Revista Innovación y Software
Vol. 1, No. 2, Mes Septiembre-Febrero, 2020
ISSN: 2708-0935
Pág. 12-26
https://revistas.ulasalle.edu.pe/innosoft

Tipo de artículo: Artículos originales
Temática: Programación paralela y distribuida
Recibido: 22/09/2020 | Aceptado: 23/09/2020 | Publicado: 30/09/2020

MPI vs OpenMP: Un caso de estudio sobre la generación del conjunto de Mandelbrot

MPI vs OpenMP: A case study on parallel generation of Mandelbrot set

Ernesto Soto Gómez [1][0000-0001-6521-2221]*

[1] Universidad de las Ciencias Informáticas. Carretera a San Antonio de los Baños, Km. 2 ½. Torrens, La Lisa, La Habana, Cuba. esoto@uci.cu

* Autor para correspondencia: esoto@uci.cu

Resumen
Algunas de las herramientas más populares hoy en día para la programación paralela son Interfaz de Paso de Mensajes y Multiprocesamiento Abierto. Es de interés comparar estas herramientas en la resolución de los mismos tipos de problemas, debido a la utilización de diferentes enfoques en la comunicación entre tareas. Este trabajo tiene como objetivo contribuir a este empeño al ejecutar pruebas en una arquitectura de memoria compartida y centralizada en el caso de problemas con una solución completamente paralela. El caso de estudio seleccionado fue la computación paralela del conjunto de Mandelbrot. Las pruebas se realizaron para diferentes límites de iteración, cantidad de procesadores y variantes de implementación en C++. Los resultados muestran un mejor desempeño en el caso de Multiprocesamiento Abierto.

Palabras clave: C++, computación paralela, conjunto de Mandelbrot, MPI, OpenMP.

Abstract
Nowadays, some of the most popular tools for parallel programming are Message Passing Interface and Open Multi-Processing. It is of interest to compare these tools in solving the same kind of problems, because of the use of different approaches to inter-task communication. This work attempts to contribute to this goal by running trials in a centralized shared memory architecture in the case of problems with an entirely parallel solution. The selected case study was the parallel computation of Mandelbrot set. Trials were conducted for different iteration limits, processors amount, and C++ implementation variants. The results show better performance in the case of Open Multi-Processing.

Keywords: *C++, Mandelbrot set, MPI, OpenMP, parallel computing.*

Introduction

There are diverse tools for parallel programming. Some of the most popular nowadays are Message Passing Interface (MPI)[1] and Open Multi-Processing (OpenMP)[2]. Both tools are essentially dissimilar because of the use of different approaches to inter-task communication: OpenMP uses shared-memory (tasks are realized by using threads in the same operating system process) [1,2] but MPI uses message-passing (tasks are realized by using a different operating system processes) [3,4]. For this reason, it is of interest to compare these tools in solving the same kind of problems. That is, which is the best in computing the same kind of solution for the same kind of problems taking into account that Do the concerned tools use different inter-task communication mechanisms? This article attempts to contribute to the answer of this question in a centralized shared memory architecture [5] in the case of problems with an entirely parallel solution, that is, a solution with the absolute absence of the need for synchronization –except for the gathering of the partial solutions from several subtasks in order to construct one final solution–.

To accomplish this goal, the parallel generation of the Mandelbrot set has been chosen as an example. This case has been studied in the parallel computing context, usually as a didactic example [1,6,7] because it can be generated from a simple mathematical expression. Also, the Mandelbrot set is a fractal: a figure that possesses a detailed structure in a wide range of scales. Fractal geometrical relations are found in several natural structures, thereby fractals are of great interest to science [8]. This last point adds to the motivation of the study of this example.

The parallel computing of the Mandelbrot set has been already studied in the case of MPI and OpenMP independently from each other [1, 9, 10]. The current work makes comparisons between the straightforward sequential implementation and corresponding parallel versions implemented in MPI and OpenMP with different schedule strategies. C++ has been used as the programming language and the comparisons were made for different iteration limits and number of processors. All generated data as well as all used code may be found in https://github.com/EStog/mandelbrotc-/tree/0.1.

The current document is structured in the following manner. First, the fundamental theoretical elements, the proposed sequential algorithm, and the corresponding parallel versions are exposed. Second, the characteristics of the experiment and the obtained results are described. Last, final remarks are made.

Methods and materials

[1] https://www.open-mpi.org/
[2] https://www.openmp.org/

Revista Innovación y Software
Vol. 1, No. 2, Mes Septiembre-Febrero, 2020
ISSN: 2708-0935
Pág. 12-26
https://revistas.ulasalle.edu.pe/innosoft

Sequential implementation

The Mandelbrot set is the set of all $c \in \mathbb{C}$ for which the recurrence relation (Equation 1):

$$z_n = z_{n-1}^2 + c \tag{1}$$

does not diverge with $z_n \in \mathbb{C}$ and $z_0 = 0$.

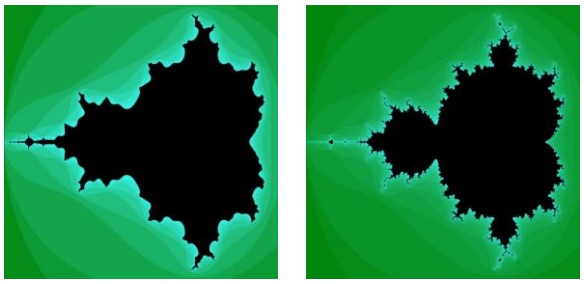
It is known [6,7] that such sequence does not diverge when (Equation 2):

$$|z_n| \leq 2 \tag{2}$$

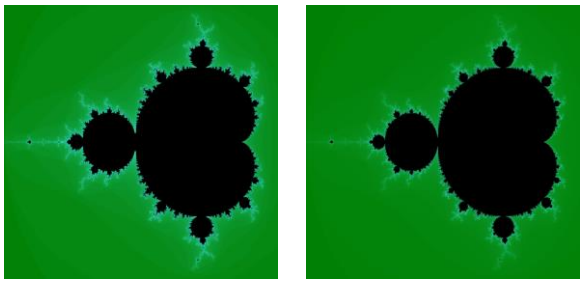
for all $z_n \in \mathbb{C}$ where[3] (Equation 3):

$$|z_n| = \sqrt{\Re(z_n)^2 + \Im(z_n)^2} \tag{3}$$

As a way of visualization, the values of *c* that are members of the Mandelbrot set may be drawn in the complex plane. Figure 1 shows images of the Mandelbrot set. The images were generated by using the C++ solution developed for this research.



(a) iteration limit = 10 (b) iteration limit = 20



(c) iteration limit = 40 (d) iteration limit = 80

Figure 1. Representation of the Mandelbrot set in the complex plane.

[3] $\Re(z)$ and $=\Im(z)$ stands for real and imaginary parts, respectively.

A straightforward algorithm that gives an approximation of the Mandelbrot set is to move along a subset of a discrete version of the domain of c and verify that z_n does not diverge by using (Equation 2). The generation of the sequence determined by (Equation 1) is made while a given iteration limit is not exceeded [6,7]. A sequential C++ implementation of the mentioned algorithm is given in (Listing 1).

Listing 1. Sequential C++ implementation of the straightforward algorithm to approximately compute the Mandelbrot set.

```
1    void compute_mandelbrot_subset(int* result, int iter_limit, int x_resolution,
2                                   int y_resolution, double x_begin, double y_begin) {
3        double x_step = (x_end-x_begin) / (x_resolution-1);
4        double y_step = (y_end-y_begin) / (y_resolution-1);
5        int i, j;
6        complex<double> c, z;
7        for (i = 0; i < x_resolution * y_resolution; i++) {
8            c = complex<double>(x_begin + (i % x_resolution) * x_step,
9                                y_begin + (i / x_resolution) * y_step );
10           z = 0; j = 0;
11           while (norm(z) <= 4 && j < iter_limit) { z = z*z + c; j++; }
12           result[i-start] = j;
13       }
14   }
```

Procedure *compute_mandelbrot_set* in (Listing 1) receives an array *result* where the computed set will be stored. Although it represents the complex plane, *result* is a unidimensional array. This will allow the implementation of similar parallel versions for MPI and OpenMP even though, in the moment of the visualization of the set in a two-dimensional space, some transformations must be done. The mandelbrot set and its complement are given as an array of integers. Each of these values is the number of iterations before 2 is found true. This is useful when visualizing the mandelbrot set. Figure 1 shows some examples. The images were generated for different iteration limits with a similar procedure[4] to those described in [10] and [7, pp. 103–108]. The iteration limit is given by parameter *iter_limit*. Parameters *x_resolution* and *y_resolution* stand for how big the computed set is, that is, the amount of computed detail. In this case, the length of *result* is the product of *x_resolution* and *y_resolution*. Parameters *x_begin*, *x_end*, *y_begin*, and *y_end*

[4] See procedure *print_result* in https://github.com/EStog/mandelbrotc-/tree/0.1/code/common/print_result.cpp

denote the domain of real and imaginary dimensions, respectively. That is, if[5] $c = x + yi$ then $x \in [x\_begin, x\_end]$ and $y \in [y\_begin, y\_end]$. When visualizing the set, the ranges are usually around $x \in [-2.5, 1]$ and $y \in [-1, 1]$. Variables *x_step* and *y_step* determine the level of discretization of the plane, that is, the width of the steps taken in each dimension. The full C++ sequential implementation may be found in folder *code/mandelbrot_sequential*[6].

Parallel implementation

The parallel computing of z_n may be difficult due to the nonlinear character of (Equation 1). Moreover, if (Equation 1) is expanded the following relations hold (Equation 4 and 5):

$$Re(z_n) = \Re(z_{n-1})^2 - \Im(z_{n-1})^2 + \Re(c) \quad (4)$$

$$Im(z_n) = 2\Re(z_{n-1})\Im(z_{n-1}) + \Im(c) \quad (5)$$

May be observed that (Equation 4) and (Equation 5) reference to each other recursively, making more difficult the problem of the parallel computing of z_n. For these reasons, normally, the parallel computing of the Mandelbrot set is realized by making parallel computations of the iterations. In this case, the plane is divided into parts. In the proposed sequential procedure, *result* a unidimensional array, which means that only one loop must be parallelized.

Implementation in OpenMP is straightforward by using directive *omp for* [1, pp. 53–78]. The fact that a unidimensional array has been chosen to store the solution simplifies the division of its range, making it possible to use the same procedure code in the sequential version as well as in the OpenMP implementation and in each subtask of the MPI implementation. In both parallel versions, because each part is independent among each other, it is not necessary to synchronize the execution of the tasks. The C++ code for this procedure is shown in (Listing 2). Its implementation may be found in file *code/common/compute_mandelbrot_subset.cpp*[7].

In this case, parameters *start* and *end* mark the beginning and the ending of the corresponding part. This will allow using the procedure in each subtask in the MPI implementation. In the sequential version, the procedure is called with *start=0* and *end=x_resolution*y_resolution*. When the procedure is used by the sequential and MPI variants the directives of OpenMP have not effect because the compiler flags for OpenMP are no used. Also, in this general procedure, all the other referenced variables (*x_resolution*, *x_begin*, *y_begin*, *x_step*, and *y_step*) are defined as global

[5] i denotes the imaginary unit. That is, $i^2 = -1$.
[6] https://github.com/EStog/mandelbrotc-/tree/0.1/code/mandelbrot_sequential
[7] https://github.com/EStog/mandelbrotc-/tree/0.1/code/common/compute_mandelbrot_subset.cpp

constants because their values will not change while the execution of the programs. The full C++ implementation using OpenMP may be found in folder *code/mandelbrot_openmp*[8].

Listing 2. C++ procedure used to compute the Mandelbrot set in the sequential implementation as well as in the parallel ones.

```
1   void compute_mandelbrot_subset(int* result, int iter_limit, int start, int end) {
2       int i, j;
3       complex<double> c, z;
4
5       # pragma omp parallel shared(result, iter_limit, start, end) private(i, j, c, z)
6       # pragma omp for schedule(runtime)
7       for (i = start; i < end; i++) {
8           c = complex<double>(x_begin + (i % x_resolution) * x_step,
9                               y_begin + (i / x_resolution) * y_step );
10          z = 0; j = 0;
11          while (norm(z) <= 4 && j < iter_limit) { z = z*z + c; j++; }
12          result[i-start] = j;
13      }
14  }
```

In the case of MPI, partition of the loop has to be done by hand. That is, to follow the master-slave procedure [11]:

1. Divide the range of the array into *p* parts, approximately of the same size, where *p* is the number of available processors.
2. Compute the *i*-th part by using processor *i*.
3. Group the result of each partial computation together into one array.

In MPI, two variants may be considered to realize this procedure. One of the variants is to use *MPI_Send* and *MPI_Recv* functions to send and receive messages directly between the processors [12,13]. One of the processors, the master, distribute the tasks between the others and group the results together into one array. That processor also computes a part of the whole solution. The C++ code for this processor is shown in (Listing 3). The other processors, the slaves, only receive the indexes that define a part to be computed. After generated, they send the part to the master. The C++ code for these processors are shown in (Listing 4). In the two cases (master and slaves)

[8] https://github.com/EStog/mandelbrotc-/tree/0.1/code/mandelbrot_openmp

part_width=result_size/processors_amount. The full C++ implementation using MPI with *MPI_Send* and *MPI_Recv* functions may be found in folder *code/mandelbrot_mpi_send_recv*[9].

The other variant in MPI is to use *MPI_Gather* function which allows gathering the partial computations of each slave into one array [12]. Its use, in this case, is very concise as can be seen in (Listing 5). After space has been reserved for arrays *result* and *partial_result*, only remains to compute the part in each processor –including the master– and then gather this result by using *MPI_gather* function. In this case *part_width=result_size/processors_amount* and *start=current_processor*part_width*. The complete C++ implementation may be found in folder *code/mandelbrot_mpi_gather*[10].

Listing 3. C++ code executed by the master in one of the MPI implementation variants.

```
1       // distribute tasks
2       int start, end = 0;
3       for (int i = 1; i < processors_amount; i++) {
4               start = end; end += part_width;
5               int message[2] = {start, end};
6               MPI_Send(message, 2, MPI_INT, i, 0, MPI_COMM_WORLD);
7       }
8       compute_mandelbrot_subset(current+end, iter_limit, end, result_size);
9       // join pieces together
10      for (int i = 1; i < processors_amount; i++) {
11              MPI_Recv(current, part_width, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
12              current += part_width;
13      }
```

Listing 4. C++ code executed by the slaves in the MPI implementation.

```
1       int message[2];
2       MPI_Recv(message, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
3       int* partial_result = new int[part_width];
4       compute_mandelbrot_subset(partial_result, iter_limit, message[0], message[1]);
```

[9] https://github.com/EStog/mandelbrotc-/tree/0.1/code/mandelbrot_mpi_send_rec
[10] https://github.com/EStog/mandelbrotc-/tree/0.1/code/mandelbrot_mpi_gather

```
5       MPI_Send(partial_result, part_width, MPI_INT, 0, 0, MPI_COMM_WORLD);
6       delete[] partial_result;
```

Listing 5. C++ code for MPI using *MPI_gather* function.

```
1       compute_mandelbrot_subset(partial_result, iter_limit, start, start+part_width);
2       MPI_Gather(partial_result, part_width, MPI_INT, result, part_width, MPI_INT, 0, MPI_COMM_WORLD);
```

Results and discussion

Execution environment

The trials consisted in running each implementation for iteration limits 100, 1000, 10000, and 100000 and with one, two, four, and eight processors. In the case of OpenMP the schedule strategies *static*, *dynamic*, and *guided* were considered. The scheduling strategy and the number of processors were passed to the program through environment variables *OMP_SCHEDULE* and *OMP_NUM_THREADS* [12]. Each combination of program, iteration limit, and the number of processors were executed three times and the average of the results was studied by using high-performance computing metrics. Each program was executed in random order with respect to each other, each iteration limit and number of processors in a machine dedicated solely to the running of the trials[11]. Also, the considered resolution –that is, the size of the computed set– was 1024x1024.

The running machine was a computer model *HP Notebook - 15-db0069wm*[12]. In Tables 1 and 2 it is shown relevant information about the running machine and operating system as well as programming and execution tools and libraries, respectively. In file *data/info.txt*[13] may be found information that was automatically recorded at the beginning of the whole experiment by using program *inxi*[14] in *root* mode[15].

[11] X graphics and other services like *AppArmor* were deactivated.
[12] https://support.hp.com/us-en/product/hp-15-db0000-laptop-pc/20395843/model/24094114/document/c06125323
[13] https://github.com/EStog/mandelbrotc-/tree/0.1/data/info.txt
[14] See Linux man page by using command *man inxi*.
[15] The used command line was *sudo inxi -Ffmxxx -t c20 -z -! 31*.

A Python 3 script was developed for the purpose of automatically recording the results to a *csv* file called *data/run_data.csv*[16] and plotting the data by using Python libraries *pandas* [14] and *seaborn* [14], respectively. These results may be found in folder *data*[17]. The whole Python program may be found in folder *trials_runner*[18].

Table 1. Characteristics of the running machine and operating system.

Model	HP Notebook - 15-db0069wm
Microprocessor	AMD Ryzen™ 5 2500U Quad-Core
RAM	8 GB DDR4-2400 SDRAM
Operating System	OpenSUSE Leap 15.1
Type	64 bits
Kernel	4.12.14-lp151.28.40-default

Table 2. Development and execution tools and libraries.

Tools and libraries	**Name**	**Version**	**OpenSUSE package**
C++ compiler	GNU Compiler for C/C++	7.5.0	gcc 7-lp151.3.5
Building system	CMake	3.10.2	cmake 3.10.2-lp151.4.1
MPI development and programming environment	Local Area Multicomputer (LAM)	7.1.4	lam 7.1.4-lp151.2.38
MPI library	OpenMPI	1.10.7	openmpi 1.10.7-lp151.11.4
OpenMP library	GNU Offloading and Multi Processing Runtime Library	8.2.1	libgomp1 8.2.1+r264010-lp151.1.33

Execution time

Although OpenMP and MPI provide specialized functions to measure the execution time of a program [12,15], the execution time was measured by using a method that is valid to all the considered implementations. The function *clock_gettime* and the clock *CLOCK_MONOTONIC_RAW*[19] were used to obtain a monotonic raw hardware-based real-time that cannot be disturbed by system calls. This allowed having a normalized and non-biased way of measuring time.

[16] https://github.com/EStog/mandelbrotc-/tree/0.1/data/run_data.csv
[17] https://github.com/EStog/mandelbrotc-/tree/0.1/data
[18] https://github.com/EStog/mandelbrotc-/tree/0.1/trials_runner
[19] See Linux man page for *clock_gettime(3)* by using command *man 3 clock_gettime*.

Only the master and main thread execution time were measured in the case of MPI and OpenMP, respectively. Also, in all variants, only the code involved in computing the mandelbrot set was measured. That is, initialization and finalization code, including the allocation and deallocation of *result* array, was not measured. In (Listing 6) is shown the function used to obtain the current time. This function may be found in file *code/common/now.cpp*[20].

Listing 6. C++ function used to obtain current time.

```
1    #include <ctime>
2    #include <cstdlib>
3
4    double now() {
5        struct timespec tp;
6        if (clock_gettime(CLOCK_MONOTONIC_RAW, &tp) != 0) exit(1);
7        return tp.tv_sec + tp.tv_nsec / (double)1000000000;
8    }
```

The obtained execution time is showed in Figure 2. The graphics show how MPI variants have the worst execution time while OpenMP implementation is best when using a dynamic schedule. Also, it is important to notice that the three OpenMP schedules variants behave with different performances. Moreover, in spite of the fact that both use the same basic strategy, OpenMP with a static schedule has better results than the MPI implementations in these trials. This may be due to the fact that each slave has to allocate memory to store the computed part –and deallocate it at the end– and later sent it to the master. This may cause an overhead that is not seen in the OpenMP variants. Finally, it is observed that MPI variant with *MPI_Send* and *MPI_Recv* functions obtained better results than the variant with *MPI_gather* function. This suggests that, in some cases, it is better to use low-level functions than high-level functions to build a concrete solution in order to manifest better performance.

[20] https://github.com/EStog/mandelbrotc-/tree/0.1/code/common/now.cpp

Revista Innovación y Software
Vol. 1, No. 2, Mes Septiembre-Febrero, 2020
ISSN: 2708-0935
Pág. 12-26
https://revistas.ulasalle.edu.pe/innosoft

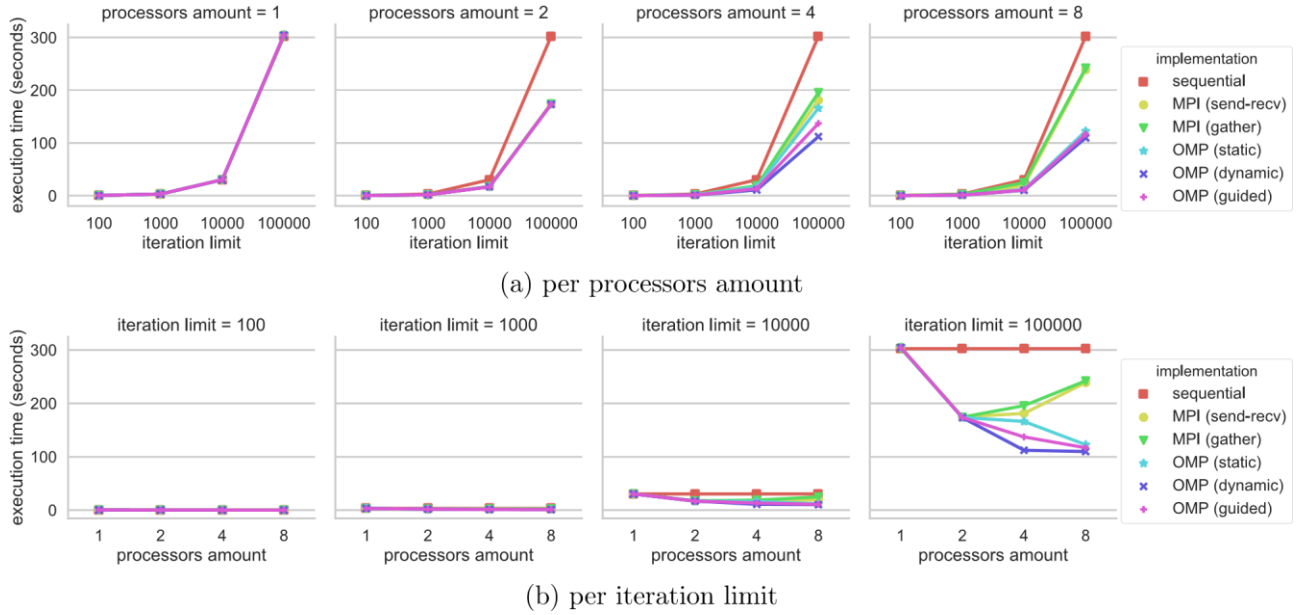


Figure 2. Execution time in seconds. Lower is better.

Speedup

Speedup is a high-performance computing metric that gives an idea of how much the parallel execution time is better than the sequential execution time. The obtained value is better while closer to the number of available processors. The speedup for p processors is (Equation 6):

$$S(p) = \frac{t(1)}{t(p)} \tag{6}$$

Here *t(1)* is the sequential execution time and *t(p)* is the execution time when *p* processors are available in the considered parallel alternative [16–18].

The obtained results for speedup are shown in Figure 3. The graphics show in a better manner the performance difference between the variants. Also, it is noticed that the speedup for four and eight processors do not come near to these values. This suggests that an increase of processors amount will not bring much more improvement to performance in the case of the considered resolution (1024x1024).

Revista Innovación y Software
Vol. 1, No. 2, Mes Septiembre-Febrero, 2020
ISSN: 2708-0935
Pág. 12-26
https://revistas.ulasalle.edu.pe/innosoft

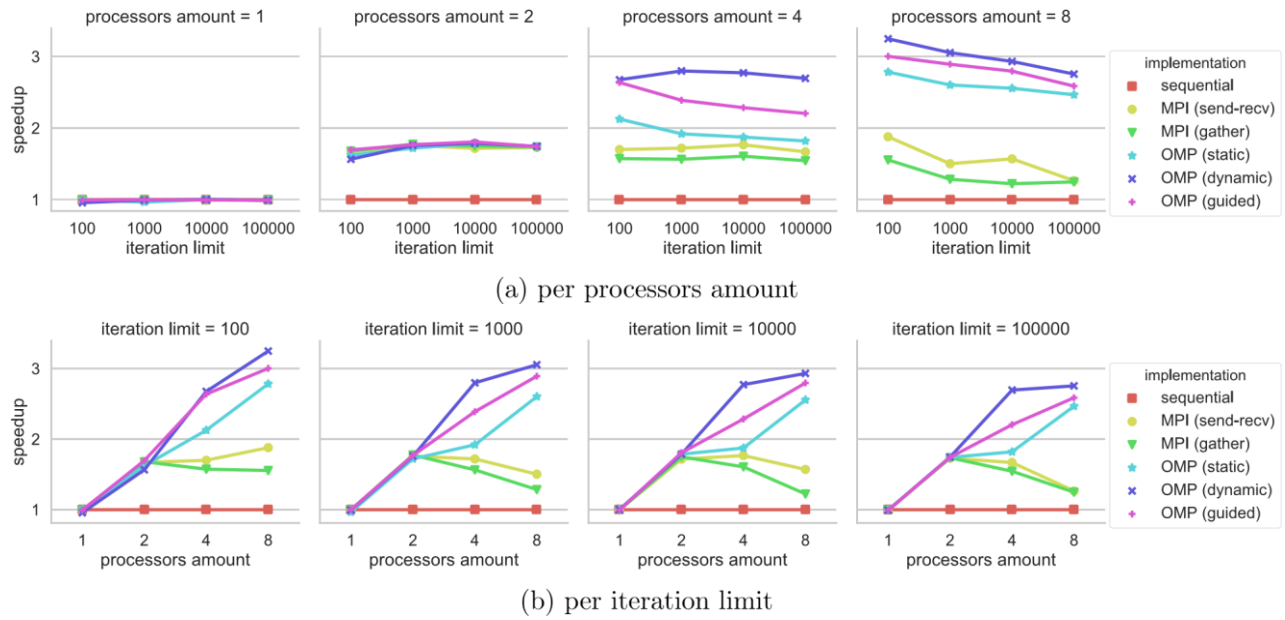


Figure 3. Speedup. Higher is better.

Parallel efficiency

Parallel efficiency is a high-performance computing metric that gives an idea of how much the speedup is close to the number of available processors, that is, how well the parallel program had used the available computational resources (processors in this case). The best-case scenery is when the speedup equals the number of available processors, meaning that the parallel program had maximum exploitation of the available processing units.

The parallel efficiency for p processors is (Equation 7) [16–18]:

$$E(p) = \frac{S(p)}{p} \tag{7}$$

The obtained results are shown in Figure 4. The graphics show the decrease of parallel efficiency with the increment of processors amount. The results are consistent in each iteration limit. This reaffirm the idea that an increase of processors amount will not bring better performance, which is more obvious in the case of MPI. In this case, the decrease in efficiency may be due to the fact that the resolution has been taken constant in these trials, and there will be a moment when the parts to compute become too small. This may bring as a consequence that little gain in performance is obtained by computing the parts in a parallel manner because the time that takes to transmit a message is almost the same as the time to compute apart.

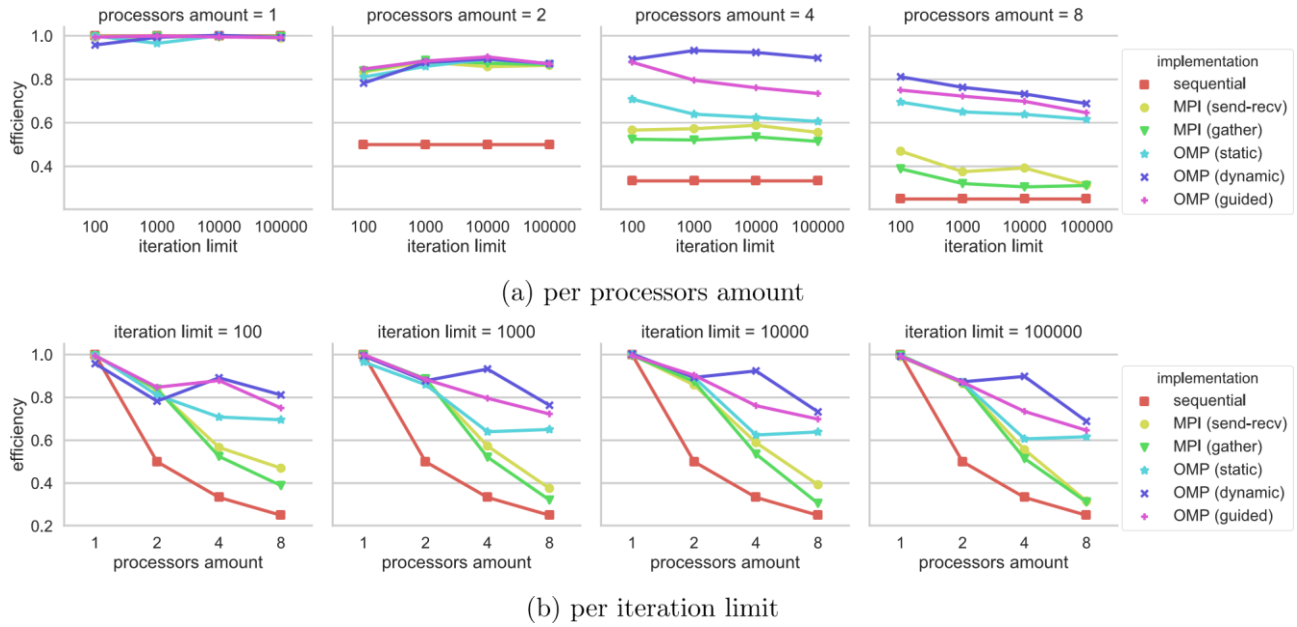


Figure 4. Parallel efficiency. Higher is better.

Conclusions

In the present work, a comparison of the parallel generation of Mandelbrot set by using OpenMP and MPI has been conducted. The trials were executed for different iteration limits, the number of processors, and C++ implementation variants. In this case, and in general, OpenMP obtained better performance results than the MPI implementations. It is worth to notice that, although the present work is a case study and for that reason, results should not be taken as conclusive, the conducted trials may contribute to further research and study. Also, running scripts, images, as well as C++ source code is provided to allow reproduction and enhancing of the experiments. Moreover, the current work may be used as a didactic example to the study of the performance of parallel programs.

References

[1] R. Trobec, B. Slivnik, P. Bulić, and B. Robič, “Programming Multi-core and Shared Memory Multiprocessors Using OpenMP,” in *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*, ser. Undergraduate Topics in Computer Science, R. Trobec, B. Slivnik, P. Bulić, and B. Robič, Eds. Cham: Springer International Publishing, 2018, pp. 47–86.

[2] M. J. Quinn, "Shared-Memory Programming," in *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education, 2003, pp. 404–435.

[3] R. Trobec, B. Slivnik, P. Bulić, and B. Robič, "MPI Processes and Messaging," in *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*, ser. Undergraduate Topics in Computer Science, R. Trobec, B. Slivnik, P. Bulić, and B. Robič, Eds. Cham: Springer International Publishing, 2018, pp. 87–132.

[4] M. J. Quinn, "Message-Passing Programming," in *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education, 2003, pp. 93–114.

[5] P. Czarnul, "Generic Taxonomy of Parallel Computing Systems," in *Parallel Programming for Modern High Performance Computing Systems*. Chapman & Hall/CRC, 2018, pp. 11–12.

[6] M. McCool, J. Reinders, and A. Robison, "Mandelbrot," in *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, Jun. 2012, pp. 131–143.

[7] J. M. Stewart, "Two-Dimensional Graphics," in *Python for Scientists*, 2nd ed. Cambridge University Press, 2017, pp. 82–108.

[8] I. Stewart and A. C. Clarke, "The Nature of Fractal Geometry," in *The Colours of Infinity: The Beauty and Power of Fractals*. Clear Press Ltd, 2004, pp. 2–23.

[9] M. Tracolli, "Parallel generation of a Mandelbrot set," *VIRT&L-COMM*, Apr. 2016. [Online]. Available: http://services.chm.unipg.it/ojs/index.php/virtlcomm/article/view/112

[10] John Burkardt, "MANDEBRTOT - ASCII Portable Pixel Map (PPM) Image of the Mandelbrot Set," Mar. 2020. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/cpp_src/mandelbrot_openmp/ mandelbrot_openmp.html

[11] P. Czarnul, "Master-Slave," in *Parallel Programming for Modern High Performance Computing Systems*. Chapman & Hall/CRC, 2018, pp. 35–39.

[12] ——, "Message Passing Interface (MPI)," in *Parallel Programming for Modern High Performance Computing Systems*. Chapman & Hall/CRC, 2018, pp. 74–102.

[13] M. J. Quinn, "Floyd's Algorithm," in *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education, 2003, pp. 137–158.

[14] W. McKinney, "Plotting and Visualization," in *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 2nd ed. O'Reilly Media, Inc., 2017, pp. 250–283.

[15] P. Czarnul, "OpenMP," in *Parallel Programming for Modern High Performance Computing Systems*. Chapman & Hall/CRC, 2018, pp. 102–118.

[16] ——, "HPC related metrics," in *Parallel Programming for Modern High Performance Computing Systems*. Chapman & Hall/CRC, 2018, pp. 34–35.

[17] R. Trobec, B. Slivnik, P. Bulić, and B. Robič, "History of Parallel Computing, Systems and Programming," in *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*, ser. Undergraduate Topics in Computer Science, R. Trobec, B. Slivnik, P. Bulić, and B. Robič, Eds. Cham: Springer International Publishing, 2018, pp. 9–11.

[18] M. J. Quinn, "Speedup and efficiency," in *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education, 2003, pp. 159–161.

Roles de Autoría

Ernesto Soto Gómez: Conceptualización, Curación de datos, Análisis formal, Investigación, Metodología, Software, Validación, Redacción - borrador original.