PAR Laboratory

11/11/2014 – Q1

LAB 1
FIRST DELIVERABLE

group: par1107
Gabriel Carrillo
Younes Zeriahi

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*.

The two most important characteristics of the task graphs generated for the two task granularities are:

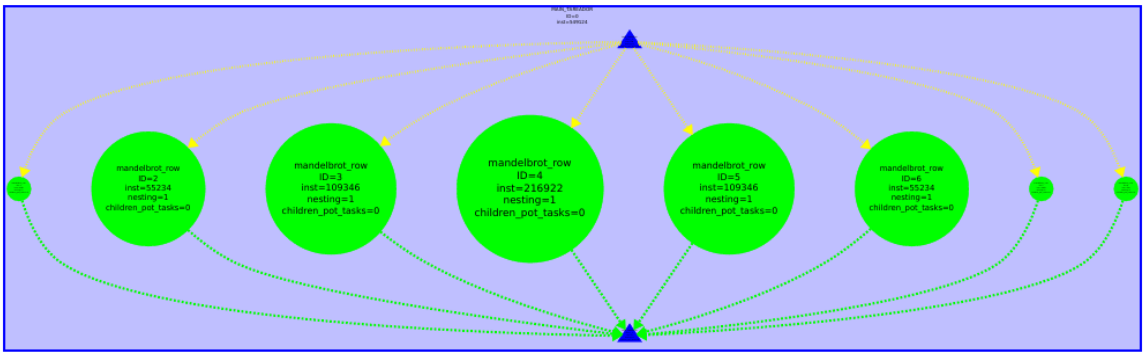
1. The work done in both graphs is distributed more heavily in their center part than in their extremes. There is an imbalance problem (the work is not well balanced) in both parallelization strategies.

We can match the granularity of the tasks between the two task graphs. Indeed, each circle in the Row task graph corresponds to a rectangular block in the Point task graph. And, bigger the circle is, more the corresponding rectangular block contains big tasks.

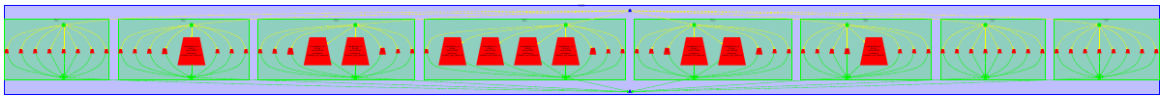
2. Both task graphs show that the work can be done in parallelization. Both divisions of tasks are totally parallelizable. There are no dependencies within a division of tasks.

As we can see, the commented common characteristics are reflected in the following dependencies graphs obtained with Tareador:

- Row:



- Point:



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?

- The section of the code causing the serialization of the tasks in *mandeld-tareador* is the _DISPLAY_ one, which has some global variables (shared between processors) that avoid the parallelization of the different tasks. Indeed, this section is only executed for the mandeld-tareador version.

This _DISPLAY_ section contains the following portion of code :

```
/* Scale color and display point  */
long color = (long) ((k-1) * scale_color) + min_color;
if (setup_return == EXIT_SUCCESS) {
    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
}
```

- We have protected this section of the parallel OpenMP code by adding a critical instruction (represented in blue in the following portion of code). This instruction provides a region of mutual exclusion where only one thread can be working at any given time.

This is the modified portion of code contained in the _DISPLAY_ section that solves the problem:

```
/* Scale color and display point  */
long color = (long) ((k-1) * scale_color) + min_color;
#pragma omp critical
if (setup_return == EXIT_SUCCESS) {
    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
}
```

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.

The execution time of the *mandel-tareador* code with one processor is:

- Row code: 549.854.001 ns
- Point code: 549.992.001 ns

These are the times we use to get the corresponding speed-up's.

	Row		**Point**	
Num. process.	**Exec. time (ns)**	**Speed-up**	**Exec. time (ns)**	**Speed-up**
1	1.615.066.001	1	1.615.578.001	1
2	825.491.001	1.95649134763	829.380.001	1.94793460061
4	496.110.001	3.25545947017	441.962.001	3.65546811116
8	488.051.001	3.30921563052	222.655.001	7.25596997033
16	487.960.001	3.30983276844	113.267.001	14.2634481953

We can see that with the “Row” parallelization strategy, we obtain nearly the same execution times with 4, 8 and 16 processors. This is due to the bigger “Row” task (not parallelized) which takes a long time, and that’s why, even if we add other threads, the total execution time doesn’t change that much.
This problem is resolved by instrumenting the “Point” parallelization strategy which enables us to decompose this biggest “Row” task (and the others “Row” tasks too) in several “Point” tasks and then improve the parallelization of the work and, with this, the execution time.
Therefore, we can see that, with the “Point” decomposition code, the speed-up increases linearly with the number of threads.

***OpenMP task*-based parallelization:**

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

The relevant portion of the code that implement the task-based parallelization for the *Row* decomposition (for the non-graphical and graphical options) is the following:

```
/* Calculate points and save/display */

        for (row = 0; row < height; ++row) {
                #pragma omp task                    //firstprivate by default
                for (col = 0; col < width; ++col) { c
                omplex z, c;
```

On the other hand, the relevant portion of the code that implement the task-based parallelization for the *Point* decomposition (for the non-graphical and graphical options) is the following:

```
/* Calculate points and save/display */

        for (row = 0; row < height; ++row) {
                #pragma omp task                    //firstprivate by default
                for (col = 0; col < width; ++col) {
                        complex z, c;
                        #pragma omp task shared(z, c)
```

As we can see, in the Point decomposition, we had to set variables “z” and “c” as shared. Otherwise, the variables would have been overwritten by the different threads and the final image'd have been a black window.

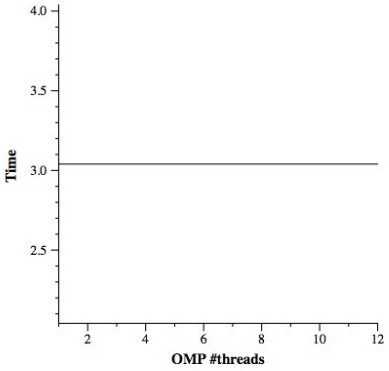
2.- For the *Row* and *Point* decompositions of the non-graphical version, include the execution time and speed-up plots obtained in the strong scalability analysis (with *-i 1000*). Reason about the causes of good or bad performance in each case.

The results we got from executing the non-graphical version of the codes are:

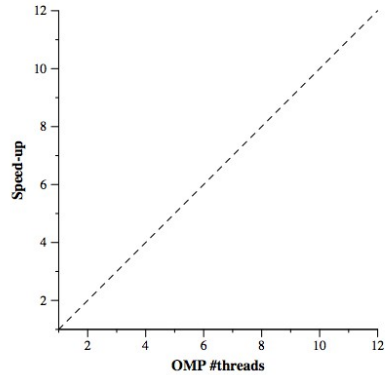
- Row:

	Elapsed	Speedup
1	3.040000000000000000	1.0021929824561403 5087
2	3.040000000000000000	1.0021929824561403 5087
3	3.040000000000000000	1.0021929824561403 5087
4	3.040000000000000000	1.0021929824561403 5087
5	3.040000000000000000	1.0021929824561403 5087
6	3.040000000000000000	1.0021929824561403 5087
7	3.040000000000000000	1.0021929824561403 5087
8	3.040000000000000000	1.0021929824561403 5087
9	3.040000000000000000	1.0021929824561403 5087
10	3.040000000000000000	1.0021929824561403 5087
11	3.040000000000000000	1.0021929824561403 5087
12	3.040000000000000000	1.0021929824561403 5087

The plot we got is the next one:



Average elapsed execution time



Speed-up wrt sequential time

As we can see, there's no speed-up with the different number of processors (and the first one is different from 1), which means we did something wrong that weren't able to find.

- Point:

	Elapsed	Speedup
1	3.040000000000000000	1.000000000000000000
2	3.040000000000000000	1.000000000000000000
3	3.040000000000000000	1.000000000000000000
4	3.040000000000000000	1.000000000000000000
5	3.040000000000000000	1.000000000000000000
6	3.040000000000000000	1.000000000000000000
7	3.040000000000000000	1.000000000000000000
8	3.040000000000000000	1.000000000000000000
9	3.040000000000000000	1.000000000000000000
10	3.040000000000000000	1.000000000000000000
11	3.040000000000000000	1.000000000000000000
12	3.040000000000000000	1.000000000000000000

In this second case, we'd not generate the plot with the images, bus we can see that the speed-up is still constant. That means we're dragging the previous error.

***OpenMP for*-based parallelization:**

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

For the “Row” parallelization strategy, we use four different types of “schedule” :

- static (static, 1) : In this case, the iteration space is broken in chunks of approximatively size N/num_threads. Then these chunks are assigned to the threads with a Round Robin algorithm.
- static, 10 : In this case, the iteration space is broken in chunks of size 10. Then these chunks are assigned to the threads with a Round Robin algorithm.
- dynamic, 10 : In this case, the threads dynamically grab chunks of 10 iterations until all iterations have been executed.
- guided, 10 : This is a variant of the dynamic. In this case, the size of the chunks decreases as the threads grab iterations, but is at least of size 10.

The static schedules provide a low overhead with a good locality usually but can have load imbalance problems.
The dynamic schedules bring a higher overhead with a not very good locality usually but they can solve imbalance problems.

Here is the relevant portion of code that implement the for-based parallelization for the “Row” decomposition :

```
/* Calculate points and save/display */
  #pragma omp for schedule(runtime)
  for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
```

The “schedule(runtime)” instruction enables us to decide which type of schedule we want to use at the execution time.
It means that we executed it with setting the “OMP_SCHEDULE” environment variable like that :
OMP_SCHEDULE=”static” for example if we want to use the static schedule.
For the “Point” parallelization strategy, we add the “collapse(2)” instruction to the openmp pragma of code.

Here is the relevant portion of code that implement the for-based parallelization for the “Point” decomposition :

```
/* Calculate points and save/display */
  #pragma omp for collapse(2) schedule(runtime)
  for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
```

We can use the “collapse(2)”, which allows to distribute the work from the set of the two nested loops, because the loops are perfectly nested and the nest traverses a rectangular iteration space (height * width).

2.- For 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 8 threads and *-i 1000*) and analysis with *Paraver*, reasoning about the results that are obtained

Here are the execution time and speed-up plots we obtained for the "Row" and "Point" decompositions for the 4 different loop schedules when using 8 threads (with -i 10000).

Execution time:

	Row decomposition	Point decomposition
static	3.043295s	3.05
static, 10	3.053612s	3.06
dynamic, 10	3.047636s	3.06
guided, 10	3.050477s	3.06

Speed-up:

	Row decomposition	Point decomposition
static	1.00000000000000000000	0.99672131147540983606
static, 10	0.99672131147540983606	0.99346405228758169934
dynamic, 10	0.99672131147540983606	0.99346405228758169934
guided, 10	0.99672131147540983606	0.99672131147540983606

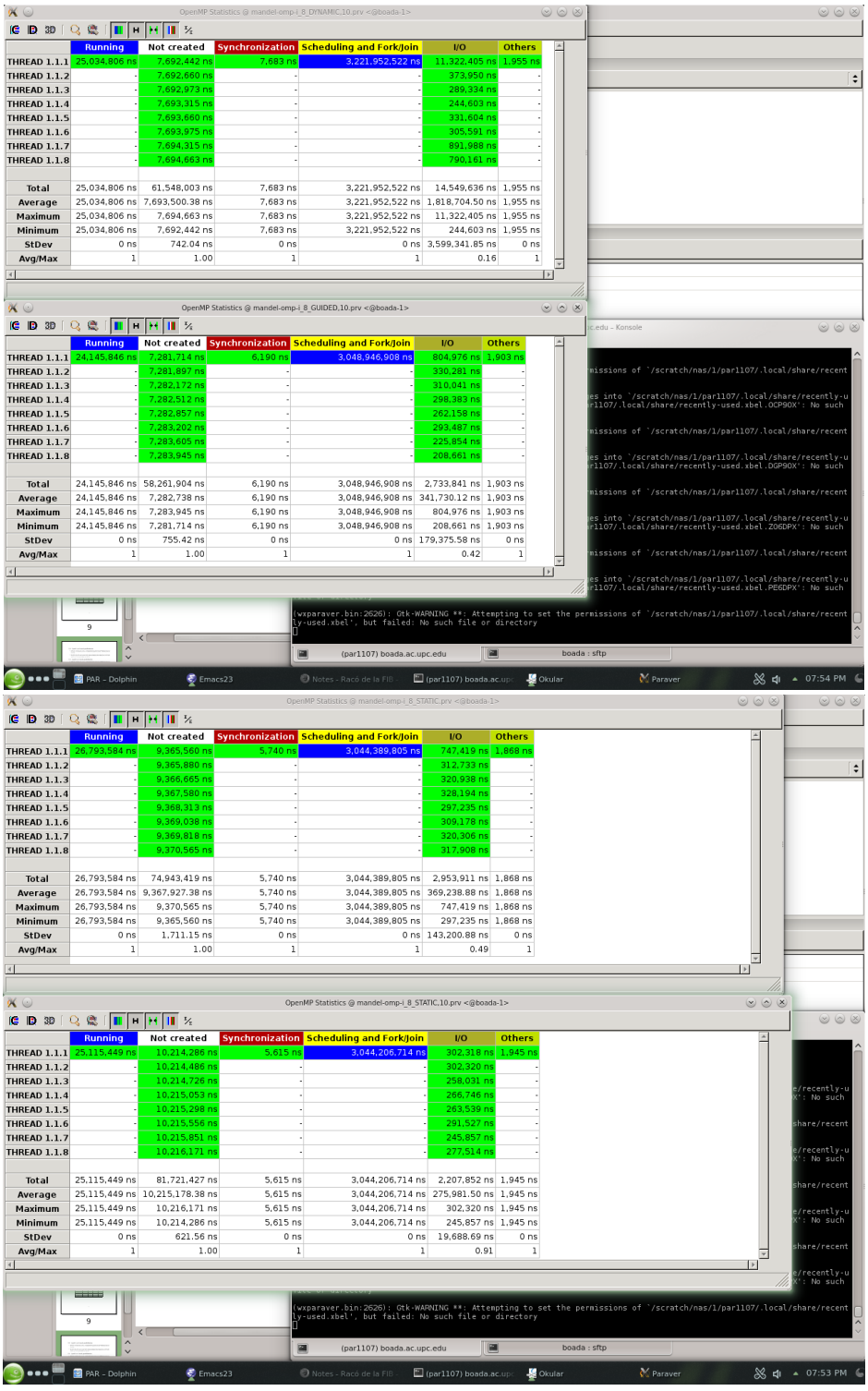
All the results seem to be very similar and honestly, we really don't know why.
We expected to see some differences showing us that for example, the Point decomposition seems to be better than the Row decomposition, and that the dynamic and the guided will provide a better execution time in both cases.

3.- For the *Row* parallelization strategy, complete the following table with the information extracted from the *Extrae* instrumented executions (with 8 threads and *-i 1000*) and analysis with *Paraver*, reasoning about the results that are obtained.

This is the table we obtain for the "Row" parallelization strategy with the information extracted from the Extrae instrumented executions (with 8 threads and -i 10000) and analysis with Paraver :

	static	**static, 10**	**dynamic, 10**	**guided, 10**
Running average time per thread	26.793.584 ns	25.115.449 ns	25.034.806 ns	24.145.846 ns
Execution unbalance (average time divided by maximum time)	26.793.584 ns / 26.793.584 ns = 1	25.115.449 ns / 25.115.449 ns = 1	25.034.806 ns / 25.034.806 ns = 1	24.145.846 ns / 24.145.846 ns = 1
SchedForkJoin (average time per thread or time if only does)	3.044.389.805 ns	3.044.206.714 ns	3.221.952.522 ns	3.048.946.908 ns

We can see on the following pictures that just one thread is working in the four different cases, that explains our numbers obtained in the previous table (with an execution unbalance always equals 1). This must be due to a problem in the instrumentation of our code but we didn't find the origin of this, after trying several times.



Optional:

Because of the problems we had to face, we unfortunately didn't have generated any relevant extra portion of code nor performance plots.