(4, "assigning row %d to rank %d\n",
        next_row, status.MPI_SOURCE);
    MPI_Send(&next_row, 1, MPI_INT, status.
        MPI_SOURCE, TAG_TASK_ROW,
        MPI_COMM_WORLD);
    assigned_rows[status.MPI_SOURCE] =
        next_row;
    next_row++;
  }
}
```

As reported in the code, the master keeps track of the rank of the process to which has been assigned which row. This is done in order to be able to reorder the returning results from the workers. The **MPI_Recv** implies for the master to wait until any worker sends back some results. For brevity, the code run by the worker processes has been omitted.

Master-slaves parallelization An improvement of the first version is to modify the master's code in order to allow it to carry out some useful computation while waiting for the slave processes to send back their results. This basically means that the master process becomes one of the workers.

```
while (recvd_rows < ny) {
    // wait for any rank to be ready
    //to receive a new task (row to be
        computed)
    MPI_Irecv(row, nx, MPI_UNSIGNED_SHORT,
        MPI_ANY_SOURCE, TAG_TASK_ROW_RESULT,
        MPI_COMM_WORLD, &row_result_recv);

    nrow = assigned_rows[status.MPI_SOURCE];
    if (nrow != -1) {
        memcpy(M + nrow * nx, row, nx *
            sizeof(mb_t));
        #ifdef VIZ
        viz_render(M, nx, ny, Imax);
        #endif
        recvd_rows++;
    }

    // master working
    MPI_Test(&row_result_recv, &
        row_result_recvd, &status);

    while(!row_result_recv && next_row < ny){
        nrow = next_row;
        row = _mandelbrot_matrix_row(
            requested_row,
            nx,
            ny,
            xL,
            yL,
            xR,
            yR,
            Imax);
        memcpy(M + nrow * nx, row, nx *
            sizeof(mb_t));
        next_row++;

        MPI_Test(&row_result_recv, &
            row_result_recvd, &status);
    }
    [...]
}
```

In this new version, the **MPI_Recv** has been replaced by an asynchronous **MPI_Irecv**, which is tested repeatedly for completion. While the request has not been fulfilled (meaning the workers haven't produced any result yet), the master takes on the available rows (one at the time) and perform the corresponding computation. Given the better results that have been obtained while testing the latter version, it has been selected for further testing and benchmarking the overall performance of the implementation

2.3 OpenMP parallelization

In addition to MPI, OpenMP is utilized to further parallelize the computation within each node by distributing the workload among the cores available in a single processor. This approach leverages the shared memory architecture, allowing multiple threads to operate concurrently on different portions of the data. Specifically, the OpenMP directive **#pragma omp parallel for schedule(dynamic)** is employed to parallelize the computation of the Mandelbrot set. The code is reported below

```
#pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < nx; i++) {
    x = xL + i * dx;
    y = yL + r * dy;
    c = x + I * y;
    matrix[i] = mandelbrot_func(0 * I, c, 0, Imax
        );
  }
```

```
}
```

This directive instructs the compiler to create a parallel region where the for loop is executed by multiple threads. The dynamic scheduling ensures that each thread is assigned a chunk of iterations dynamically at runtime, which helps in load balancing especially when the computation time varies significantly across different parts of the matrix. In this implementation, each thread calculates the Mandelbrot value for different points in the matrix, identified by their coordinates *(i,j)*, and stores the result in the matrix M. This parallelization strategy enhances the performance by fully utilizing the multi-core capabilities of the nodes.

3 Testing phase

A testing phase has been performed in order to study the scalability of this hybrid implementation. Given the hybrid structure of the code, both the MPI parallelization and the OpenMP parallelization have been measured independently and specific data about the scalability of each of them has been obtained. The following fixed parameter were used in all experimental cases: matrix dimensions (hence image resolution) 1024x1024, the coordinates outlying the already analyzed complex plane regions with the following value, $x_R = 3, x_L = -3, y_R = 3, y_L = -3$. Also the maximum value for the iteration count has been set to $I_{\max} = 65535$, which is the maximum that can be held in an `unsigned short` variable. All the graphs that are reported use one or more theoretical speedups, obtained with a trivial application of the Amdahl's law: the time obtained for the least number of processing units is put into proportion with all the other cases.

3.1 OpenMP scaling

Data about OpenMP scaling has been obtained by setting the number of MPI processes to 1. This has been performed in practice by allocating a single unboud MPI process with the command `mpirun -np 1 --bind-to none` onto a fully reserved node and varying the `OMP_NUM_THREADS`, which controls the number of cores used by OpenMP to spawn threads for parallel regions. Based on such approach, a single node was used raging the core count form 1 to 24, doubling it at each iteration. The data obtained are using

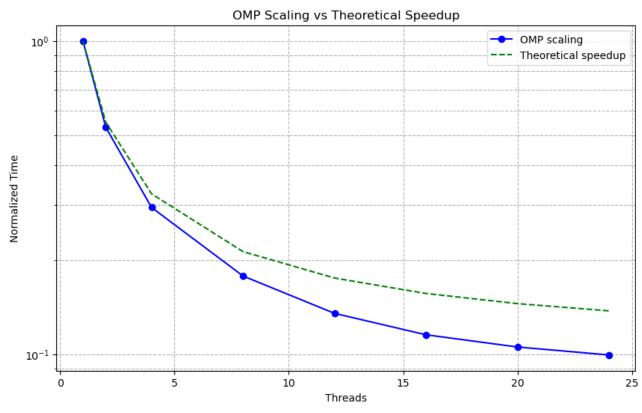

Figure 1: *OMP scalability using a single MPI process on one THIN node and 24 cores for each socket*

this approach shown in the table below The graph illustrates

Threads	Time (s)
1	63.56
2	33.67
4	18.69
8	11.29
12	8.58
16	7.33
20	6.70
24	6.32

Table 1: *OMP scaling performance data.*

the scaling behavior of the OpenMP (OMP) implementation compared to the theoretical speedup. As the number of threads increases, the normalized time decreases, indicating better performance.

3.2 MPI scaling

The MPI scaling test was conducted using a varying number of processes on a cluster with two nodes, each hosting 24 MPI tasks. The script used loaded the necessary OpenMPI module, compiled the program, and ran the Mandelbrot set computation with different process counts ranging from 2 to 48. The results, shown both in figure 2 and in the table below, indicate a significant reduction in computation time as the number of processes increases, demonstrating excellent parallel scalability. Remarkably, the actual performance surpasses the theoretical speedup, indicating an exceptionally efficient implementation with minimal overhead and excellent load balancing. The graph illustrates the MPI

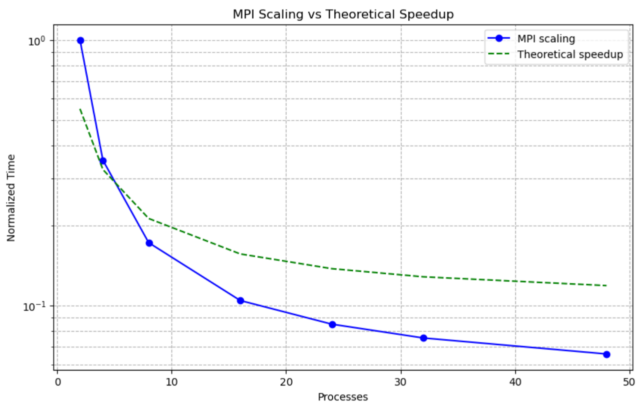

Figure 2: *MPI scalability using 2 THIN nodes*

Processes	Time (s)
2	63.55
4	22.38
8	10.95
16	6.63
24	5.40
32	4.79
48	4.17

Table 2: *MPI scaling performance data.*

scaling performance compared to the theoretical speedup. As the number of processes increases, the normalized time decreases significantly, indicating improved performance and efficiency of the parallel implementation. Impressively, the actual performance line falls below the theoretical speedup line, highlighting the effectiveness of the MPI implementation in minimizing communication overhead and achieving near-optimal load distribution. This analysis underscores the robustness of MPI in parallelizing the Mandelbrot computation, showcasing the ability to achieve better-than-expected performance gains.

Strong MPI scaling The MPI strong scaling experiment was conducted by keeping the problem size constant at 1024x1024 and varying the number of processes. The goal of strong scaling is to observe how the computation time decreases as the number of processes increases for a fixed total workload. The results, shown in the table below, indicate a substantial reduction in computation time as the number of processes increases, demonstrating effective parallelization. However, beyond a certain point, the reduction in time starts to diminish, which is indicative of overhead and communication costs becoming significant relative to the computation. This behavior is expected as more processes are used, and it highlights the practical limits of strong scaling efficiency. The comparison with the theoretical speedup time, based on

Processes	Time (s)
2	63.69
4	14.20
8	12.23
16	10.04
24	10.25
32	7.30
48	6.91

Table 3: *MPI strong scaling performance data.*

Amdahl's Law, shows that while the actual performance improves significantly with an increasing number of processes, it does not fully align with the theoretical predictions. The actual computation time flattens out after a certain number of processes, unlike the theoretical time, which continues to decrease. This discrepancy is primarily due to the overhead and communication costs associated with managing a larger number of processes, which are not accounted for in the theoretical model. Despite this, the strong scaling results are impressive and demonstrate the efficiency of the parallel implementation.

Weak MPI scaling The MPI weak scaling experiment was conducted by increasing the problem size proportionally with the number of processes. Specifically, the problem size was scaled up such that each process would handle a similar amount of workload as the number of processes increased. The results, shown in the table below, reveal that the computation time generally increases with the number of processes, which is indicative of the increased total workload. However, the significant variability and higher times at larger process counts highlight potential inefficiencies in

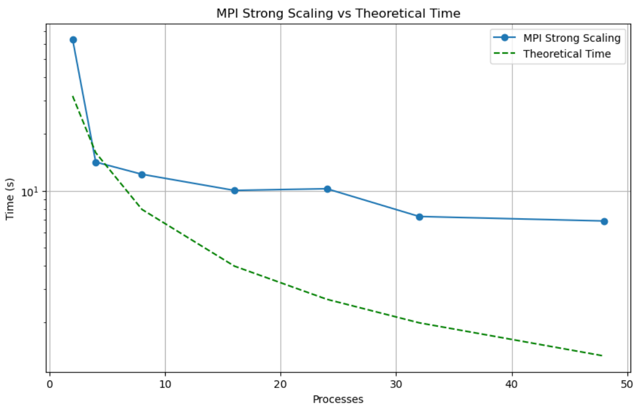

Figure 3: *MPI scalability using 2 THIN nodes*

managing and distributing larger workloads, as well as the increased communication overhead that arises with more extensive parallelism. A notable observation in the weak

Processes	Time (s)
2	244.64
4	77.54
8	217.20
16	719.14
24	1573.79
32	1393.41
48	6.25

Table 4: *MPI weak scaling performance data.*

scaling results is the significant drop in computation time when using 48 processes. This unexpected drop could be due to several factors, including improved load balancing, better utilization of network resources, or even specific optimizations in the underlying hardware or MPI implementation that become effective at this scale. It is also possible that the problem size at 48 processes hits a sweet spot where the overheads are minimized, and the processes can efficiently share the workload without significant contention. Further investigation would be required to pinpoint the exact cause of this performance improvement.

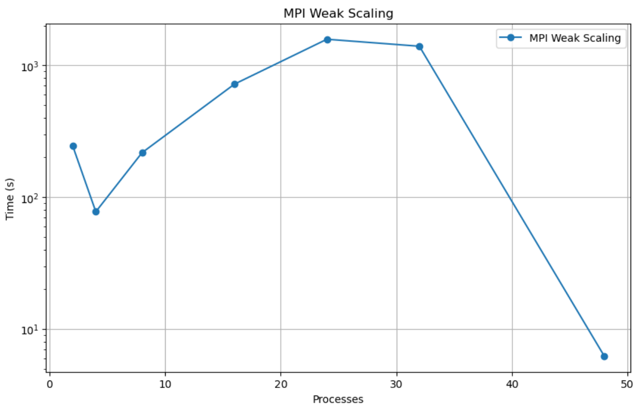

Figure 4: *MPI scalability using 2 THIN nodes*

4 Conclusion

In conclusion, the hybrid parallel implementation of the Mandelbrot set using OpenMP and MPI has demonstrated significant performance improvements and scalability on a high-performance computing cluster. The OpenMP scaling tests showed a substantial decrease in computation time as the number of threads increased, highlighting the effectiveness of shared memory parallelization within a single node. Similarly, the MPI scaling tests, both strong and weak, revealed excellent parallel scalability across multiple nodes, with actual performance often surpassing theoretical speedup predictions. This indicates efficient load balancing and minimal overhead, underscoring the robustness of the implementation. Overall, the combination of MPI for distributed memory parallelization and OpenMP for shared memory parallelization has proven to be an optimal solution for large-scale Mandelbrot set computations, leveraging the full computational power of the Orfeo cluster.