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

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

Fall 2013-14

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).

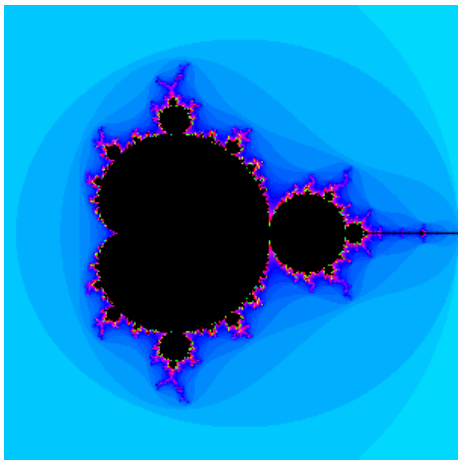

Figure 1.1: Fractal Shape

1. Read the information about the Mandelbrot set that you can find in the following page at Wikipedia:

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

   We recommend that at least you read the first part of that page and the "for programmers" section; based on the understanding of this page and the sequential code `mandel-serial.c`[1], please answer question 4.1 in the deliverables section.

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). Execute `./mandel -h` and `mandeld -h` to figure out the options that are available to run the program. For example `./mandeld -c -0.737 0.207 -s 0.01 -i 100000`.

[1] Copy the file `/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. FOR EACH ONE OF THE TWO TASK GRANULARITIES, do the following steps:

1. Complete the sequential `mandel-tareador.c` code partially instrumented to reflect the task granularity. Once instrumented, compile the instrumented code using the two targets in the `Makefile` that generate the non–graphic and graphic versions of the program.

2. Interactively execute both `mandeld-tareador` and `mandel-tareador` using the `./run-tareador.sh` script[1]. The name of the binary and the size of the image to compute (option `-w`) are given as arguments to the script. Use `16` as the size for the Mandelbrot image in order to generate a small task graph.

3. 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? Change the *Tareador* instrumentation in order to isolate the part of the code that is causing the dependence.

4. In order to analyze the potential scalability of the task decomposition for the non-graphical `mandel-tareador` version, execute the `run-dimemas.sh` script with 1, 2, 4, 8 and 16 processors. Visualize the simulated execution with *Paraver*.

After completing the previous steps for both task granularities, answer the questions in section 4.2.

[1] Alternatively you could submit the execution to the `-l execution` queue using the `submit-tareador.sh` script.

3

Shared-memory parallelization with OpenMP

Next you will parallelize with `OpenMP` the original sequential `mandel-serial.c` code (**NOT the one instrumented with *Tareador***), implementing the two parallelization strategies analyzed in the previous section. For each parallelization strategy:

1. Insert the necessary `OpenMP` directives to implement the parallelization strategy (with and without graphical output), minimizing the overheads related with parallel execution. Start with the simplest loop scheduling strategy (i.e. `static`) and default chunk static size.

2. In the `Makefile` there are two targets: `mandel-omp` and `mandeld-omp` (to generate parallel binaries without and with graphical output, respectively) assuming `maldel-omp.c` the name of the parallelized source code.

   Compile and interactively execute the graphical version to see what happens when running with different numbers of threads (1, 2, 4, 8 and 10) and a maximum of 10000 iterations per point (e.g. `OMP_NUM_THREADS=8 ./mandeld-omp -i 10000`). What do you observe? Is the execution well balanced?

3. Compile and execute the non-graphical version (`mandel-omp` program) submitting the `submit-omp.sh` script. Note that the name of the binary file you want to execute is defined inside the script (currently it has the `mandel-omp` binary name). The script runs the binary with 2, 4, 6, 8, 10 and 12 threads and generates the output file `PROG.times.txt` where `PROG` is the program name. In this file you can look at the output of the execution.

4. Create different versions of your `OpenMP` program with the following loop scheduling strategies: `(static,1)`, `(dynamic,1)` and `(guided,1)`. Is the execution time changing? Why?. You can interactively run the equivalent binaries with graphical output to visualize the effect of the different loop schedules. With the timing information obtained from the non-graphical versions, complete the table in the Deliverables section.

5. Instrument the `OpenMP` version with *Extrae*. The `Makefile` already includes a target `mandel-omp-i` to compile an `OpenMP` code with instrumentation. Execute the resulting binary using 8 threads by submitting the `submit-omp-i.sh` script. Visualize the trace with *Paraver*, for the 4 previous schedules (`static`, `static 1`, `dynamic 1`, and `guided`). 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.

4

Deliverable

Deliver a compressed tar file (`GZ` or `ZIP`) with the `PDF` that contains the answers to the questions below and the requested C source codes. In the `PDF` file clearly state the name of all components of the group. Only one file has to be submitted per group through the Raco website.

4.1 Reading activity

In this laboratory session we ask you to complete the "*...*" of the following paragraphs that would correspond to the introduction part of a complete report. To do that we suggest that you read the first part in the already mentioned Wikipedia page and take a look at the sequential code. You can also use other sources of information if you consider it necessary.

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, write THIS section in english; if not, just do it using catalan or spanish.

In this first part of the report we first briefly describe the basic formulation for the Mandelbrot set [...] [Here you should add an identification of the reference that you have used to write the simple description of the problem, and that you should include below.] and then its implementation in the sequential code available (`mandel-serial.c`). The Mandelbrot set is a particular set of points, in the complex domain ... [Here you should write about the complex space, the recurrence polynomial that is applied to each point in that complex space and the condition used to decide if it belongs or not to the set.]

The simplest algorithm for drawing a picture of the Mandelbrot set is the following. We discretize the complex space in a set of points and each point corresponds to a pixel in a two dimensional plot. To color any such pixel, let `c` (represented by the complex variable ..... [Complete it] at the `mandel-serial.c` code) be the midpoint of that pixel. ... [Here you should continue the description of the sequential algorithm clearly naming the variables that are used in `mandel-serial.c` and the criteria used to decide the color for each point.]

[...] ... [Here you should insert the identification reference and the description of that reference: title, authors, year of publication, website, etc.]

4.2 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 `mandel-tareador`?

2. Which section of the code is causing the serialization of all tasks in `mandeld-tareador`?

3. Using the *Dimemas* results, 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 both task granularities. Comment the results obtained highlighting the reason for the poor scalability, if detected?

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

4.3 `OpenMP` execution analysis

1. For each parallelization strategy of the non-graphical version, complete the following table with the execution time for different loop schedules and number of threads, reasoning about the results that are obtained.

Num. threads	**static**	**static,1**	**dynamic,1**	**guided,1**
1				
2				
4				
6				
8				
10				
12				

2. For each parallelization strategy, complete the following table with the information extracted from the *Extrae* instrumented executions with 8 threads and analysis with *Paraver*, reasoning about the results that are obtained.

	static	**static,1**	**dynamic,1**	**guided,1**
Running average time per thread				
Execution unbalance (average time divided by maximum time)				
SchedForkJoin (average time per thread or time if only one does)				

4.4 Source codes

Include the source code for the two `OpenMP` parallelization strategies (*Row* and *Point*).