

PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

PAR2102

Alejandro Alarcón Torres

Guillem Reig Gaset

1. Task decomposition and granularity analysis

We have used two different types of granularities, one creates one task to compute one point and the other one creates one task to compute a whole row. The main difference between the two granularities is the workload that each task has to do, but as we know, more tasks imply more overheads, so probably the row granularity is always a better choice, but in a system with lots of processors using the point granularity it may not be a bad idea.

In the graphical execution we can see that the tasks are executed sequentially due to the calls to paint in the X window, so we may need to use the `#pragma omp critical` clause to provide a region of mutual exclusion.

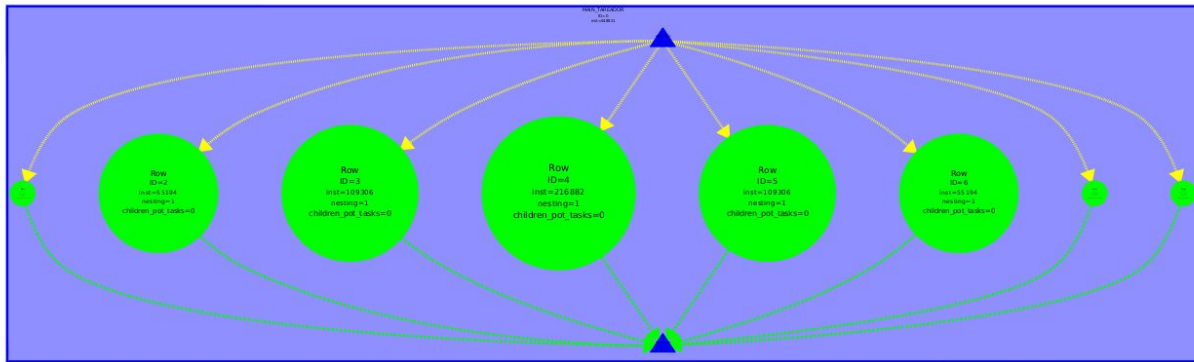


Figure 1.1: Mandel-tar Row decomposition

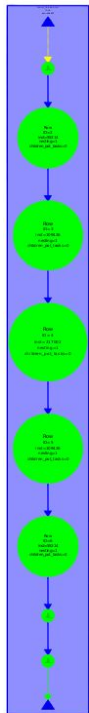


Figure 1.2: Mandel-tar Row decomposition

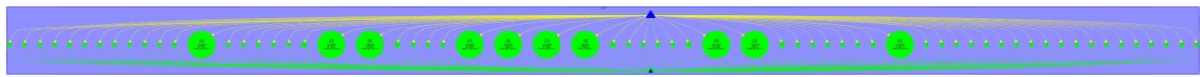


Figure 1.3: Mandel-tar Point decomposition

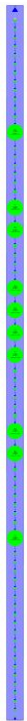


Figure 1.4: Mandel-tar Point decomposition

2. Point decomposition in OpenMP

The image generated using `OMP_NUM_THREADS=1 ./mandeld-omp -i 10000` is the same as the one executing the sequential version, but its execution is slower because we are only using one thread, so we are having overheads due to the tasks that we have defined but we are not exploiting them. If we execute it using `OMP_NUM_THREADS=8 ./mandeld-omp -i 10000` we get the same image but much faster than before, because now we are exploiting the task decomposition.

```
par2102@boada-1:~/lab3$ ./mandeld -i 10000
Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000
Total execution time: 3.067653s

par2102@boada-1:~/lab3$ OMP_NUM_THREADS=1 ./mandeld-omp -i 10000
Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000
Total execution time: 3.210975s

par2102@boada-1:~/lab3$ OMP_NUM_THREADS=8 ./mandeld-omp -i 10000
Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000
Total execution time: 1.240795s
```

Figure 2.1: Execution of mandeld, Point decomposition

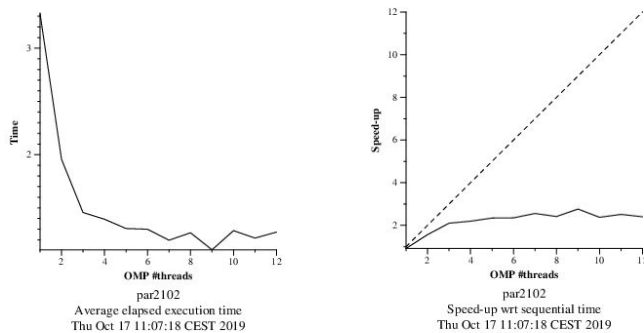


Figure 2.2: Strong scalability plots

For the instrumented execution we need to use the `omp_tasks_profile.cfg` in order to see the tasks created and executed for each thread. To see how many times have the parallel or the single construct been invoked we have to use the `omp_parallel_construct` and the `omp_worksharing_constructs`, respectively.

```

[par2102@boada-1:~/lab3$ ./mandel-omp1

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 1000

Total execution time: 1.490530s
[par2102@boada-1:~/lab3$ ./mandel-omp2

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 1000

Total execution time: 1.200383s
[par2102@boada-1:~/lab3$ ./mandel-omp3

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 1000

Total execution time: 0.105271s

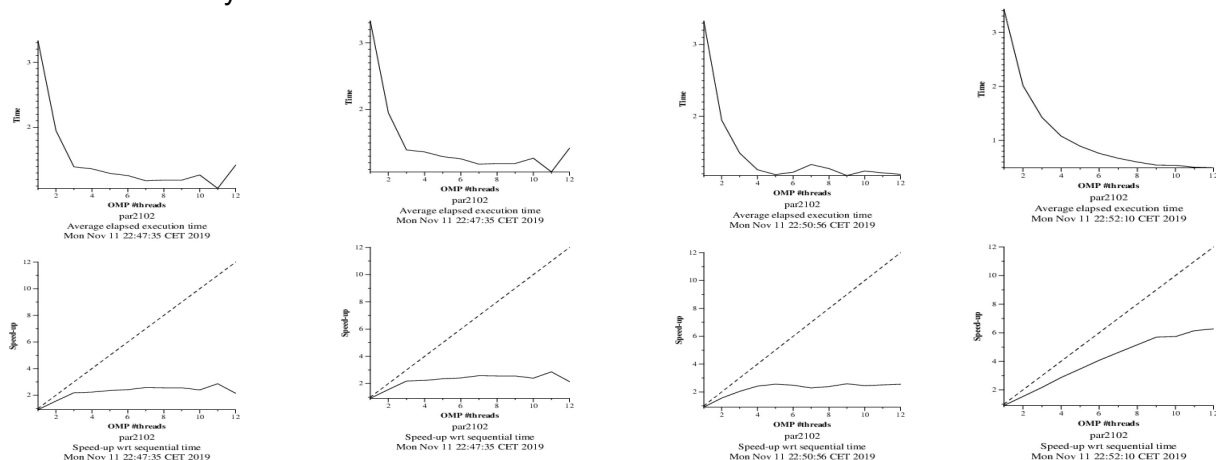
```

Figure 2.3: Execution of each mandel modification

We can appreciate a clear improvement in the execution time after the modifications for each version, specially for the pragma taskloop clause.

The improvement from the 1 to 2 is due to the exteriorization of the taskgroup clause outside the inner loop, which allows to reduce the delays produced by the synchronization of the tasks.

The biggest improvement, however, is given by the taskloop clause added to the externalization of the firstprivate(row). This improves the parallelization as it reduces the overheads and synchronization tasks, at the same time that it distributes the work among the threads in a much more efficient way.

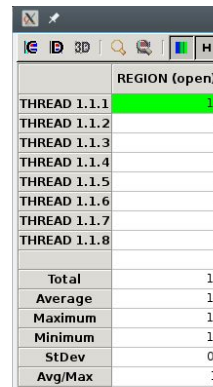


Mandel OMP1



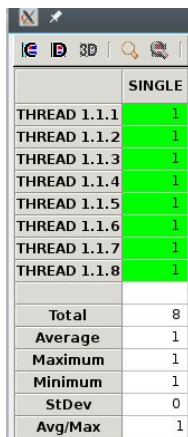
	Running
THREAD 1.1.1	845,146,056 ns
THREAD 1.1.2	501,317,341 ns
THREAD 1.1.3	499,957,437 ns
THREAD 1.1.4	502,829,303 ns
THREAD 1.1.5	502,015,189 ns
THREAD 1.1.6	501,453,824 ns
THREAD 1.1.7	501,137,984 ns
THREAD 1.1.8	502,418,643 ns
Total	4,256,422,219 ns
Average	544,552,777.38 ns
Maximum	845,146,056 ns
Minimum	499,957,437 ns
StDev	113,616,485.35 ns
Avg/Max	0.64

Figure 2.4: Omp1



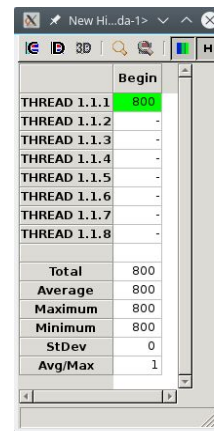
	REGION (open)
THREAD 1.1.1	1
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	1
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Figure 2.5: Omp1 parallel constructs



	SINGLE
THREAD 1.1.1	1
THREAD 1.1.2	1
THREAD 1.1.3	1
THREAD 1.1.4	1
THREAD 1.1.5	1
THREAD 1.1.6	1
THREAD 1.1.7	1
THREAD 1.1.8	1
Total	8
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Figure 2.6: Omp1 worksharing (single) constructs



	Begin
THREAD 1.1.1	800
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-
Total	800
Average	800
Maximum	800
Minimum	800
StDev	0
Avg/Max	1

Figure 2.7: Omp1 - taskwait

Mandel OMP2

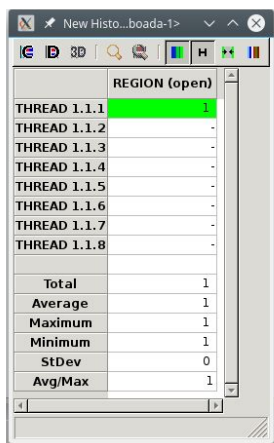


Figure 2.8: Omp2 - parallel constructs

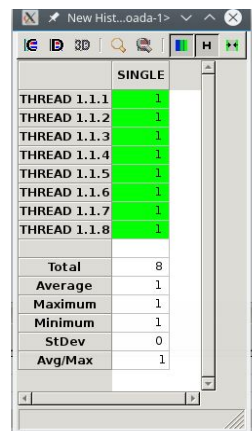


Figure 2.9: Omp2 single

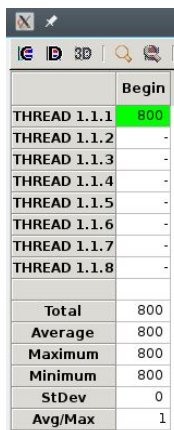


Figure 2.10: Omp2 taskgroup

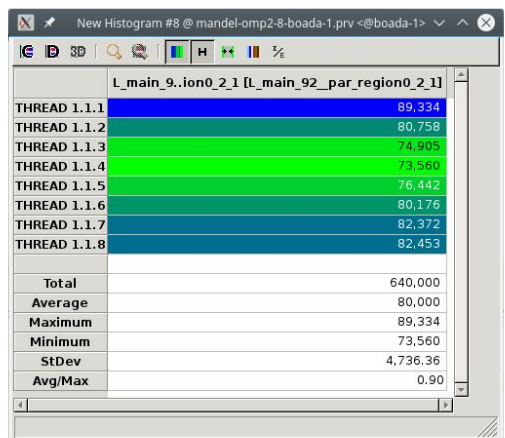


Figure 2.11: Omp2 #tasks



Figure 2.12: Omp2

Mandel OMP3

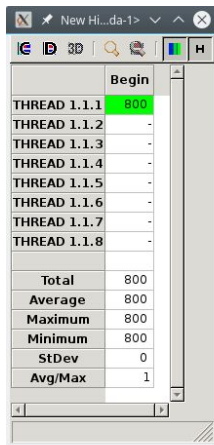


Figure 2.13: Omp3 #tasks

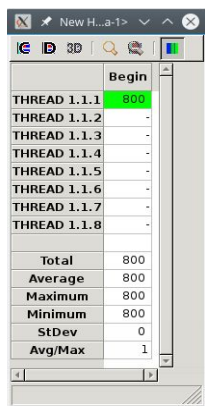


Figure 2.14: Omp3 taskloop

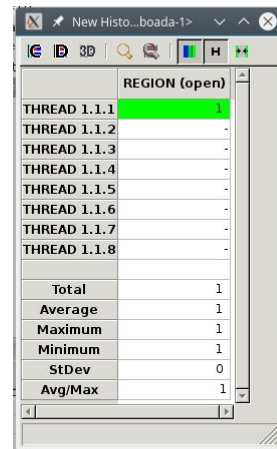


Figure 2.15: Omp3 parallel

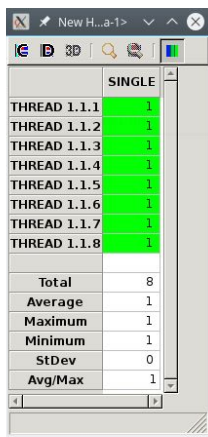


Figure 2.16: Omp3 single

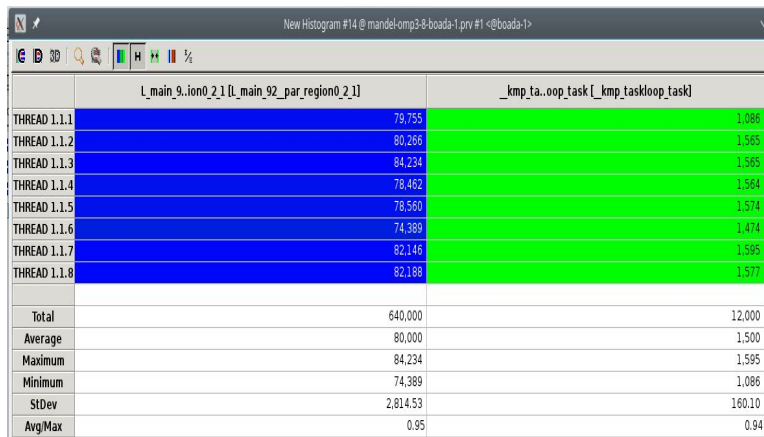


Figure 2.17: Omp3 #tasks

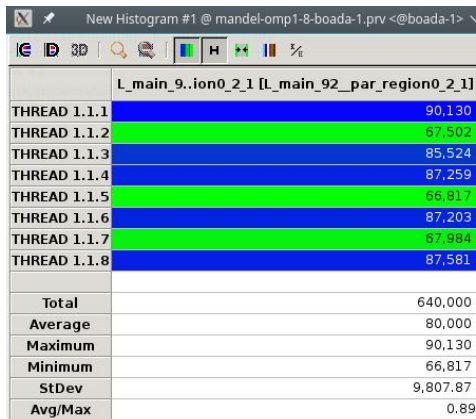


Figure 2.18 Omp3

3. Row decomposition in OpenMP

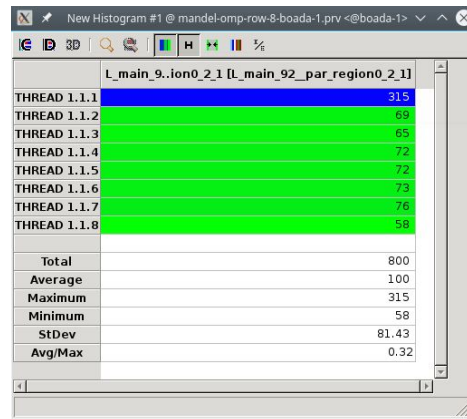
To use the row decomposition strategy we only need a taskloop before the outermost loop using a grainsize of height/num_threads.

We want to create a task for each block of iterations of the first loop (depending on the grain_size), because doing so we can reduce the overheads generated due to the creation of tasks. As we can see in the strong scalability plot, this decomposition is much better than the point one.



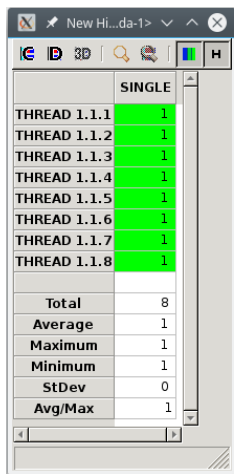
L_main_9..ion0_2_1 [L_main_92_par_region0_2_1]	
THREAD 1.1.1	90,130
THREAD 1.1.2	67,502
THREAD 1.1.3	85,524
THREAD 1.1.4	87,259
THREAD 1.1.5	66,817
THREAD 1.1.6	87,203
THREAD 1.1.7	67,984
THREAD 1.1.8	87,581
Total	640,000
Average	80,000
Maximum	90,130
Minimum	66,817
StDev	9,807.87
Avg/Max	0.89

Figure 3.1: Paraver #tasks 1-8



L_main_9..ion0_2_1 [L_main_92_par_region0_2_1]	
THREAD 1.1.1	315
THREAD 1.1.2	69
THREAD 1.1.3	65
THREAD 1.1.4	72
THREAD 1.1.5	72
THREAD 1.1.6	73
THREAD 1.1.7	76
THREAD 1.1.8	58
Total	800
Average	100
Maximum	315
Minimum	58
StDev	81.43
Avg/Max	0.32

Figure 3.2: Omp-row #tasks



SINGLE	
THREAD 1.1.1	1
THREAD 1.1.2	1
THREAD 1.1.3	1
THREAD 1.1.4	1
THREAD 1.1.5	1
THREAD 1.1.6	1
THREAD 1.1.7	1
THREAD 1.1.8	1
Total	8
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Figure 3.3: omp-row single

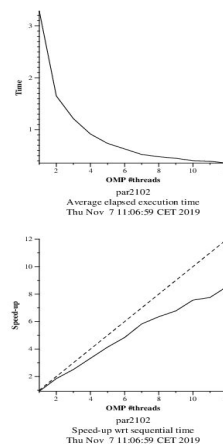


Figure 3.4: omp-row strong

4. For-based parallelization

We've decided to try three different scheduling for the for-based parallelization.

Static: as we can see in the strong scalability plot, this scheduling is pretty bad for this program, because the workload of each iteration is different.

Dynamic: this is the best scheduling, as we can see in the scalability plot.

Guided: this scheduling isn't too bad, but for a large number of threads it begins to fall down compared to the dynamic one.

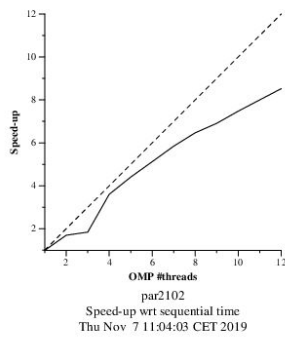
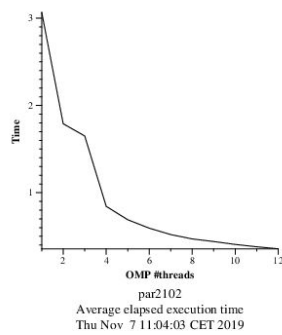


Figure 4.1: Dynamic strong

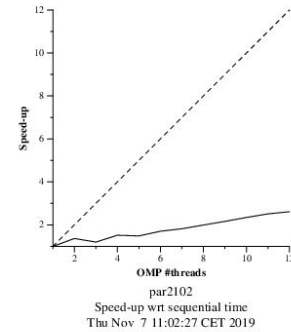
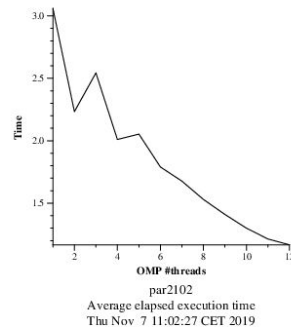


Figure 4.2: Static strong

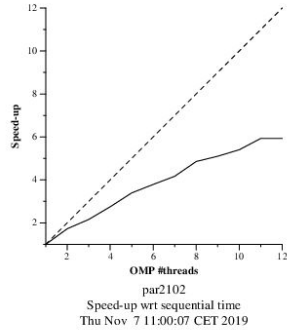
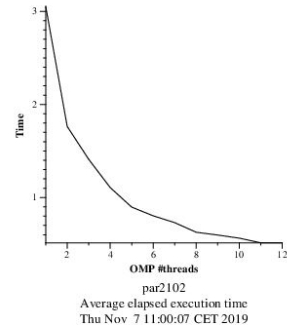


Figure 4.3: Guided strong

5. Conclusions

In this lab we've seen that the overheads can be a problem, so having lots of tasks doesn't mean that we are going to get better results.

First of all we've analyzed the taredor graphs to see how can we parallelize the sequential code.

After this first approach we've used a Point decomposition strategy to parallelize the code, toying with different omp clauses to obtain better results and then we've compared them with the Row decomposition strategy.

Finally we've observed the different results that we can get using different types of scheduling (using the `#pragma omp for schedule clause`).