PAR Laboratory Assignment
Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

Mario Acosta, Eduard Ayguadé, Rosa M. Badia (Q2), Jesus Labarta,
Josep Ramon Herrero (Q1), Daniel Jiménez-González, Pedro Martínez-Ferrer, Adrian Munera,
Jordi Tubella and Gladys Utrera

Fall 2023-24

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Note:

- This laboratory assignment will be done in two sessions (2 hours each). For the first session we suggest you do chapter 2 and start with chapter 3.
- All files necessary to do this laboratory assignment are available in a compressed tar file available from the following location: `/scratch/nas/1/par0/sessions/lab3.tar.gz` in `boada.ac.upc.edu`. Uncompress it with this command line:
`"tar -zxvf ~par0/sessions/lab3.tar.gz"`.

1

Before starting this laboratory assignment ...

Before going to the classroom to start this laboratory assignment, we strongly recommend that you take a look at this section and try to solve the simple questions we propose to you. This will help to better face your first programming assignment in OpenMP: the Mandelbrot set computation.

1.1 Iterative task decompositions

As you have already realised, loops and other iterative constructs are one of the main sources of parallelism in programs. **Iterative task decompositions** are parallelisation strategies that try to exploit this parallelism. For example, consider the simple loops in Figure 1.1 computing a) the elements of vector `C` as the sum of the elements of vectors `A` and `B` and b) the dot product of vectors `A` and `B`.

```
void vectoradd(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++)
       C[i] = A[i] + B[i];
}
```

a)

```
int dotproduct(int *A, int *B, int n) {
   int sum=0;
   for (int i=0; i< n; i++)
       sum += A[i] * B[i];
   return(sum);
}
```

b)

Figure 1.1: Simple examples performing a) vector sum and b) dot product.

In an iterative task decomposition tasks correspond with the execution of one or more loop iterations. The granularity of the task decomposition would be determined by the number of iterations executed per task; iterations do not need to be consecutive, but sometimes this may help to better exploit cache locality and avoid other artefacts you will be learning during the course.

An iterative task decomposition is named "embarrassingly parallel" if the execution of all tasks can be performed totally in parallel without the need of satisfying data sharing and/or task ordering constraints. This is the case in the example in the upper part of Figure 1.1; there are no data races preventing the execution of all iterations in parallel. However the computation in the example in the lower part of the same figure will require some sort of synchronisation in order to avoid the possible data races caused by the access to the variable `sum`.

1.2 Should I remember something from previous laboratory assignments?

Would you be able to write a parallel version in *OpenMP* for these two simple code fragments? We are quite sure that your answer is "Yes!, of course". In the previous laboratory assignment you should have learned about the different options in *OpenMP*, and when to use them, to express tasks out of loops (either with implicit tasks or explicit tasks with `task` and `taskloop`), with the appropriate thread creation (`parallel` and `single`) and how to enforce task order with task barriers (`taskwait` and `taskgroup`), and data sharing constraints (`critical`, `atomic` and `reduction` operations).

In addition you should remember from the first assignment how to use the *Tareador* API and GUI to understand the potential parallelism available in a sequential code, as well as the causes that limit this parallelism. And also the use of *modelfactors* and *Paraver* to visualise the execution of your parallel *OpenMP* program and understand its performance.

1.3 So, what should I do next?

Simply we ask you to think about, better to write in paper, the multiple alternatives to code the parallel versions in *OpenMP* for the two simple codes shown in Figure 1.1. You don't need to deliver them, but we will comment your different solutions in the lab session, if necessary.

2

Task decomposition analysis for the Mandelbrot set computation

2.1 The Mandelbrot set

In this laboratory assignment you are going to explore the tasking model in *OpenMP* to express **iterative task decompositions**. But before that you 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 2.1).

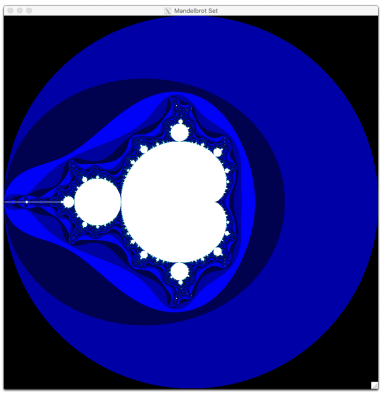

Figure 2.1: Fractal shape

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 n 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`[1]

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

[1] Website last visited on August 31, 2022

1. Open the `mandel-seq.c` sequential code to identify the loops that traverse the two dimensional space (loops with induction variables `row` and `column` and the loop that is used to determine if a point belongs to the Mandelbrot set (`do--while` loop).

2. Compile the sequential version of the program by using `"make mandel-seq"`. You will get a binary that can be executed with different output options: 1) the Mandelbrot set is computed but no output is displayed or written to disk (just for timing purposes of the computational part); 2) `-h` option: the histogram for the values in the Mandelbrot set is also computed; 3) `-d` option: the Mandelbrot set is displayed, for visual inspection; and 4) `-o` option: the values for Mandelbrot set and/or histogram are written to disk, to compare the numerical values obtained with the reference output. When executed with the `-d` option 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. The program includes additional parameters to specify the domain in the complex space where the computation of the Mandelbrot set has to be performed. Execute `"./mandel-seq -help"` to see 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 `"./mandel-seq -d -c -0.737 0.207 -s 0.01 -i 100000"` to obtain a different view of the set.

4. In order to have a reference of the sequential execution, execute the sequential version with `"./mandel-seq -h -i 10000 -o"` to get its execution time and the output file, to be used later to check the correctness of the different parallel versions you will write.

As you can guess at this point, the computation of the Mandelbrot set problem is totally amenable for applying the iterative task decomposition parallelisation strategy. Figure 2.2 shows the double-nested loop that you have found in the source code for the sequential version in `mandel-seq.c`.

```
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        ... // computation of a single point in the Mandelbrot set
    }
}
```

Figure 2.2: Two loops traversing the iteration space for the Mandelbrot set.

Two different alternatives to generate tasks will be explored in this laboratory assignment, with different granularities: a) *Row* decomposition in which a task will correspond with the computation of one or more (consecutive) rows of the Mandelbrot set; and b) *Point* decomposition in which a task will correspond with the computation of one or more (consecutive) points in the same row of the Mandelbrot set. In other words, *Row* will correspond with the distribution of iterations of the loop indexed by the `row` induction variable while *Point* will correspond with the distribution of iterations of the loop indexed by the `col` induction variable, for a fixed value of `row`. The granularity of both *Row* and *Point* decompositions will be determined by the number of iterations per task in the respective loops.

2.2 Task decomposition analysis with *Tareador*

In this section you will study the main characteristics of each proposed task decomposition strategy (*Row* and *Point*) using *Tareador*, in both cases with a granularity of one iteration per task. Then, in the next session you will implement them using the *OpenMP* programming model and explore different task granularities. We recommend that you follow the guidelines in the following bullets and reach the appropriate conclusions.

1. Complete the sequential `mandel-tar.c` code partially instrumented in order to analyse the potential parallelism for the *Row* strategy, with granularity of one iteration of the `row` loop per task. Then

compile the instrumented code using the appropriate target in the `Makefile` that generate the binary for analysis with *Tareador*. **Note:** `tareador_ON()` and `tareador_OFF()` calls are already done in the `main` program.

2. Interactively execute `mandel-tar` binary using the `./run-tareador.sh` script, only indicating the name of the instrumented binary (with no additional options). The size of the image to compute 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).

   For the deliverable: Which are the two most important characteristics for the task graph that is generated? Save the TDG generated for later inclusion in the deliverable.

3. Next interactively execute `mandel-tar` binary using the `./run-tareador.sh`, but now indicating the name of the instrumented binary and the `-d` option. **Important:** look for a small window that will be opened to display the tiny Mandelbrot set that is computed and close it in order to finish its execution, or to be more efficient, press the `Enter/Return` key in your keyboard immediately after launching the execution.

   For the deliverable: Which are the two most important characteristics for the task graph that is generated? Which part of the code is making the big difference with the previous case? How will you protect this section of code in the parallel *OpenMP* code that you will program in the next sessions? Save the new TDG generated for later inclusion in the deliverable.

4. Finally, interactively execute `mandel-tar` binary using the `./run-tareador.sh`, but now indicating the name of the instrumented binary and only the `-h` option.

   For the deliverable: What does each chain of tasks in the task graph represents? Which part of the code is making the big difference with the two previous cases? How will you protect this section of code in the parallel *OpenMP* code that you will program in the next sessions? Don't forget to save the new TDG generated for later inclusion in the deliverable.

Once the study for the *Row* strategy is completed, modify the instrumented `mandel-tar.c` code to analyse the potential parallelism for the *Point* strategy. Repeat the analysis for the three previous executions.

For the deliverable: Which is the main change that you observe with respect to the *Row* strategy?

3

Implementation and analysis of task decompositions in *OpenMP*

In this session you will explore different options in the *OpenMP* tasking model to express the iterative task decomposition strategies 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.

3.1 *Point* decomposition strategy

Implementation using `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 3.1: each iteration of the `col` loop will be executed as an independent `task`.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
        {
        ...
        }
    }
}
```

Figure 3.1: Simplest tasking code for *Point* granularity

But before that, the program needs to create a team of threads that will execute the tasks generated by the previous pragma. The team of threads is created once by using the `parallel` construct; immediately after that, a single task generator is specified by using the `#pragma omp single` work–distributor in *OpenMP*: the thread 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; the rest of threads (i.e. those that do not gain access to the `single`) simply wait at the implicit barrier at its end and, in the meanwhile, execute any of the tasks that are generated. As indicated in the `task` pragma, each task gets a private copy of variables `row` and `col` initialised with the value they have 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 these two variables necessary for the computation of a specific point in the Mandelbrot set. Once all tasks are finished, threads will leave the implicit barriers at the end of the `single` and `parallel` and one of them continue with the execution of the sequential part.

1. Edit the initial task version in `mandel-omp.c`. Check that it corresponds with the *Point* parallelization shown in Figure 3.1 and **insert the missing *OpenMP* directives to make sure that all the dependences you detected with *Tareador* in the previous section are honoured**.

Make sure you use **the most efficient mechanism** to protect each dependence. You should not worry about the two "fake" parallel regions that have been added to delimit its start and end of the `main` program; this is used by the scripts that will trace the execution with *Extrae*, cutting the trace appropriately for your proper analysis.

2. Use the `mandel-omp` target in the `Makefile` to compile and generate the parallel binary for this first version of the parallel code. In order to visually check the correctness of your parallelization, interactively execute it with the `-d` option 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 ./mandel-omp -d -h -i 10000`). Is the image generated correct? Now interactively execute the binary with 2 threads and the same number of iterations per point (e.g. `OMP_NUM_THREADS=2 ./mandel-omp -d -h -i 10000`). Is the image generated still correct?

3. In order to measure the reduction in time due to the parallel execution, submit the execution of the `submit-omp.sh` script indicating the `mandel-omp` binary with 1 and 8 threads. This script does not include the `-d` option but generates an output file using the `-o` option. At this point it is important that you verify that the output files that are generated are correct, by comparing them [1] with the output file obtained from the original sequential version (if necessary compile the binary and execute it).

4. Submit the `submit-strong-omp.sh` script with `sbatch` to execute the binary generated and obtain the execution time and speed–up plots for the 1–20 processor range. Visualise the *PostScript* file generated. Is the scalability appropriate? Let's analyze your implementation.

Overall analysis with *modelfactors*

In order to better understand how the execution goes, submit the `submit-strong-extrae.sh` script, which performs *Extrae* instrumented executions for 1, 4, 8, 12 and 16 and invokes `mfLite.py` to perform a first overall analysis of the parallel execution metrics. The instrumented execution uses a smaller Mandelbrot set (320x320, set with the `-w 320, without changing the maximum number of iterations -i 10000` option inside the `submit-strong-extrae.sh` script) in order to reduce the size of the trace and the time it takes to generate and process it.

Note that a set of files will be generated within subdirectory `mandel-omp-strong-extrae` [2]. In addition to studying the summary offered in file `modelfactor-tables.pdf` you can use *Paraver* to inspect the trace corresponding to the execution with 12 threads.

1. Spend a minute with the first table reported by *modelfactors* in the file `modelfactor-tables.pdf` inside the directory `mandel-omp-strong-extrae`. Notice that the elapsed times and speed–ups reported do not coincide with the numbers previously reported for 1, 4, 8, 12 and 16 threads by `submit-strong-omp.sh`. This is due to the smaller Mandelbrot set computed now (320x320 instead of 800x800). In any. case, the efficiency is really low and getting worse as the number of threads is increased. Take a look at the execution times that are reported.

   For the deliverable: Is the speed–up appropriate? Is the scalability appropriate? Include the *modelfactors* tables and the *PostScript* file generated.

2. Next, do a quick scan of the values reported for the efficiency metrics included in the second table, which as you remember are measured only for the parallel fraction, which is also reported in this second table in `modelfactor-tables.pdf`. First, notice that the parallel fraction is really covering all the execution of the program, so no problems on this. And second, which of the two metrics that contribute to the parallelization strategy efficiency is really making the values low? Should you blame to "load balancing" or to "in execution efficiency"? Scan the information provided in the new third table that is included in the `modelfactor-tables.pdf` file. That table contains information about:

[1] Note that the output is written in binary format storing numbers using the internal representation of integers. You can compare two binary files using the `cmp` linux command, which ends silently if the two files are identical.

[2] Each invocation of `submit-strong-extrae.sh` will overwrite directory `mandel-omp-strong-extrae`. Thus, you can either change its name, or just process its output before moving to the next sections.

- Total number of explicit tasks executed, and load balancing of these number of tasks in terms of number of tasks assigned to threads and the time spent in executing them (which of the two is actually relevant?).
- Average execution time per explicit task (in microseconds) and percentage of that time that the overheads due to synchronization of tasks (synch %) and creating and extracting them from the pool of tasks (sched %) represent.
- And finally, number of task synchronizations (taskwait/taskgroup) that occur (in this version of the code is null).

Notice that in this table the number of tasks executed is independent of the number of threads used and that the average execution time of these explicit tasks seems to be small (less than 10 microseconds). But more importantly observe that the overheads of managing these tasks is really high (even larger that 100%). This gives a clue of the low "In execution efficiency" reported in the second table.

Detailed analysis with *Paraver*

Open the trace generated for the execution with 12 threads and spend some time visually analizing things, trying to see if the analysis with *Paraver* is providing additional clues about what is happening in the parallel execution.

1. Unfold the *Hints* menu in the main *Paraver* window and look for the configuration files to visualise the *implicit tasks* originated in the `parallel` construct and when *explicit tasks* are created and executed. Also, open a default new single timeline window, syncrhonize all of them and zoom in part of the trace to understand what is happening.

   For the deliverable: Which threads are creating and executing the explicit tasks? Is one of the threads devoted to create them, and therefore executing much less? If yes, this should be a clear contributor to load unbalance, right?

2. Next look for the appropriate configuration file in the *Hints* menu that opens the histogram showing the duration of explicit tasks. Each bin of the histogram shows the number of tasks executed by the thread with a specific task duration. Observe the high dispersion in task durations (look for the duration of the shortest and longest tasks). You can change the *Statistic* to see for example % of time executing tasks with a certain task duration. For this histogram the *3D.plane* selector shows the only task type that is generated and executed in our program.

3. You may find convenient to load a configuration file `lab3-point-tasks.cfg` that has all the Paraver timelines already configured for this case using a point decomposition with tasks. Make sure that all the timelines are synchronised and they do cover the entire time execution by executing the command *Fit time scale*.

For the deliverable: Based on the *modelfactors* tables, performance plots, *Paraver* timelines and profiles, do you think the granularity of the tasks is appropriate for this parallelization strategy? **Important:** Take note of all the relevant information in order to draw some conclusions about the behaviour of this code version. You will have to include them in the deliverable for this laboratory assignment and support with the appropriate *Paraver* window captures, *modelfactors* tables and performance plots.

Optimization: granularity control using `taskloop`

Next you 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 (i.e. the number of iterations per task). 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.

Implementation

1. Edit the last version in `mandel-omp.c` in order to make use of `taskloop` to implement the *Point* strategy. At this point you can leave unspecified the granularity for the tasks (i.e. do not use neither `num_tasks` nor `grainsize`), leaving to the *OpenMP* implementation to take the decision about the granularity to apply.

2. Compile this new version and interactively execute it with 1 and 2 threads with the `-d` option to verify that the code still visualizes the correct Mandelbrot set. Then, submit the execution of the `submit-omp.sh` script for 1 and 8 threads to validate the output files generated with respect to the output of the original sequential program and to observe the new execution times.

Overall Analysis:

Submit the `submit-strong-extrae.sh` script to perform the analysis of the *Extrae* instrumented executions for 1, 2, 4, 12 and 16 with `mfLite.py`. Also, submit the `submit-strong-omp.sh` script and visualise the new scalability plot that is obtained.

1. Look at the first table in `modelfactor-tables.pdf` and visualise the new scalability plot that is obtained.

   For the deliverable: Is the version with `taskloop` performing better than the last version based on the use of `task`?

2. More interesting will be the information reported in the second table of `modelfactor-tables.pdf` containing the efficiency metrics.

   For the deliverable: Do you notice a relevant change in the two metrics that contribute to the parallelization strategy efficiency? Should you blame to "load balancing" or to "in execution efficiency"? Scan the information provided in the third table that is included in the `modelfactor-tables.pdf` file. What can you tell now from the total number of tasks generated and the new average execution time for explicit tasks and the synchronization and scheduling overheads percentage? How many tasks are executed per `taskloop`? Is the task granularity better? But notice that task synchronization still takes a big %.

3. Look at the last row of the third table of the `modelfactor-tables.pdf`.

   For the deliverable: Include *modelfactor* tables in the deliverable. Why there are now task synchronizations in the execution? Where are these task synchronisations happening? Continue with the following section to be able to answer these questions.

Detailed Analysis and Possible Optimization:

Open with *Paraver* the trace generated with 12 threads and use the same configuration files as before to visualize the new execution timeline and task executions. Look for the appropriate hint to visualise which kind of task synchronisation (`taskwait` or `taskgroup`) is being used in this version of the program. Alternatively, load the configuration file `lab3-point-taskloop.cfg` that has all the Paraver timelines already configured for this case using a point decomposition with a taskloop. Make sure that all the timelines are synchronised and they do cover the entire time execution by executing the command *Fit time scale*. Observe that the execution does not continue until all tasks generated up to each one of these points are finished, introducing a certain load unbalance.

For the deliverable: Do you think these task barriers are necessary? If your answer was negative, you can add the `nogroup` clause to the `taskloop` clause in order to eliminate the implicit *taskgroup*. Edit the code and submit again the `submit-strong-extrae.sh` script. Has the scalability improved? What about task granularity and overheads?

3.2 *Row* decomposition strategy

Based on the experience and conclusions you obtained from the previous sections, finally we ask you to:

1. Implement a new parallel version for `mandel-omp.c` that obeys to the *Row* strategy and check that the result is correct.

2. Perform an overall analysis using `submit-strong-extrae.sh` and do the analysis of its strong scalability. Obtain the appropriate conclusions with this coarser-grain decomposition.

3. Perform a detailed analysis of some of the execution traces and compare them to the *Point* strategy. You may find convenient to load a configuration file `lab3-row-taskloop.cfg` that has all the Paraver timelines already configured for this case using a row decomposition with taskloop. Make sure that all the timelines are synchronised and they do cover the entire time execution by executing the command *Fit time scale*.

For the deliverable: Support your reasoning and draw the attention to the main differences that you observe when comparing the *Row* and *Point* granularities. Analyse the number of tasks created vs. the number of tasks executed and their granularities. Is there any load unbalance? Any task synchronisation to eliminate?