Parallelism

Lab 2: Embarrassingly parallelism with OpenMP: Mandelbrot set

Marian Danci & David Valero
Group 42, par4207
Fall 2018-19

Indice

Session 1: Task decomposition analysis for the Mandelbrot set computation

1.1 The Mandelbrot set

The program used is the computation of the Mandelbrot set, a particular set of points, in the complex domain, whose boundary generates a two dimensional fractal shape. We followed the steps correctly, and we could generate our own Mandelbrot set.

1.2 Task decomposition analysis with Tareador

After trying the programs with different inputs, we analyzed, using Tareador, the potential parallelism for two possible task granularities that can be exploited in this program:

- Point: a task corresponds with the computation of a single point (row,col) of the Mandelbrot set.
- Row: a task corresponds with the computation of a whole row of the Mandelbrot set.

First of all, we computed the non-graphical versions of the two granularities, and we get these two dependency graphs.
The main common characteristics of both are that all the tasks that we had paralyzed don't have dependencies and they all can be computed parallel.

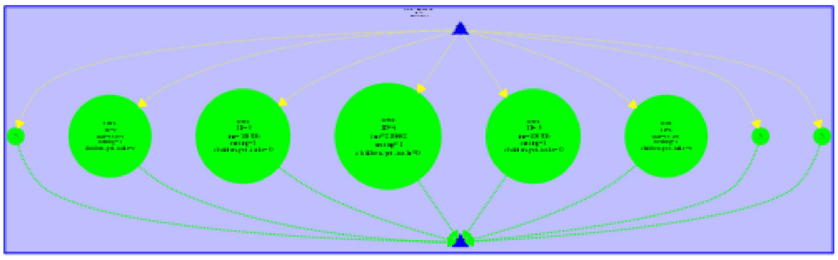


Illustration 1: Dependencies graphs by rows of the non-graphical version

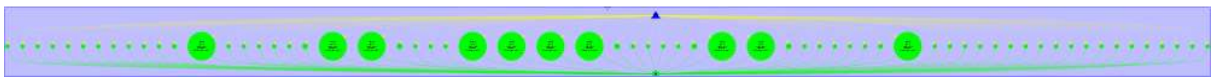


Illustration 2: Dependencies graphs by columns of the non-graphical version

After that, we computed the graphical versions of the two granularities, and we get these two dependency graphs.
We saw that both have a similar graph to the non-graphical version, but this time they can not paralyze all the task.
The main reason for that is because the program cannot execute the `XSetForeground (display, gc, color);` and the `XDrawPoint (display, win, gc, col, row);` instructions at the same time. To guarantee that these instructions are executed at the same time (it would break the program) we put the line `#pragma omp critical` before them.

mandel-tar.c:
```
/ * Scale color and display point * /
long color = (long) ((k-1) * scale_color) + min_color; //
color to calculate in function of the k
if (setup_return == EXIT_SUCCESS) {
     #pragma omp critical
     {
          XSetForeground (display, gc, color);
          XDrawPoint (display, win, gc, col, row);
     }
}
```

Because of on the real screen 256 and 65,000 (256 * 256). Graphs 8 and 64. To avoid better overheads 256 (per rows). For the number of processors we have at the boada (24), better 256. If we have one thousands 65,000.

Illustration 3: Dependencies graphs by rows of the graphical version

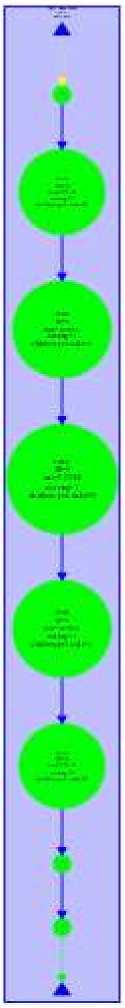


Illustration 4: Dependencies graphs by columns of the graphical version

Session 2: Point decomposition in OpenMP

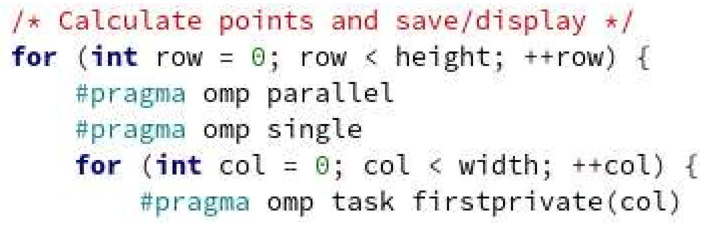
2.1 Tasking model in *OpenMP*

As we analysed, the tasking model extends the basic fork–join model by allowing threads to generate tasks to be executed by other threads.
A task is a unit of work that can be instantiated by a thread for later execution. Instantiated tasks are placed in the so–called task pool, and can be extracted from the pool by either the same thread that instantiated them or by a different thread in the team.
After working with the Tareador, we analysed the program using *OpenMP.*

2.2 *Point strategy* implementation

First of all, we change the program in order to use a parallel construct, where the team of threads is created in each iteration of the row loop.
Immediately after that, if we want to have a single task generator then we have to use the #pragma omp single work–distributor in OpenMP. This construct allows only one of the threads in the team to traverse the iteration space of the col loop; this means that the thread that enters the single construct will generate all the tasks, one for each iteration of the col loop, as expressed with the task construct mentioned before. As indicated in the task pragma, each thread gets a private copy of variable col initialised with the value it has at task creation time (firstprivate clause). The rest of threads in the team that do not enter into the single construct simply wait at the implicit barrier at the end of the construct. While waiting, they will continuously check the availability of tasks in the task pool and, if available, executed them. When one of the tasks is extracted from the pool, it has the values for variables row (shared variable by default) and col (privatised and initialised at task creation time) necessary for the computation of a specific point in the Mandelbrot set. When the thread that entered the single construct finishes with the generation of all tasks, then it will also contribute to their execution while waiting at the implicit barrier. Once all tasks are finished, threads will leave the implicit barrier and continue execution.

At the first version, we edit the initial task version in mandel-omp.c, inserting the missing *OpenMP* directives to make sure that the dependencies you detected for the graphical version in the previous section are honoured.



```
/* Calculate points and save/display */
for (int row = 0; row < height; ++row) {
    #pragma omp parallel
    #pragma omp single
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(col)
```

Illustration 5: Version 1, parallel per row.

Secondly, we use the mandeld-omp target in the Makefile to compile and generate the parallel binary for the graphical version.
We execute it 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 ./mandeld-omp -i 10000).
The image is generated correctly but it takes 3.3 seconds to execute, it takes more the *OpenMP* version since it loses time configuration of #pragma and is defined with a thread cannot take advantage of the parallelization, it's as if it fuses sequentially.
When we execute it with 8 threats (i.e. OMP_NUM_THREADS=8 ./mandeld-omp -i 10000) it only spends around 1 second to execute it because of it take advantage of the parallel part of the program that calculates the points of the loop.

After that, we use the mandel-omp target in the Makefile to compile and generate the parallel binary for the non-graphical version. We execute this version with 1 and 8 threads, seeing that the output file generated is identical in both the sequential and parallel version. We submit the submit-strong-omp.sh script with qsub to execute the binary generated and we obtain the execution time and speed–up plots.
For one threat it spends 3.3s and for 8, only 1s, the time is reduced when we increase the number of threads, but in a certain time, the overhead is so big that it doesn't increase much.

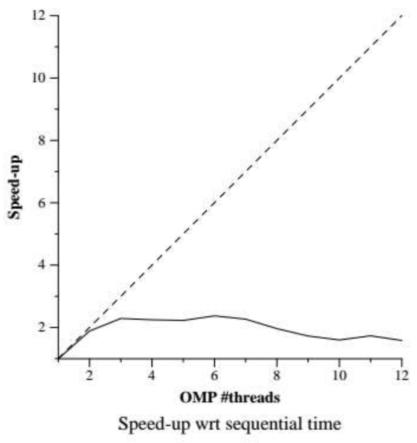
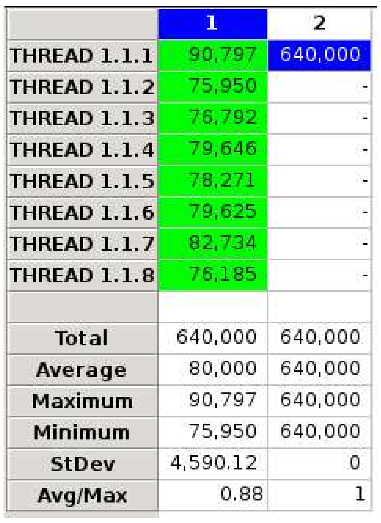


Illustration 6: Strong scalibility, version 2.

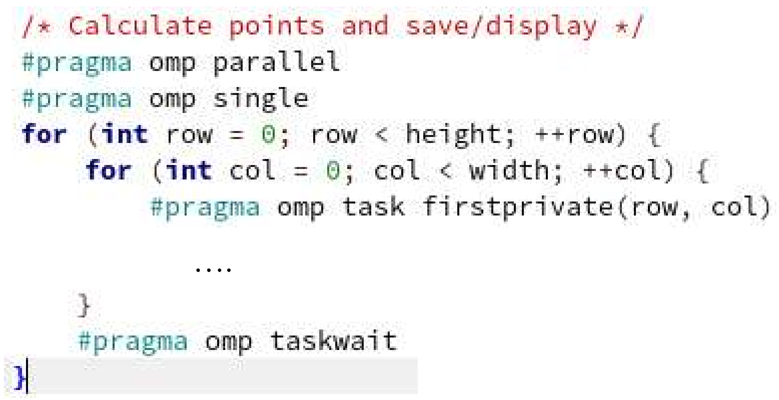


	1	2
THREAD 1.1.1	90,797	640,000
THREAD 1.1.2	75,950	-
THREAD 1.1.3	76,792	-
THREAD 1.1.4	79,646	-
THREAD 1.1.5	78,271	-
THREAD 1.1.6	79,625	-
THREAD 1.1.7	82,734	-
THREAD 1.1.8	76,185	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	90,797	640,000
Minimum	75,950	640,000
StDev	4,590.12	0
Avg/Max	0.88	1

In order to better understand how the execution goes, we do an Extrae instrumented execution, opening the trace generated with Paraver. For now, we open OMP tasks.cfg to visualize when tasks are created and executed.
We see that it create one task per point, and threat 1 creates all, that's why it spends more time executing itself than the others. The parallel construct is invoked one per row as the single worksharing construct.

Table 1: Paraver Mandel omp with 8 threads profile configuration of Version1.

We observe that now only one thread, the one 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 (point).

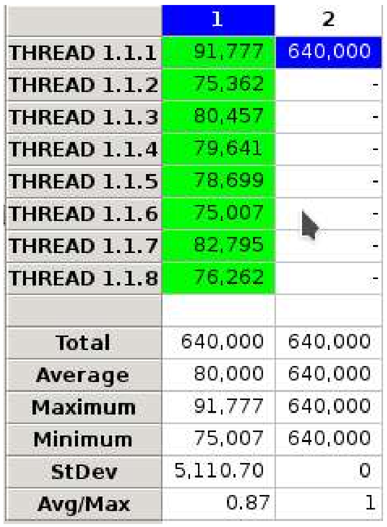


```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)

            ….
    }
    #pragma omp taskwait
}
```

Illustration 7: Version 2, parallel + taskwait

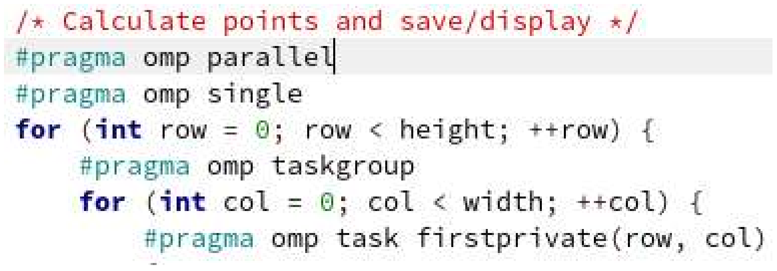
We have introduced the #pragma omp taskwait synchronization construct, which defines a point in the program to wait for the termination of all child tasks generated up to that point. In this case, the thread will wait until all tasks for each one of the rows finish; after that, the thread will advance one iteration of the row loop and generate a new bunch of tasks. We modify the code in mandel-omp.c to reflect this change and we repeat the previous evaluation (scalability and tracing).
As same as the previews implementation the first thread creates all the tasks and the parallel construct is invoked one per point and the single worksharing construct and the taskwait construct are invoked one per row.
It creates on task per point. The granularity of tasks has changed because the 1st threads that create the tasks have to wait for all child tasks.



	1	2
THREAD 1.1.1	91,777	640,000
THREAD 1.1.2	75,362	-
THREAD 1.1.3	80,457	-
THREAD 1.1.4	79,641	-
THREAD 1.1.5	78,699	-
THREAD 1.1.6	75,007	-
THREAD 1.1.7	82,795	-
THREAD 1.1.8	76,262	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	91,777	640,000
Minimum	75,007	640,000
StDev	5,110.70	0
Avg/Max	0.87	1

Table 2: Paraver Mandel omp with 8 threads profile configuration of Version 2.

Alternatively to the use of taskwait one can use the #pragma omp taskgroup construct, which defines a region in the program, at the end of which the thread will wait for the termination of all descendant (not only child) tasks.
We see that the taskwait construct is not really necessary, because it is not necessary to wait for the termination of all tasks in a row before generating the tasks for the next rows.
Also, we see that the number of tasks created/executed doesn't change.

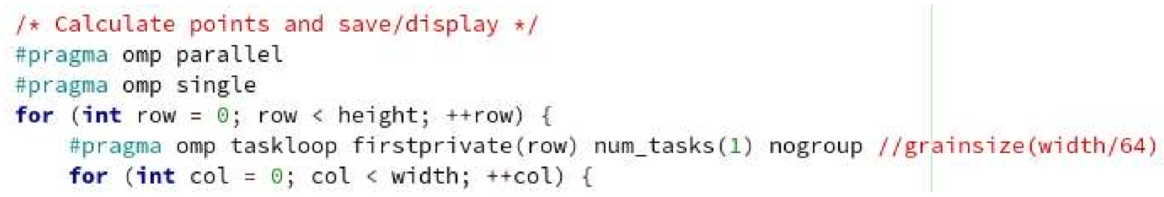


```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskgroup
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
```

Illustration 8: Version 3, parallel

2.3 Granularity control with `taskloop`

Finally, we 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. To control the number of tasks generated out of the loop we use the num tasks clause, with the appropriate number to specify the number of tasks; to control the granularity of tasks we use the grain size clause.



```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(1) nogroup //grainsize(width/64)
    for (int col = 0; col < width; ++col) {
```

Illustration 9: Version 4, taksloop

We edit the last version in mandel-omp.c in order to make use of taskloop. We use the mandeld-omp target in the Makefile to compile and generate a parallel binary for the graphical version. Interactively we execute it to see that the code produces the appropriate result.
The new version based on taskloop performs better than the version based on task, due to in this one we do not create a new task per point, we fixed the number of tasks, so almost we do not have overhead

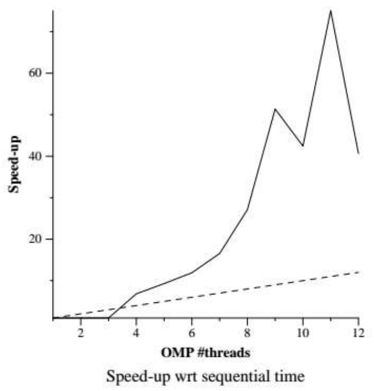


Illustration 10: Strong scalibility, version 4.

The taskloop construct accepts a nogroup clause that eliminates the implicit task barrier (taskgroup) at the end and task barrier can't be eliminated, so nogroup clause has no effect in taskloop.

Finally, we explore how the new version behaves in terms of performance when using 8 threads in execution queue (boada-2 to boada-4) and for different task granularities, we see that….

num_tasks(800):
3.77user 0.14system 0:00.51elapsed 761%CPU (0avgtext+0avgdata 7216maxresident)k 88inputs+8outputs (1major+1104minor)pagefaults 0swaps
num_tasks(400):
3.80user 0.08system 0:00.51elapsed 760%CPU (0avgtext+0avgdata 7208maxresident)k 88inputs+8outputs (1major+1110minor)pagefaults 0swaps
num_tasks(200):
3.73user 0.05system 0:00.49elapsed 758%CPU (0avgtext+0avgdata 7176maxresident)k 88inputs+8outputs (1major+1106minor)pagefaults 0swaps
num_tasks(100):
3.72user 0.03system 0:00.49elapsed 760%CPU (0avgtext+0avgdata 7192maxresident)k 88inputs+8outputs (1major+1096minor)pagefaults 0swaps
num_tasks(50):

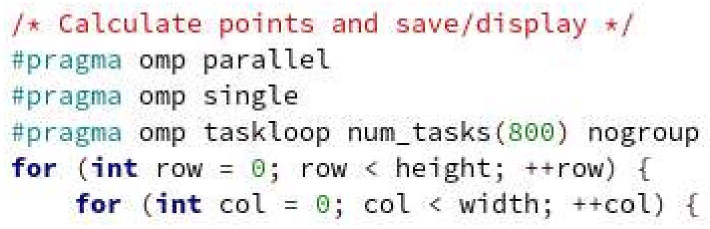
3.79user 0.00system 0:00.49elapsed 761%CPU (0avgtext+0avgdata 6928maxresident)k 88inputs+8outputs (1major+1009minor)pagefaults 0swaps
num_tasks(25):
3.76user 0.01system 0:00.49elapsed 760%CPU (0avgtext+0avgdata 6848maxresident)k 88inputs+8outputs (1major+1003minor)pagefaults 0swaps
num_tasks(10):
3.68user 0.00system 0:00.49elapsed 741%CPU (0avgtext+0avgdata 6784maxresident)k 88inputs+8outputs (1major+1001minor)pagefaults 0swaps
num_tasks(5):
3.70user 0.00system 0:00.48elapsed 758%CPU (0avgtext+0avgdata 6800maxresident)k 88inputs+8outputs (1major+1005minor)pagefaults 0swaps
num_tasks(2):
3.73user 0.00system 0:00.49elapsed 757%CPU (0avgtext+0avgdata 6792maxresident)k 88inputs+8outputs (1major+1007minor)pagefaults 0swaps
num_tasks(1):
3.57user 0.00system 0:00.47elapsed 759%CPU (0avgtext+0avgdata 6868maxresident)k 88inputs+8outputs (1major+1003minor)pagefaults 0swaps

Table 3: Results of the new version behaves in terms of performance when using 8 threads and for different task granularities.

Session 3: Row decomposition in OpenMP

In this session, we wrote a new parallel version that follows a Row strategy, consisting of create a new task for each row of the matrix. The previous session we implemented differents ways of point strategy. This one, we'll work with one of them, with the aim of comparing both strategies.

In the following code we making use of taskloop. Taskloop construct generates tasks out of the iterations of a for loop, allowing to control the number of tasks generated (`num_tasks()`) or their granularity (`grainsize()`). *num_tasks* control the number of tasks generated out of the loop, specifying the number of tasks. *Grainsize* control the granularity of tasks specifying the number of iterations per task.
The nogroup clause, eliminates the implicit task barrier at the end.



```
/* Calculate points and save/display */
#pragma omp parallel
#pragma omp single
#pragma omp taskloop num_tasks(800) nogroup
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
```

Illustration 11: Row descomposition, taskloop

We used submit-strong-omp.sh to evaluate the strong scalability of mandel-omp.c, this script executes the program with different numbers of threads and makes two plots, one with speed-up and the other with time, both depending on the number of threads.

In the plot, we can see that row decomposition has a smaller overhead, due to there are not so many tasks to manage, than the point strategy. So the taskloop differences between row and point decomposition are not significant, because each one specific the number of tasks, but with the other implementations of points there is a huge speed-up improve.

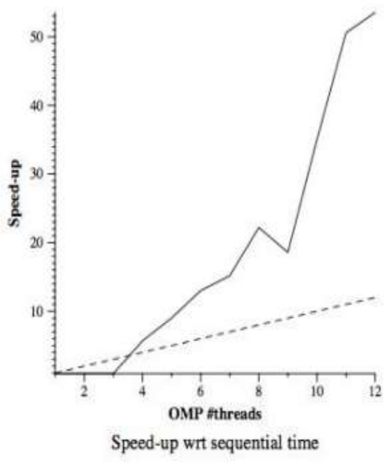


Illustration 12: Strong scalability, taskloop speed-up