Devising and benchmarking a hybrid MPI + OpenMP parallel implementation of a Mandelbrot set generator algorithm

Andrea Esposito (mat. SM3600005)
University of Trieste - Scientific and Data Intensive Computing (SDIC)

High Performance Computing Final 2023/24

Abstract: This document reports procedures and results for **Exercise 2c** of the final assignment of the High Performance Computing course, implemented with a **hybrid approach**. The goal of the assignment is to implement the algorithm for generating the Mandelbrot set using a hybrid parallel approach, namely exploiting both MPI scaling for distributed memory computation and OpenMP for shared memory computation. The presented implementation divides the matrix to be computed in rows, distributing them among MPI processes in a round-robin fashion. Within each row, points (pixels in the final image) are computed in a parallel OpenMP region, dynamically allocated. The output is a pgm image file. Tests have been performed on AMD Epyc Rome nodes of the ORFEO cluster and show best performances 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

The first step performed in the development of the parallel algorithm of interest is understanding the task and devising a serial algorithm to solve it. What follows is the adaptation of such implementation to a distributed and parallelized context, i.e. using multi-processing e multi-threading, in a hybrid configuration. The resulting code can therefore be run either serially, multi-processed with MPI and multi-threaded with OpenMP[1].

1.1 Problem

Computing the Mandelbrot set involves mathematical computation in the complex plane to be performed recursively on the considered two-dimensional matrix. The goal is to determine whether the function

$$f(z) = z^2 + c$$

does not diverge to infinity for each point c in the analyzed region. The region of interest is specified as a square through the coordinates passed as input parameters x_L, y_L, x_R, y_R. A simple visualization highlighting the fractal nature of this set in a 2D space can be obtained by mapping each point to the (bounded) iteration count needed to establish that the function diverges for each point c. The criterion for such discrimination is defined as

$$|z| >= 2.0$$

The total iteration count for each point is bounded, for the sake of a computer-run implementation, to the maximum value I_{max} that can be stored using the datatype chosen for the output matrix. As the recursive function for a given point c surpasses the maximum number of iterations without the criterion for divergence being satisfied, such point is assumed to belong to the Mandelbrot set and the value 0 is associated to c in the output matrix. Otherwise, as the criterion is first met during recursion, the value for the current iteration n is associated to point c in the output matrix.

The output matrix therefore contains values in the $[0, I_{max}]$ range, which can be made into image pixels in a straightforward way by mapping each value into a grayscale range. A simple image format, portable pixmap format or PPM, is then used to encapsulate the obtained data, in order for it to be displayed as a grayscale image. The resolution of such image, i.e. the size of the computed matrix, is given as parameters n_x and n_y to the program.

1.2 Hybrid parallelization

The simplest approach to hybrid parallelization in the context of operations on a matrix is choosing a different strategy for distributing rows and columns across the available computing resources. A sensible approach is to parallelize across multiple processes, i.e. in distributed memory, when dealing with physically detached hardware, e.g. different nodes or memory regions with lower affinity. On the other hand, use multiple threads (or nonetheless a shared memory paradigm) when dealing with affine computational resources, e.g. cores on the same processor.

2 Parallelization

For the sake of this hybrid implementation, it was chosen to distribute the computation of different rows across multiple processor sockets and parallelize calculations within each row across cores in each socket. In practice, an MPI process is spawned for each processor socket, within which the calculation is split across all available cores with OpenMP threads. This choice was dictated by the impossibility to exploit multithreading on the machines that were used for testing. Sockets were chosen over nodes to better investigate MPI scaling with a larger number of available computational units.

2.1 Hardware

Two different types of nodes of the ORFEO computing cluster were used in testing in different conditions: *Epyc* and *Thin* nodes. A description of Epyc nodes is presented in the solution of Assignment 1. Thin nodes bear a simpler CPU

[1] As later described, for the sake of this implementation, multi-threading was not always exploitable on the machines on which testing was performed. Multi-threading in this context therefore refers to OpenMP terminology for computational units employed in parallel regions for shared memory computing.

configuration, with two sockets providing 12 cores each, for a total of 24 physical per node. Multithreading capabilities is disabled on both types of nodes.

2.2 MPI parallelization

Computing the Mandelbrot set is an intrinsically parallel task, as an independent calculation is performed for each element of the considered matrix. The presented C implementation is based on a recursive function which computes the value for a given point, applied to all points of the considered matrix. Multi-processing is achieved by spawning an MPI process for each available CPU socket. In order to distribute the computation of different rows, a round-robin approach is used.

Slave-only workers In the first tested approach, the first process (the *master* process) serves as coordinator for all other processes (the *slave* processes), waiting for their availability, sending them a row to compute, receiving and storing the computed data back and repeating until all rows have been processed and received back. No computational work is carried out by the master process itself. What follows is a snippet of a portion of code run by the master process.

```
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)
    );
    recvd_rows++;
  }

  // send the ready rank some work to do,
  // i.e. the next available row to be computed
  if (next_row < ny) {
    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 shown, the master keeps track of which rank was assigned which row, in order to be able to reorder them upon arrival of the computation results from the workers. The blocking `MPI_Recv` call implies the wait of the master process until any worker returns some result. Code running on the worker processes is omitted for brevity and available in the project repository.

Master+Slaves workers An improvement upon the first version is to tweak the master code to allow it to carry out useful work while waiting for slave processes to return their computation results. In other words, make it so 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
  );

  // master doing work!
  MPI_Test(
    &row_result_recv,
    &row_result_recvd,
    &status
  );
  while (!row_result_recvd && next_row < ny) {
    nrow = next_row;
    row = _mandelbrot_matrix_row(
      next_row,
      nx,
      ny,
      xL,
      yL,
      xR,
      yR,
      Imax
    );
    memcpy(M + nrow * nx, row, nx * sizeof(mb_t));
    next_row++;
    recvd_rows++;

    MPI_Test(
      &row_result_recv,
      &row_result_recvd,
      &status
    );
  }

  // [...]
}
```

In this modified version the blocking `MPI_Recv` call is replaced by an asynchronous `MPI_Irecv`, which is then tested repeatedly for completion. While the request has not been fulfilled (and thus while no result is received by any worker), the master process takes on the available rows (one at a time) and performs the corresponding computation. Given the good results obtained when testing this latter version, it was selected for further testing and for benchmarking the overall performance of the implementation.

2.3 OpenMP parallelization

As previously described, hybrid parallelization is exploited to further distribute the computation over each row, each of which gets assigned to a process running on a socket, among the available cores on such socket, using OpenMP.

The most straightforward form of OpenMP parallelization in this scenario is the `for` work-sharing construct. With each process computing a row as a loop over its elements, applying the appropriate `pragma` directive to the loop is sufficient. Further tweaking of the parameters is possible, through preprocessor directives and environment variables, in order to change thread binding, work scheduling and number of spawned threads.

```
#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
  );
}
```

The chosen scheduling policy for the parallel region is *dynamic*, as the computation of each element bears an unpredictable load with respect to the others. Using such policy, it is left to OpenMP to distribute the work among available threads, without a predetermined subdivision in chunks.

3 Testing

Extensive testing of the final hybrid implementation has been carried out in order to establish its scalability. Given the hybrid approach, both the MPI parallelization and the OpenMP parallelization have been measured independently, therefore obtaining MPI scalability data and OpenMP scalability data. All graphs include one or more theoretical speedups, obtained by trivially applying Amdahl's law: the time obtained for the least number of processing units is put into proportion with all other cases. Except where noted, the following fixed parameters were used in all experimented cases: matrix dimensions (image resolution) of 4096x4096, coordinates outlining the analyzed complex plane region: $x_L = -3, y_L = -3, x_R = 3 y_R = 3$, maximum iteration count $I_{max} = 65535$, i.e. the maximum value that can be held in a `unsigned short` variable.

3.1 Strong MPI scaling

Data about MPI strong scaling has been obtained by running the program with a fixed number of OpenMP threads by means of the `OMP_NUM_THREADS` environment variable set to 1, practically disabling it. By mapping MPI processes to CPU sockets of 2 Epyc nodes, with the topology described in 2.1 (64 places i.e. cores each), the following data has been gathered.

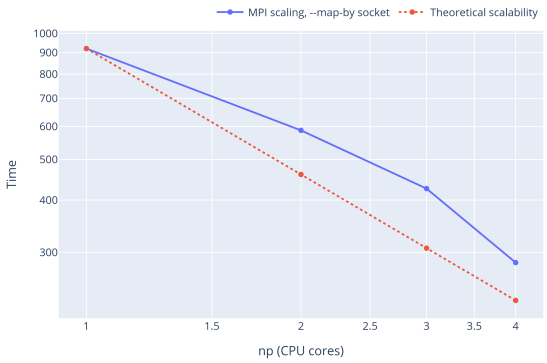

Figure 1: *MPI scalability with 4 sockets over 2 Epyc nodes*

It can be easily noted that the timings rapidly diverge from the theoretical speedup, proceeding roughly in parallel to it. No definitive conclusion can be drawn because of the limited amount the data that could be gathered, but it can be assumed that the additional time spent per socket accounts for the overhead induced by message passing.

3.2 Hybrid scaling

Similar testing has been carried out varying both the number of sockets (MPI) and the number of cores (OpenMP), in order to benchmark the taken hybrid approach. With the same conditions outlined for the previous experiments, the following results have been obtained.

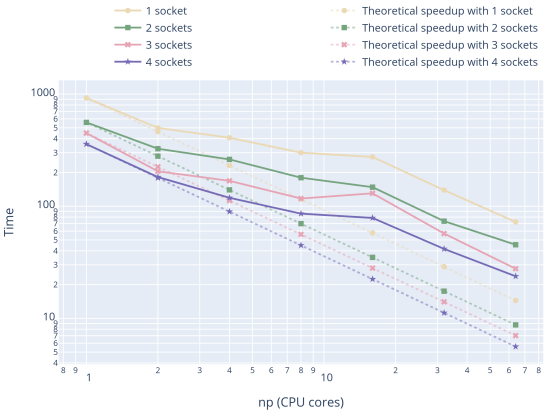

Figure 2: *Hybrid scalability with MPI processes on 4 sockets over 2 Epyc nodes and OpenMP threads on 64 cores for each socket.*

Experimental data follows expectations built up from previous results: timings for different socket counts stack

on top of each other in decreasing order (more sockets, less time spent) and all follow a similar shape as the core count varies.

3.3 OpenMP scaling

Data about OpenMP scaling has been obtained, conversely to MPI scaling, by allocating a single unbound MPI process (`--np 1 --map-by node --bind-to none`) onto a fully reserved node and varying `OMP_NUM_THREADS` controlling the number of places (in this case, cores) used by OpenMP to spawn threads for parallel regions. With such a shared memory approach, a single node was used, ranging the core count from 128 to 2, halving at every iteration.

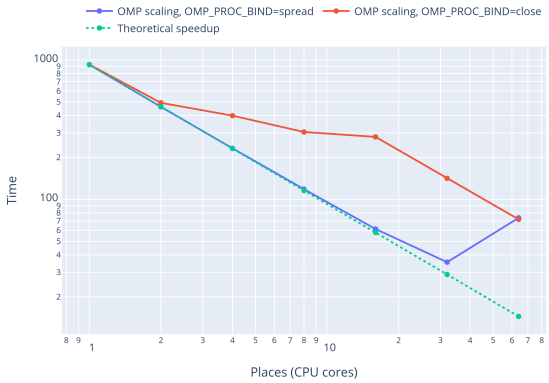

Figure 3: *OpenMP scalability with a single MPI process on 1 Epyc node and OpenMP threads on 128 cores for each socket, with OMP_PROC_BIND set to spread and close.*

Two measurements were performed, varying the `OMP_PROC_BIND` to change how OpenMP schedules the spawned threads: evenly spread among the available resources when `OMP_PROC_BIND=spread` or close together when `OMP_PROC_BIND=close`. A noticeable difference is visible in the resulting plot, as execution times when threads are close together tends to quickly diverge from the theoretical speedup, allegedly because of memory contention. On the other hand, executing the same task with threads being spawned spread apart, timings follow the theoretical speedup for the most part, only diverging at the end, when execution is carried out by the maximum number of available cores, in this case 128.

For the sake of verification, a similar run was carried out in the same conditions, only varying the maximum number of iterations I_{max} from 65535 to 8192.

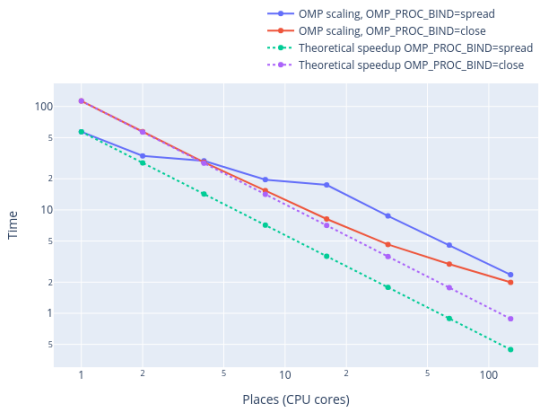

Figure 4: *OpenMP scalability with a single MPI process on 1 Epyc node and OpenMP threads on 128 cores for each socket, with OMP_PROC_BIND set to spread and close.*

Execution times are expectedly sensibly lower, but the trend highlighted in the previous experiment, at least for `OMP_PROC_BIND=spread`, is no longer present.

Memory improvements Assuming a memory contention or false sharing issue, further testing was performed applying the *touch by all* policy, namely parallelizing matrix initialization across threads with the same scheduling applied when performing computations upon the same matrix, so as to locate the portions of such matrix to be used by each thread as close to it in memory as possible. Although easily applicable for the nature of the utilized parallelization techniques, namely the `parallel for` clause on each row, the need to use the same scheduling during initialization and computation steps amounts to using static scheduling with `schedule(static)`, which hinders performance quite noticeably. Nonetheless, repeating measurements with static scheduling, with and without the distributed initialization (*touch by all*), did not highlight any remarkable difference in execution time.

3.4 Weak MPI scaling

Data about weak scaling has been obtained by applying Gustafson's law, i.e. comparing different runs in which the size of the problem and the available resources scale simultaneously, in order for every processor to be tasked with the same load at each run. The goal of performing weak scaling is figuring out the parallel fraction of the code, namely the part of the code that scales as more computational resources are available, or, simply speaking, *how much more work* can be performed in the same amount of time. Ideally, i.e. in the condition of a completely parallel code, different runs with problem size and available resources scaling together accordingly should exhibit roughly the same execution time.

Figure 5: *Weak scalability with OpenMP threads spawned on cores over 1 Epyc node (128 cores).*

As shown in the results, the implemented solution presents a pretty poor weak scaling performance. Although the amount of work per process is kept fairly constant across the runs, the communication (for MPI) or synchronization (for OpenMP, as in the presented case) overhead amounts to a substantial loss of performance throughout. What can be seen from $nx = 512$ is the expected trend of execution times, with only a slight upwards tendency.

4 Conclusions

A feasible implementation of an algorithm to compute and visualize the Mandelbrot set was presented and tested, showing how it is possible to adapt it in order to achieve hybrid parallelization. Extensive testing was carried out the ORFEO cluster in order to establish the parallel performance of such algorithm, highlighting its decent (strong) MPI scalability, allegedly hindered by communication overhead, and better OpenMP scalability, with some interesting patterns in execution times, to be possibly attributed to memory contention or false sharing. More research upon the topic can be targeted at reducing communication time, e.g. devising an effective static partitioning of the task to be done, or at containing memory issues when operating in shared memory, e.g. studying in more depth the topology of the tested machines and experimenting with relevant OpenMP data directives. A different direction in further development of the presented solution would be to use hybrid MPI, which offers both distributed and shared memory features, in order to study how it compares with the presented solutions.