PAR Laboratory Assignment
Lab 1: Embarrassingly parallelism with OpenMP: Mandelbrot set

E. Ayguadé, J. Garcia, J. R. Herrero,
D. Jiménez, J. Tubella and G. Utrera

Fall 2014-15

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

1

The Mandelbrot set

The Mandelbrot set is a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognizable two-dimensional fractal shape (Figure 1.1).

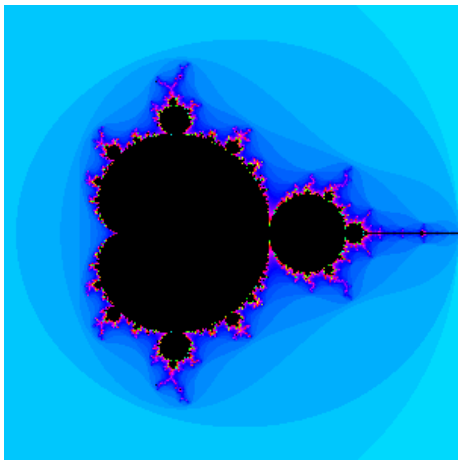

Figure 1.1: Fractal Shape

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 coloring 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-serial.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 sequential version 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 wait for a key to be pressed inside the created X window; once the program finishes drawing the Mandelbrot set, waits again 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 the center x0+iy0 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"`.

[1] Copy the all files necessary to perform this laboratory assignment from `/scratch/nas/1/par0/sessions/lab1.tar.gz`.

2

Task granularity 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) *Row:* a task corresponds with the computation of a whole `row` of the Mandelbrot set; and b) *Point:* a task corresponds with the computation of a single point `(row,col)` of the Mandelbrot set.

1. Complete the sequential `mandel-tareador.c` code partially instrumented to reflect the tasks at the two granularity levels indicated (*Row* and *Point*). Once instrumented, 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-tareador` and `mandel-tareador` using the `./run-tareador.sh` script. The name of the binary and the size of the image to compute (option `-w`) are given as arguments to the script. Observe that 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 (be patient!).

3. Which are the two most important common characteristics of the task graphs generated for the two task granularities (*Row* and *Point*) for the non-graphical version of `mandel-tareador`? Why the task graphs generated for `mandel-tareador` and `mandeld-tareador` 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. In order to analyze the potential scalability of the task decomposition for the non-graphical `mandel-tareador` version, simulate its parallel execution with 1, 2, 4, 8 and 16 processors for each task granularity independently and for a size of `-w 16` (you will have to modify the `run-tareador.sh` script for this), observing how the simulated parallel execution time evolves when increasing the number of processors.

3

Parallelization using `task`

First you will parallelize the original sequential code using tasks, implementing the two strategies analyzed in the previous section (*Row* and *Point*). See the **Optional 1** point below to detect some common errors incurred in this process.

1. Edit the incomplete parallel version `mandel-omp.c` inserting all missing `OpenMP` directives required to implement the *Row* parallelization strategy (making sure that the dependences you detected in the previous section are honored for both the non-graphical and graphical versions).

2. Use the `mandeld-omp` target in the `Makefile` to compile and generate parallel binary for the graphical version. Interactively execute it to see what happens when running it with 1 and 8 threads and a maximum of 10000 iterations per point (e.g. `OMP_NUM_THREADS=8 ./mandeld-omp -i 10000`). Is the image generated correct? In which order are tasks executed when using more than one thread?

3. Use the `mandel-omp` target in the `Makefile` to compile and generate parallel binary for the non-graphical version. Submit the `submit-strong-omp.sh` script to execute the binary generated with 1, 2, 4, 6, 8, 10 and 12 threads and generate the execution time and speed–up plots. Visualize the *PostScript* file generated. Is the scalability appropriate?

4. Once the behavior and performance of the *Row* strategy are appropriate[1], edit the `mandel-omp.c` to implement the *Point* parallelization strategy (with and without graphical output). Repeat the two previous points with this new parallelization strategy and reason about the differences that you observe.

Optional 1: You can use the `mandel-correctness` entry in the `Makefile` to test the correctness of the `task` pragmas introduced and give some suggestions to make the parallelization correct. This entry in the `Makefile` does not generate any executable code, just a report on the screen, as the one shown below for the initial task parallelization that we provide you:

```
mandel-omp.c:100: omp-warning: Variables 'row, col' are in a race condition due to
                               a concurrent usage. Consider synchronizing all
                               concurrent accesses or privatizing the variables.
```

This tool is currently under development and only detects errors related to tasks (not to worksharings). Tests currently included are data-sharing attributes, task synchronization required to ensure data storage, race conditions and task dependencies. The results of this correctness test are also recorded to gather statistics about the most common errors when parallelizing with `OpenMP` tasks. Results are stored totally anonymized, so no worries about being used during the evaluation of your deliverable.

[1] **Important:** save the two final versions generated in order to include them in the deliverable for this laboratory session.

4

Parallelization using `for`

Next you will parallelize the original sequential code using the `for` worksharing construct, again implementing the two parallelization strategies analyzed in Section 2 (*Row* and *Point*).

1. Edit the `mandel-omp.c` to remove (or comment) the directives necessary to implement the `task`–based parallelization and insert the new directives to implement a `for`–based parallelization for the *Row* strategy (with and without graphical output), minimizing the overheads related with parallel execution. The original code already contains a (commented) hint from which you can elaborate the complete parallel code for the `for`-based parallelization. Since we are going to test different loop schedules, we propose to use the `runtime` schedule kind:

   ```
   #pragma omp for schedule(runtime)
   ```

   Later at execution time you can specify the schedule to be used by setting the `OMP_SCHEDULE` environment variable, for example by doing:

   ```
   OMP_SCHEDULE="static" OMP_NUM_THREADS=8 ./mandeld-omp -i 10000
   ```

2. Compile and generate the parallel binary for the graphical version (`"make mandeld-omp"`). Interactively execute it using the simplest `static` schedule to see what happens when running it with 1 and 8 threads and a maximum of 10000 iterations per point. Is the image generated correct? In which order are rows executed when using more than one thread? Is the execution well balanced?

3. Change `OMP_SCHEDULE` to try different loop scheduling strategies: `"static,10"`, `"dynamic,10"` and `"guided,10"`. Visualize the effect of each loop schedule in the way the image is generated. Is the execution time changing? Why?.

4. Compile and generate the parallel binary for the *Extrae* instrumented non-graphical version. The `Makefile` already includes a target `mandel-omp-i` to compile the `OpenMP` code with instrumentation. Execute the resulting binary using 8 threads by submitting the `submit-schedule-omp-i.sh` script. The script executes the binary for the four schedules mentioned above and generates the *Paraver* trace. Visualize all the traces generated with *Paraver* and reason about the parallel execution observed. Measure the load unbalance incurred (quotient between the average time and the maximum time spent by threads in the execution of the parallel region), obtain a profile of the `OpenMP` overheads and complete the table in the Deliverables section. To do that, use the appropriate *Paraver* configuration files.

5. To finish with the analysis of the *Row* parallelization, compile and generate the parallel binary for the non-instrumented non-graphical version (`"make mandel-omp"`). Execute the resulting binary using 8 threads by submitting the `submit-schedule-omp.sh` script which executes the binary generated with 8 threads using the 4 scheduling strategies mentioned above and generates two plots with the execution time and speed–up with respect to sequential. Visualize the *PostScript* file generated. Reason about the results obtained.

6. Finally, to finish this laboratory session, edit again the `mandel-omp.c` to implement the *Point* parallelization strategy (with and without graphical output), minimizing the overheads related with parallel execution. Compile the non-graphical version (`"make mandel-omp"`) and execute the resulting binary using 8 threads by submitting the `submit-schedule-omp.sh`. Compare the behavior of *Row* and *Point* by looking at the execution time and speed–up plots that are generated. Compile the graphical version (`"make mandeld-omp"`) and interactively execute to visualize how the Mandelbrot set is computed, helping you to understand/reason about the performance results obtained.

Important: save the two final versions generated in order to include them in the deliverable for this laboratory session.

Optional 2: How is the Mandelbrot space computed and what is the performance for the different schedules when using the collapse clause? Look at the following incomplete code:

```
#pragma omp for collapse(2) schedule(runtime)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
```

Optional 3: How is the Mandelbrot space computed and what is the performance for the different schedules when combining both `for` and `task`? Look at the following incomplete code:

```
#pragma omp for schedule(runtime)
for (row = 0; row < height; ++row) {
    #pragma omp task
    for (col = 0; col < width; ++col) {
```

Optional 4: Use the information gathered by *Extrae* from the processor performance counters together with the appropriate *Paraver* configuration files to base your explanations on low-level information about number of instructions and floating–point operations executed, number of execution cycles, IPC and number of data cache misses, etc.

5

Deliverable

Deliver a compressed tar file (`GZ` or `ZIP`) including a document that addresses all the questions below (only PDF format will be accepted) and the C source codes for Tareador instrumentation and all the OpenMP `task`– and `for`–based parallelization strategies for the *Row* and *Point* decompositions. Please clearly state the name of all components of the group, the identifier of the group (username `parXXYY`), title of the assignment, date, academic course/semester, ... and any other information you consider necessary. Only one file has to be submitted per group through the Raco website.

As you know, this course contributes to the generic competence "Tercera llengua", in particular G3.2 "To study using resources written in English. To write a report or a technical document in English. To participate in a technical meeting in English." If you want this competence to be evaluated, please refer to the "Rubrics for the third language competence evaluation" document to know which aspects will be evaluated and the criteria that will be used for evaluation.

5.1 Task granularity analysis

1. Which are the two most important common characteristics of the task graphs generated for the two task granularities (*Row* and *Point*) for the non-graphical version of `mandel-tareador`? Include the task graphs that are generated in both cases for `-w 8`.

2. Which section of the code is causing the serialization of all tasks in `mandeld-tareador`? How have you protected this section of code in the parallel OpenMP code?

3. Using the results obtained from the simulated parallel execution for `mandel-tareador` and for a size of `-w 16`, complete the following table with the execution time and speed-up (with respect to the execution with 1 processor) obtained for the non-graphical version, for each task granularity. Comment the results highlighting the reason for the poor scalability.

	Row		Point	
Num. procesors	Exec. time	Speed-up	Exec. time	Speed-up
1				
2				
4				
8				
16				

5.2 `OpenMP task`–based parallelization

1. Include the relevant portion of the codes that implement the `task`-based parallelization for the *Row* and *Point* decompositions (for the non–graphical and graphical options), commenting whatever necessary.

2. For the the *Row* and *Point* decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.

5.3 `OpenMP for`–based parallelization

1. Include the relevant portion of the codes that implement the `for`-based parallelization for the *Row* and *Point* decompositions (for the non–graphical and graphical options), commenting whatever necessary.

2. For the the *Row* and *Point* decompositions of the non-graphical version, include the execution time and speed–up plots that have been obtained for the 4 different loop schedules when using 8 threads (with `-i 10000`). Reason about the performance that is observed.

3. For the *Row* parallelization strategy, complete the following table with the information extracted from the *Extrae* instrumented executions (with 8 threads and `-i 10000`) and analysis with *Paraver*, reasoning about the results that are obtained.

	static	**static,10**	**dynamic,10**	**guided,10**
Running average time per thread				
Execution unbalance (average time divided by maximum time)				
SchedForkJoin (average time per thread or time if only one does)				

5.4 Optional

1. If you have done any of the optional parts in this laboratory assignment, please include your experience, additional information collected or the relevant portion of the code and performance plots obtained in your report.