PAR Laboratory Assignment
Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

E. Ayguadé, R. M. Badia, J. R. Herrero, J. Morillo, J. Tubella and G. Utrera

Spring 2019-20

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Note: Each chapter in this document corresponds to a laboratory session (2 hours).

1

Task decomposition analysis for the Mandelbrot set computation

1.1 The Mandelbrot set

In this laboratory assignment we are going to explore the tasking model in *OpenMP* to express iterative task decompositions. But before that we will start by exploring the most appropriate ones by using *Tareador*. The program that will be used is the computation of the *Mandelbrot set*, a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognisable two-dimensional fractal shape (Figure 1.1).

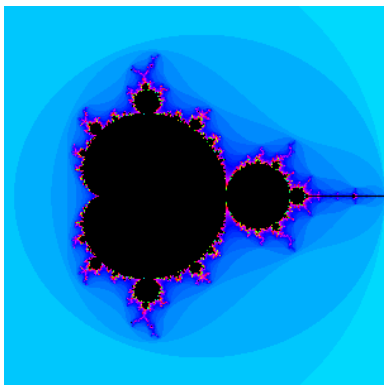

Figure 1.1: Fractal shape

For each point c in a delimited two-dimensional space, the complex quadratic polynomial recurrence $z_{n+1} = z_n^2 + c$ is iteratively applied in order to determine if it belongs or not to the Mandelbrot set. The point is part of the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of z_n never exceeds a certain number however large n gets.

A plot of the Mandelbrot set is created by colouring each point c in the complex plane with the number of steps max for which $|z_{max}| \geq 2$ (or simply $|z_{max}|^2 \geq 2 * 2$ to avoid the computation of the square root in the modulus of a complex number). In order to make the problem doable, the maximum number of steps is also limited: if that number of steps is reached, then the point c is said to belong to the Mandelbrot set.

If you want to know more about the Mandelbrot set, we recommend that you take a look at the following Wikipedia page:

`http://en.wikipedia.org/wiki/Mandelbrot_set`

In the *Computer drawings* section of that page you will find different ways to render the Mandelbrot set.

1. Open the `mandel-seq.c`[1] sequential code to identify the loops that traverse the two dimensional space and the loop that is used to determine if a point belongs to the Mandelbrot set.

2. Compile the two sequential versions of the program using `"make mandel"` and `"make mandeld"`. The first one generates a binary that will be used for timing purposes and to check for the numerical validity of the output (`-o` option). The second one generates a binary that visualizes the Mandelbrot set. Execute both binaries with no additional parameters and compare the execution time reported. When executing `mandeld` the program will open an X window to draw the set; once the program draws the set, it will wait for a key to be pressed; in the meanwhile, you can find new coordinates in the complex space by just clicking with the mouse (these coordinates could be used to zoom the exploration of the Mandelbrot set).

3. Execute `"./mandel -h"` and `"mandeld -h"` to figure out the options that are available to run the program. For example the `"-i"` specifies the maximum number of iterations at each point (default 1000) and `-c` and `-s` specify the center $x_0 + iy_0$ of the square to compute (default origin) and the size of the square to compute (default 2, i.e. size 4 by 4). For example you can try `"./mandeld -c -0.737 0.207 -s 0.01 -i 100000"` to obtain a different view of the set.

1.2 Task decomposition analysis with *Tareador*

Next you will analyze, using *Tareador*, the potential parallelism for two possible task granularities that can be exploited in this program: a) *Point:* a task corresponds with the computation of a single point `(row,col)` of the Mandelbrot set; and b) *Row:* a task corresponds with the computation of a whole `row` of the Mandelbrot set.

1. Complete the sequential `mandel-tar.c` code partially instrumented in order to individually analyze the potential parallelism when tasks are defined at the two granularity levels indicated (*Point* or *Row*). For each granularity, compile the instrumented code using the two targets in the `Makefile` that generate the non–graphical and graphical versions of the program.

2. Interactively execute both `mandeld-tar` and `mandel-tar` using the `./run-tareador.sh` script, indicating the name of the instrumented binary to be executed. The size of the image to compute (option `-w`) is defined inside the script (we are using `-w 8` as the size for the Mandelbrot image in order to generate a reasonable task graph in a reasonable execution time, look for the small Mandelbrot window that will be opened when using `mandeld-tar` and click inside in order to finish its execution).

3. Which are the two most important common characteristics of the task graphs generated for the two task granularities (*Point* and *Row*) for the non-graphical version of `mandel-tar`? Why the task graphs generated for `mandel-tar` and `mandeld-tar` are so different? Which section of the code do you think is causing the serialization of all tasks in the graphical version? How will you protect this section of code in the parallel *OpenMP* code in the next sections?

4. Reason which one of the two task granularities would be more appropriate to apply to implement a parallel version of the Mandelbrot code.

[1] Copy the all files necessary to perform this laboratory assignment from `/scratch/nas/1/par0/sessions/lab3.tar.gz`.

2

Point decomposition in *OpenMP*

In this session you will explore different options in the *OpenMP* tasking model to express the *Point decomposition* for the Mandelbrot computation program. You will analyse the scalability and behaviour of your implementation using the instrumentation and analysis tools you should start to be familiar with.

2.1 *Point* strategy implementation using *OpenMP*`task`

The simplest way to create a task in *OpenMP* for the computation of each point in the Mandelbrot set (*Point* task decomposition) is shown in Figure 2.1: each iteration of the `col` loop will be executed as an independent task. But before that, the program needs to create the pool of threads that will execute the tasks generated by the previous pragma. In this first version of the program the team of threads is created in each iteration of the `row` loop, by using the `parallel` construct; immediately after that, a single task generator is specified by using the `#pragma omp single` work–distributor in *OpenMP*. Observe that, as indicated in the `task` pragma, each task gets a private copy of variable `col` initialised with the value it has at task creation time (`firstprivate` clause). With this, when one of the tasks is extracted from the pool by any of the threads in the team, it has the values for variables `row` (shared variable by default) and `col` (privatised and initialised at task creation time) necessary for the computation of a specific point in the Mandelbrot set. Once all tasks are finished, threads will leave the implicit barrier and continue execution.

```
for (row = 0; row < height; ++row) {
    #pragma omp parallel
    #pragma omp single
    for (col = 0; col < width; ++col) {
        #pragma omp task firstprivate(col)
        {
        ...
        do  {
            temp = z.real*z.real - z.imag*z.imag + c.real;
            z.imag = 2*z.real*z.imag + c.imag;
            z.real = temp;
            lengthsq = z.real*z.real + z.imag*z.imag;
            ++k;
        } while (lengthsq < (N*N) && k < maxiter);

        output[row][col]=k;
        }
    }
}
```

Figure 2.1: Simplest tasking code for *Point* granularity

1. Edit the initial task version in `mandel-omp.c`. Check that it corresponds with the *Point* parallelization shown in Figure 2.1 and insert the missing *OpenMP* directives to make sure that the dependences you detected for the graphical version in the previous section are honoured.

2. Use the `mandeld-omp` target in the `Makefile` to compile and generate the parallel binary for the graphical version. Interactively execute it to see what happens when running it with only 1 thread and a maximum of 10000 iterations per point (i.e. `OMP_NUM_THREADS=1 ./mandeld-omp -i 10000`). Is the image generated correct? Compare the execution time with the sequential version (i.e. `./mandeld -i 10000`). Why the *OpenMP* version takes more time to execute? Interactively execute it with 8 threads and the same number of iterations per point (e.g. `OMP_NUM_THREADS=8 ./mandeld-omp -i 10000`). Is the image generated correct? Is the execution time reduced?

3. Use the `mandel-omp` target in the `Makefile` to compile and generate the parallel binary for the non-graphical version. Execute this version with 1 and 8 threads, making sure that the output file generated (use the `-o` option) is identical in both the sequential and parallel version. Once verified, submit the `submit-strong-omp.sh` script with `qsub` to execute the binary generated and obtain the execution time and speed–up plots. Visualise the *PostScript* file generated. Is the scalability appropriate?

4. In order to better understand how the execution goes, do an *Extrae* instrumented execution, opening the trace generated with *Paraver*. The instrumented execution and trace loading will take some time, please be patient. Table 2.1 adds some new configuration files to the ones you used during the first laboratory assignment to do this tasking analysis, all of them inside the `cfgs/OMP_tasks` directory. For now open `OMP_tasks.cfg` to visualize when tasks are created and executed. How many tasks are created/executed and which thread(s) is/are creating/executing them? To count tasks, please open the `OMP_tasks_profile.cfg` configuration file, making sure that it is applied to the whole trace. How many times the `parallel` construct is invoked? How many times the `single` worksharing construct is invoked? How do you obtain these numbers?

***Paraver* cfg file**	**Timeline showing ...**
OMP_tasks	when tasks in `task` and `taskloop` are created and executed
OMP_tasks_functions	task function created and executed (file and line inside file)
OMP_taskloop	when the `taskloop` construct is executed
OMP_taskwait	when a `taskwait` construct is executed
OMP_taskgroup	when a `taskgroup` construct is executed
OMP_in_taskgroup	when a task is starting or waiting in a `taskgroup`
***Paraver* cfg file**	**Profile showing ...**
OMP_task_profile	the total time, percentage of time, number of instances or average duration spent in task creation/execution

Table 2.1: Set of configuration files to support tasking analysis in *Paraver*– upper part: timeline views; lower part: statistical summaries.

If you change the previous parallelization for *Point* to the code shown in Figure 2.2, what would it change? Observe that now only one thread, the one that gets access to the `single` region, traverses all iterations of the `row` and `col` loops, generating a task for each iteration of the innermost loop (point). We have introduced the `#pragma omp taskwait` synchronization construct, which defines a point in the program to wait for the termination of all child tasks generated up to that point. In this case, the thread will wait until all tasks for each one of the rows finish; after that, the thread will advance one iteration of the `row` loop and generate a new bunch of tasks. Modify the code in `mandel-omp.c` to reflect this change and repeat the previous evaluation (scalability and tracing). How many times the `parallel` construct is now invoked? How many times the `single` worksharing construct is now invoked? How many times the `taskwait` construct is executed? How many tasks are created/executed and which thread(s) is/are creating/executing them? Has the granularity of tasks changed?

Alternatively to the use of `taskwait` one can use the `#pragma omp taskgroup` construct, which defines a region in the program, at the end of which the thread will wait for the termination of all

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
        {
        ...
        }
    }
    #pragma omp taskwait // waiting point for all child tasks
}
```

Figure 2.2: Second tasking code for *Point* granularity

descendant (not only child) tasks, as shown in Figure 2.3. Do you observe any difference in the behaviour and timing when this task synchronization construct is used?

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    #pragma omp taskgroup
    {
        for (col = 0; col < width; ++col) {
            #pragma omp task firstprivate(row, col)
            {
            ...
            }
        }
    } // waiting point for all descendant tasks in taskgroup region
}
```

Figure 2.3: Third tasking code for *Point* granularity using `taskgroup` instead of `taskwait`

Do you think the `taskwait` construct in Figure 2.2 is really necessary? Or in other words, is it necessary to wait for the termination of all tasks in a row before generating the tasks for the next rows? Modify the code in `mandel-omp.c` to remove this task barrier and repeat the previous evaluation (scalability and tracing). Has the number of tasks created/executed changed? Why the threads generating tasks stops task generation, then executes some tasks, and then proceeds generating new tasks?

2.2 Granularity control with `taskloop`

Finally we will make use of the `taskloop` construct, which generates tasks out of the iterations of a `for` loop, allowing to better control the number of tasks generated or their granularity. As you know, you can make use of the `num_tasks` clause to control the number of tasks generated out of the loop, with the appropriate number to specify the number of tasks; alternatively, you can use the `grainsize` clause to control the granularity of tasks, with the appropriate number to specify the number of iterations per task; if none of these clauses is specified, the *OpenMP* implementation decides a value for them, depending on the number of threads in the parallel region. Figure 2.4 shows the last version for the *Point* parallelization making use of `taskloop`.

1. Edit the last version in `mandel-omp.c` in order to make use of `taskloop`.
2. As before, use the `mandeld-omp` target in the `Makefile` to compile and generate parallel binary for the graphical version. Interactively execute it to see that the code produces the appropriate result.
3. Use the `mandel-omp` target in the `Makefile` to compile and generate parallel binary for the non-graphical version. For `grainsize(1)` (or equivalently for `num_tasks(800)`), which is equivalent

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(64) // grainsize(width/64)
    for (col = 0; col < width; ++col) {
        ...
    }
}
```

Figure 2.4: Forth tasking code for *Point* granularity

to the second `task` version that you analyzed, why the new version based on `taskloop` performs better than the version based on `task`? In order to justify your answer, do an *Extrae* instrumented execution, opening the trace generated with *Paraver* and using the appropriate configuration files.

4. The `taskloop` construct accepts a `nogroup` clause that eliminates the implicit task barrier (`taskgroup`) at the end. Do you think this task barrier can be eliminated? If so, edit the code and insert it before doing the performance evaluation below.

5. Explore how the new version behaves in terms of performance when using 8 threads and for different task granularities, i.e. setting the number of tasks or iterations per task to 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1, for example. Reason about the results that are obtained.

3

Row decomposition in *OpenMP*

Based on the experience and conclusions you obtained from the previous chapter, in this session you will have to write a new parallel version fo `mandel-omp.c` that obeys to a *Row* task decomposition. Do the complete analysis of scalability and tracing to conclude about the performance that is obtained with this coarser-grain task decomposition, reasoning about the performance differences that you observe when comparing both strategies (*Row* and *Point*).

Optional: Write a parallel version for `mandel-omp.c` making use of the `for` work–sharing construct instead of tasking to distribute the work among the threads in the team created in the `parallel`, You should explore the effect of the different `schedule` kinds in *OpenMP* for the `for` work–sharing construct and observe how they appropriate tackle the load balancing problem in the *Mandelbrot* program.

4

Deliverable

Important:

- Deliver a document that describes the results and conclusions that you have obtained when doing the assignment. In the following subsections we highlight the main aspects that should be included in your document. Only PDF format will be accepted.
- The document should have an appropriate structure, including, at least, the following sections: Introduction, Parallelization strategies, Performance evaluation and Conclusions. The document should also include a front cover (assignment title, course, semester, students names, the identifier of the group, date, ...) and, if necessary, include references to other documents and/or sources of information.
- Include in the document and comment, at the appropriate sections, relevant fragments of the C source codes that are necessary to understand the parallellization strategies and their implementation (i.e. *Row* and *Point* for Tareador instrumentation and for the `task`– and `for`–based OpenMP parallelization strategies, both non–graphical and graphical options).
- You also have to deliver the complete C source codes for Tareador instrumentation and all the OpenMP parallelization strategies that you have done. Include both the PDF and source codes in a single compressed tar file (`GZ` or `ZIP`). Only one file has to be submitted per group through the Raco website.

As you know, this course contributes to the **transversal competence "Tercera llengua"**. Deliver your material in English if you want this competence to be evaluated. Please refer to the "Rubrics for the third language competence evaluation" document to know the *R*ubric that will be used.

4.1 Task decomposition and granularity analysis

Explain the different task decomposition strategies and granularities explored with *Tareador* for the *Mandelbrot* code, including in your deliverable the task dependence graphs obtained and clearly indicating the most important characteristics for them (use the small test case `-w 8` for this analysis). Explain which section of the code is causing the serialization of all tasks in `mandeld-tar` and how this section of code should be protected when parallelizing with *OpenMP*. Reason when each strategy/granularity should be used.

4.2 *Point* decomposition in *OpenMP*

For the *Point* strategy implemented in *OpenMP*, describe and reason about how the performance has evolved for the three `task` versions of the code that you have evaluated, using the speed–up plots obtained and *Paraver* captures. After that, explain the influence of the granularity control available

in the `taskloop` construct, showing how the execution behaves when setting the number of tasks or iterations per task to 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1, for example. Include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000), again including *Paraver* captures to help you in the reasoning.

Include the parallel version that makes use of the `taskgroup` construct. Explain the two fundamental differences with `taskwait` and if they are relevant in this code. Also include the strong scalability analysis and use tracing to show the differences with `taskwait`.

4.3 *Row* decomposition in *OpenMP*

For the *Row* strategy implemented in *OpenMP*, describe the parallelization strategy that you have decided to implement. Reason about the performance that is obtained comparing with the results obtained for the best *Point* implementation, including the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000) and *Paraver* captures that help you to better explain the results that have been obtained.

4.4 Optional: `for`–based parallelization

If you decided to implement the `for`-based parallelization, Include the relevant portion of the *OpenMP* code commenting whatever necessary. Also include the execution time and speed–up plots that have been obtained for different loop schedules when using for example 8 threads (with `-i 10000`). Reason about the performance that is observed.