DELIVERABLE

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

UPC - EPSEVG
Antoni Cifre | Eva Hernández
Grupo: paco0013

Fecha: 19/11/2019
3r año, 1r cuatrimestre

Contenido

Task decomposition analysis with Tareador 2
2 Point decomposition in OpenMP 5
    2.2 Granularity control with taskloop 12
3 Row decomposition in OpenMP 20

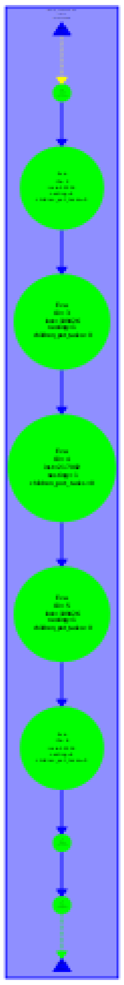
Task decomposition analysis with Tareador

1. Complete the sequential mandel-tar.c code partially instrumented in order to individually analyze the potential parallelism when tasks are defined at the two granularity levels indicated (Point or Row). For each granularity, compile the instrumented code using the two targets in the Makefile that generate the non–graphical and graphical versions of the program.

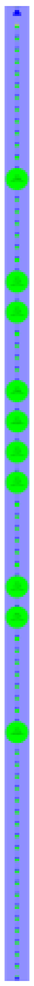
```
void mandelbrot(int height, int width, double real_min, double imag_min,
                double scale_real, double scale_imag, int maxiter,
#if _DISPLAY_
                int setup_return,Display *display, Window win,
                GC gc, double scale_color, double min_color)
#else
                int ** output)
#endif
{

    for (row = 0; row < height; ++row) {
         tareador_start_task("Tareador para Fila");
         for (col = 0; col < width; ++col) {
            tareador_start_task("Tareador para Punto");

            complex z, c;
            z.real = z.imag = 0;
            c.real = real_min + ((double) col * scale_real);
            c.imag = imag_min + ((double) (height-1-row) * scale_imag);
            int k = 0;
            double lengthsq, temp;
            do  {
                temp = z.real*z.real - z.imag*z.imag + c.real;
                z.imag = 2*z.real*z.imag + c.imag;
                z.real = temp;
                lengthsq = z.real*z.real + z.imag*z.imag;
                ++k;
            } while (lengthsq < (N*N) && k < maxiter);

#if _DISPLAY_
            long color = (long) ((k-1) * scale_color) + min_color;
            if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
            }
#else
            output[row][col]=k;
#endif
            tareador_end_task("Tareador para Punto");
        }
        tareador_end_task("Tareador para Fila");
    }

}
```

2. Interactively execute both mandeld-tar and mandel-tar using the ./run-tareador.sh script, indicating the name of the instrumented binary to be executed. The size of the image to compute (option -w) is defined inside the script (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, look for the small Mandelbrot window that will be opened when using mandeld-tar and click inside in order to finish its execution).

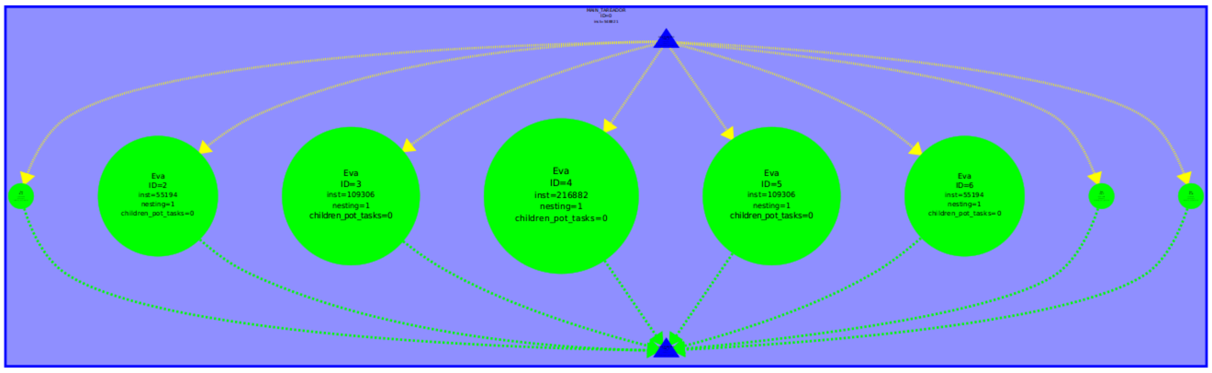
Granularity row with display



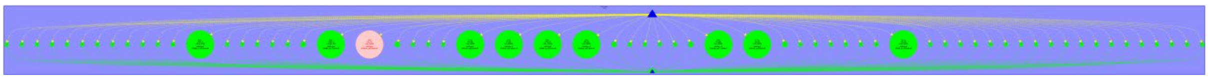
Granularity point with display



Granularity row without display



Granularity point without display



3. Which are the two most important common characteristics of the task graphs generated for the two task granularities (Point and Row) for the non-graphical version of mandel-tar? Why the task graphs generated for mandel-tar and mandeld-tar 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?

Las dos características son: las tareas son muy paralelizables y algunas tareas tienen más peso que otras.

Son tan diferentes porque en mandeld-tar hay dependencia de datos con la variable 11x-COLOR y esas tareas no se pueden paralelizar.

La sección de código que lo provoca es:

```
#if _DISPLAY_
            long color = (long) ((k-1) * scale_color) + min_color;
            if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
            }
#else
```

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

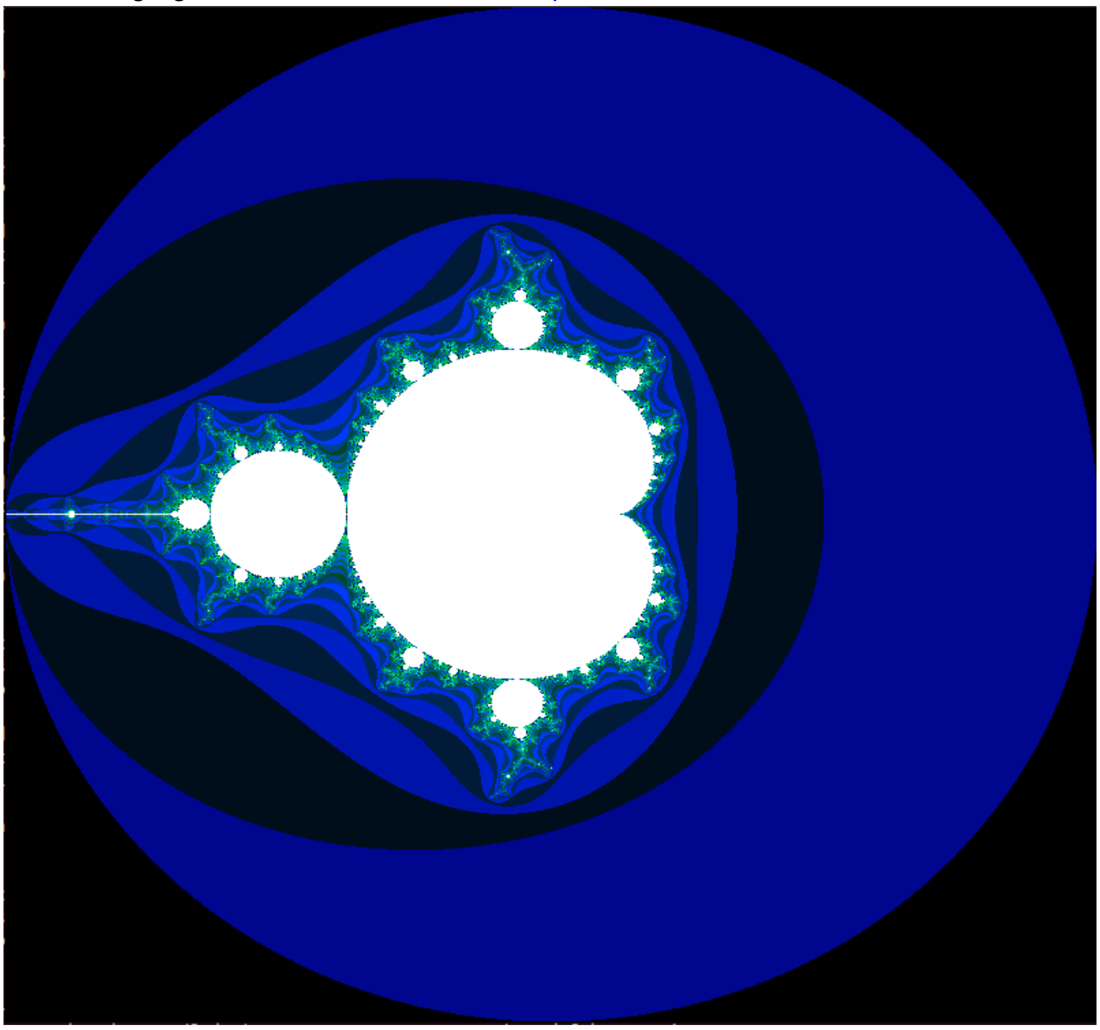
Se protegerá creando un #pragma omp critical

4. Reason which one of the two task granularities would be more appropriate to apply to implement a parallel version of the Mandelbrot code.

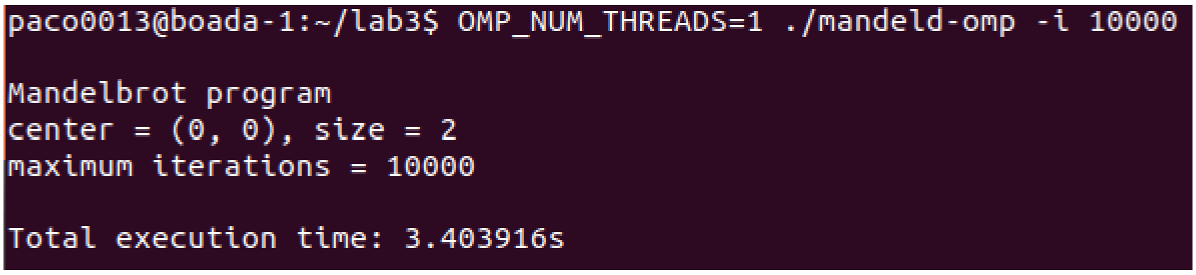
tareador-disable_object(& name_var)
….
tareador_enable_object(&name_var)

2 Point decomposition in OpenMP

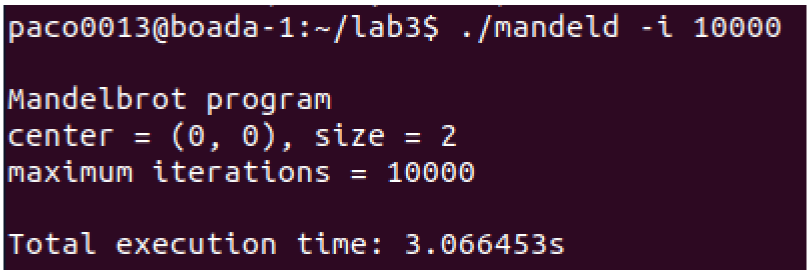
2. Use the mandeld-omp target in the Makefile to compile and generate the parallel binary for the graphical version. Interactively execute it to see what happens when running it with only 1 thread and a maximum of 10000 iterations per point (i.e. OMP NUM THREADS=1 ./mandeld-omp -i 10000).
Is the image generated correct? Como se puede observar, si es correcta.



Compare the execution time with the sequential version (i.e. ./mandeld -i 10000). Why the OpenMP version takes more time to execute?



```
paco0013@boada-1:~/lab3$ OMP_NUM_THREADS=1 ./mandeld-omp -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 3.403916s
```



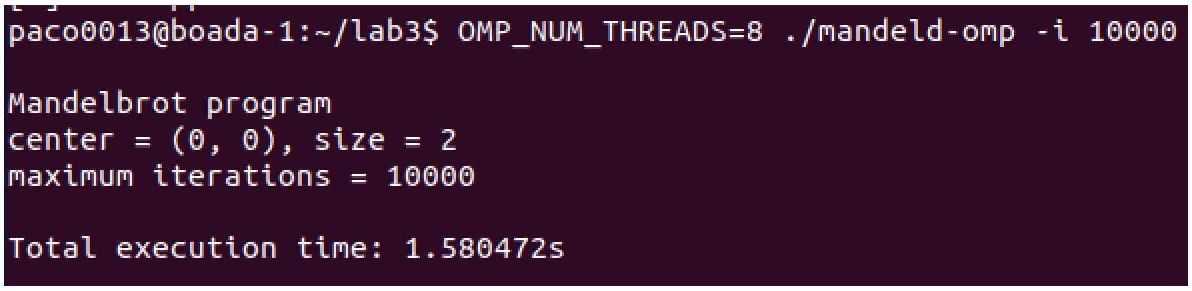
```
paco0013@boada-1:~/lab3$ ./mandeld -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 3.066453s
```

Tarda 0.34 segundos más porque openmp utiliza thread.

Interactively execute it with 8 threads and the same number of iterations per point (e.g. OMP NUM THREADS=8 ./mandeld-omp -i 10000). Is the image generated correct?

Sí, es correcta ya que con anterioridad hemos añadido la sección critica

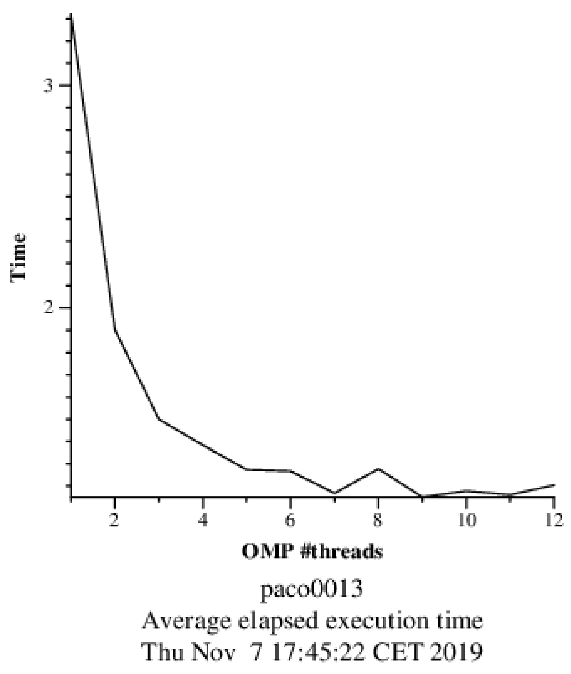
Is the execution time reduced?
Como se puede observar, si que se ha reducido el tiempo de ejecución en comparación a las anteriores.

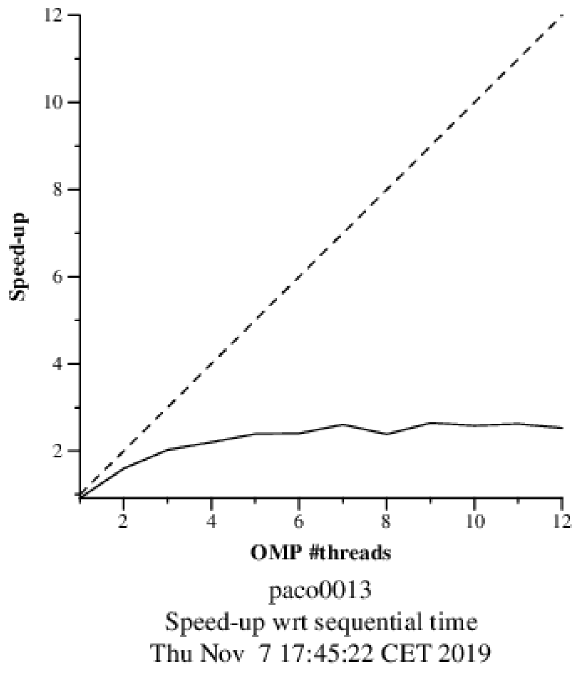


```
paco0013@boada-1:~/lab3$ OMP_NUM_THREADS=8 ./mandeld-omp -i 10000

Mandelbrot program
center = (0, 0), size = 2
maximum iterations = 10000

Total execution time: 1.580472s
```

3. Use the mandel-omp target in the Makefile to compile and generate the parallel binary for the non-graphical version. Execute this version with 1 and 8 threads, making sure that the output file generated (use the -o option) is identical in both the sequential and parallel version. Once verified, submit the submit-strong-omp.sh script with qsub to execute the binary generated and obtain the execution time and speed–up plots. Visualise the PostScript file generated. Is the scalability appropriate?

Como se puede observar, la escalabilidad de este código es muy mala.

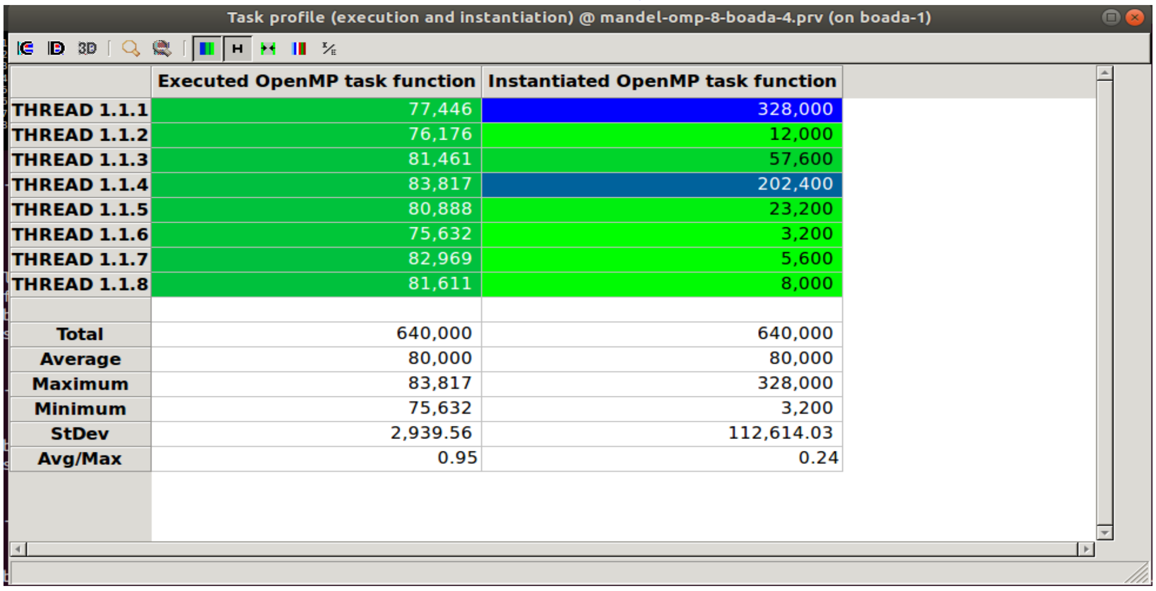




mandel omp

4. In order to better understand how the execution goes, do an Extrae instrumented execution, opening the trace generated with Paraver . The instrumented execution and trace loading will take some time, please be patient. Table 2.1 adds some new configuration files to the ones you used during the first laboratory assignment to do this tasking analysis, all of them inside the cfgs/OMP tasks directory.
For now open OMP tasks.cfg to visualize when tasks are created and executed. How many tasks are created/executed and which thread(s) is/are creating/executing them?

Como hemos indicado, se han creado 8 threads y 640000 tasques entre los 8 threads.


Task profile (execution and instantiation) @ mandel-omp-8-boada-4.prv (on boada-1)

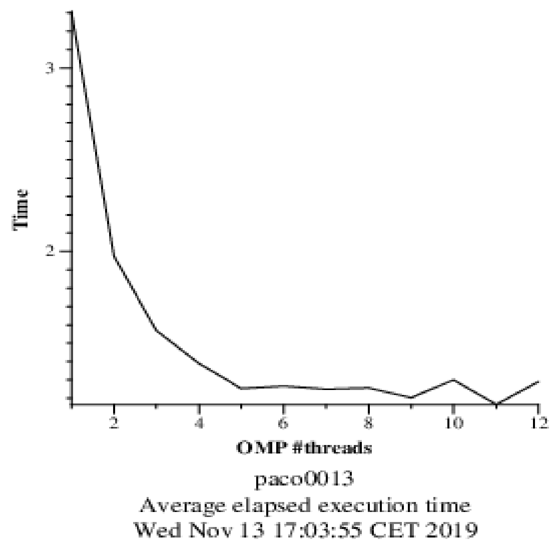
	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	77,446	328,000
THREAD 1.1.2	76,176	12,000
THREAD 1.1.3	81,461	57,600
THREAD 1.1.4	83,817	202,400
THREAD 1.1.5	80,888	23,200
THREAD 1.1.6	75,632	3,200
THREAD 1.1.7	82,969	5,600
THREAD 1.1.8	81,611	8,000
Total	640,000	640,000
Average	80,000	80,000
Maximum	83,817	328,000
Minimum	75,632	3,200
StDev	2,939.56	112,614.03
Avg/Max	0.95	0.24

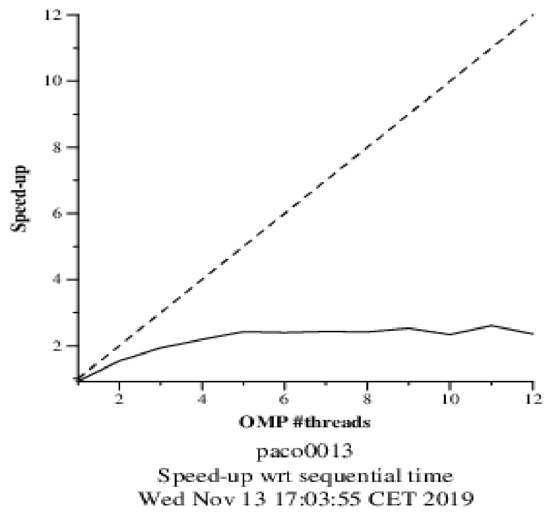
Figure 1 mandel omp o threads

How many times the parallel construct is invoked?
Se ha invocado 800 veces, definida por la variable height.
How many times the single worksharing construct is invoked?
800 veces, igual que la zona paralela.
How do you obtain these numbers?
Analizando el código podemos observar que la zona paralela y el singel están dentro del primer for, el cual se ejecuta “height” veces.
Si analizamos el código podremos observar que la variable height vale 800.
También se podría obtener esta información utilizando paraver y aplicando la configuración OMP_task_profile.

If you change the previous parallelization for Point to the code shown in Figure 2.2, what would it change?
En la figura 2.2 se saca la creación de la zona paralela i el omp single fuera del for, de tal forma que la zona paralela y el single solo será invocado una sola vez.

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). 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. Modify the code in mandel-omp.c to reflect this change and repeat the previous evaluation (scalability and tracing).





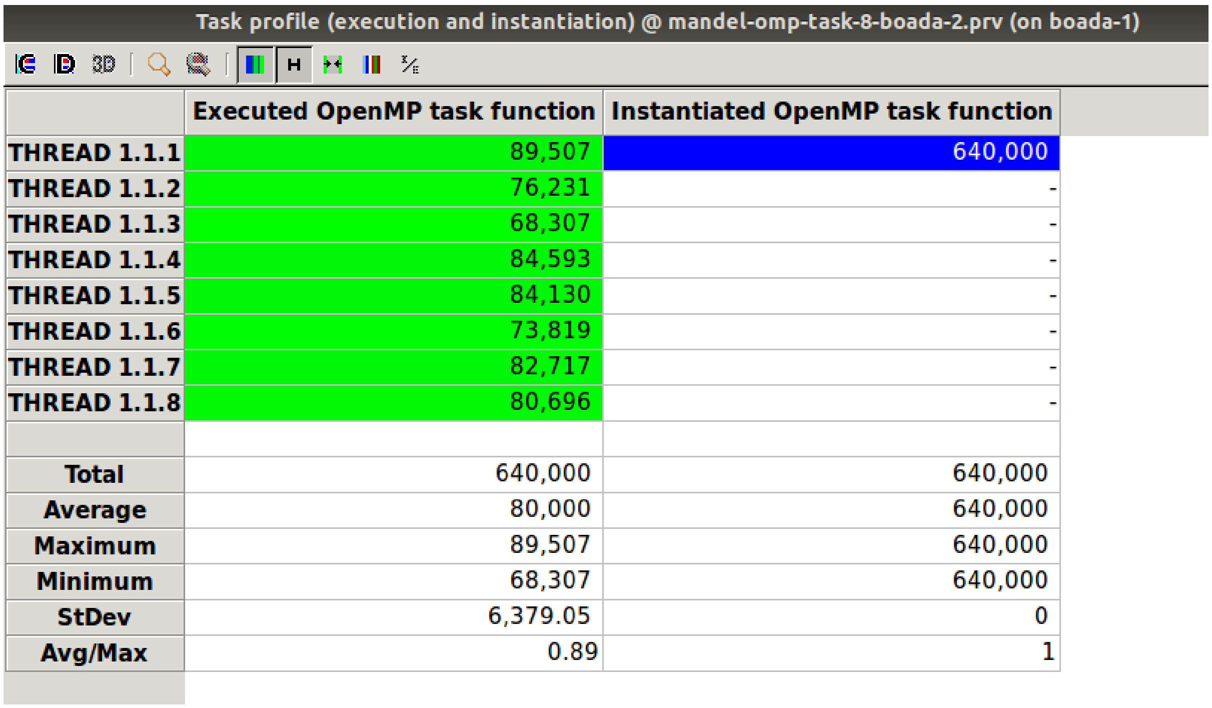
mandel omp task

Comparado con la escalabilidad sin añadir #pragma omp taskwait podemos decir que son prácticamente iguales. Esto es debido a que aplicamos una barrera de espera (task wait), pero en mandel-omp ya existe una barrera implícita del single, de tal modo que no existe ninguna diferencia entre los dos, la única diferencia sería en número de zonas paralelas generadas.

How many times the parallel construct is now invoked? How many times the single worksharing construct is now invoked?
Tanto la zona paralela como el omp single solo serán invocados una sola vez, debido a que su llamada se ha sacado fuera del bucle y solo será llamado una sola vez por el primer bucle.

How many times the taskwait construct is executed?
Será ejecutada 640000 veces

How many tasks are created/executed and which thread(s) is/are creating/executing them?


Task profile (execution and instantiation) @ mandel-omp-task-8-boada-2.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,507	640,000
THREAD 1.1.2	76,231	-
THREAD 1.1.3	68,307	-
THREAD 1.1.4	84,593	-
THREAD 1.1.5	84,130	-
THREAD 1.1.6	73,819	-
THREAD 1.1.7	82,717	-
THREAD 1.1.8	80,696	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	89,507	640,000
Minimum	68,307	640,000
StDev	6,379.05	0
Avg/Max	0.89	1

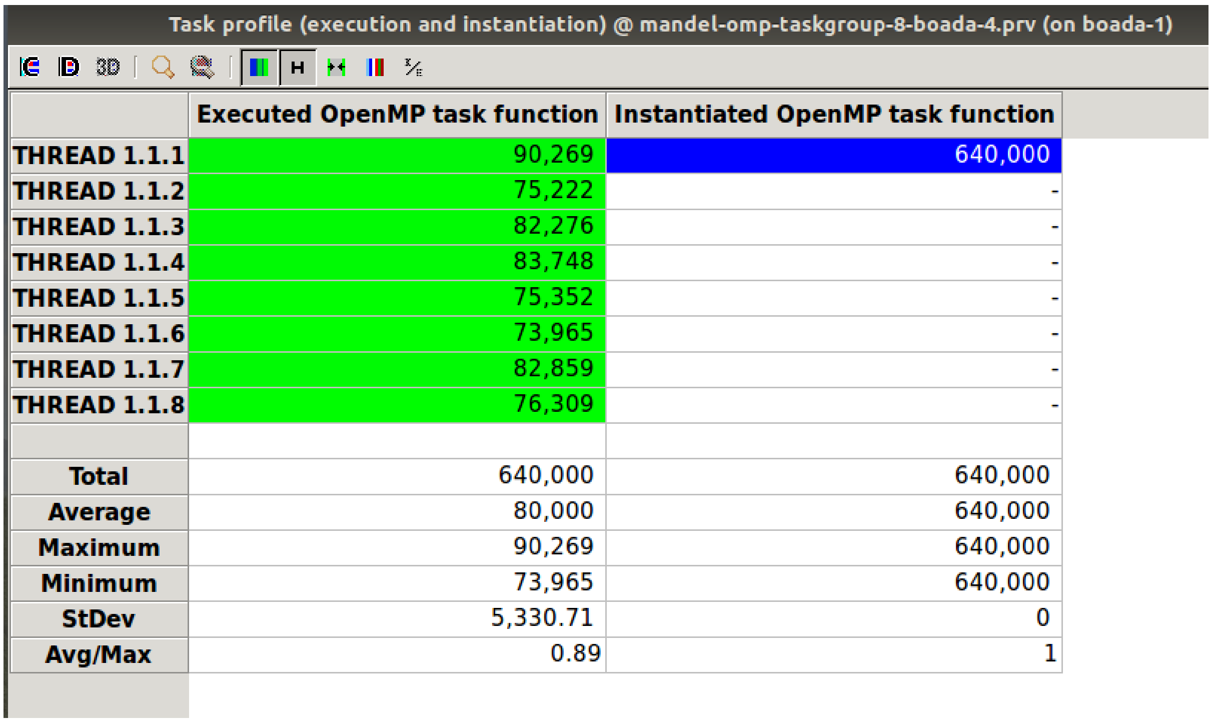
Figure 2 mandel omp task o threads

El este caso, thread 1.1.1 es el encargado de repartir la 640000 tareas entre los 8 threads disponibles (incluido el mismo).

Has the granularity of tasks changed?
Podemos deducir que la granularidad no ha cambiado debido a que se generan el mismo número de tareas. (una por cada iteración del bucle mas interno)

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, as shown in Figure 2.3.
Do you observe any difference in the behaviour and timing when this task synchronization construct is used?

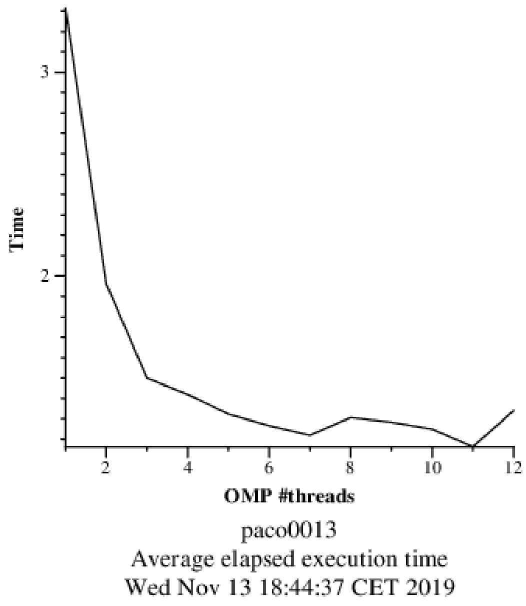
Si comparamos el task profle y las gráficas de escalabilidad, no se puede observar ninguna diferencia significativa.

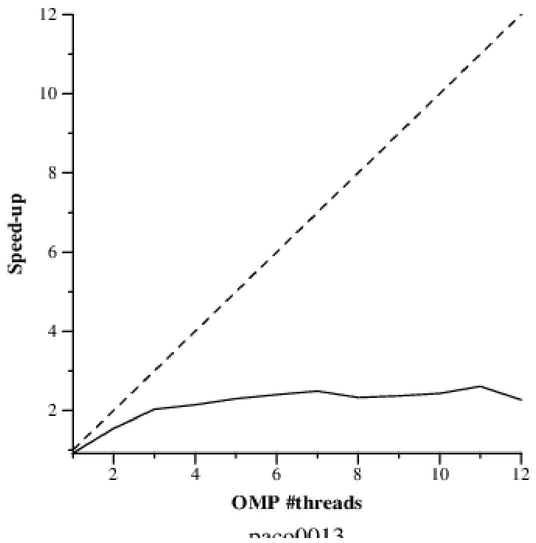


Task profile (execution and instantiation) @ mandel-omp-taskgroup-8-boada-4.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	90,269	640,000
THREAD 1.1.2	75,222	-
THREAD 1.1.3	82,276	-
THREAD 1.1.4	83,748	-
THREAD 1.1.5	75,352	-
THREAD 1.1.6	73,965	-
THREAD 1.1.7	82,859	-
THREAD 1.1.8	76,309	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	90,269	640,000
Minimum	73,965	640,000
StDev	5,330.71	0
Avg/Max	0.89	1

Figure 3 mandel omp taskgroup 8 threads



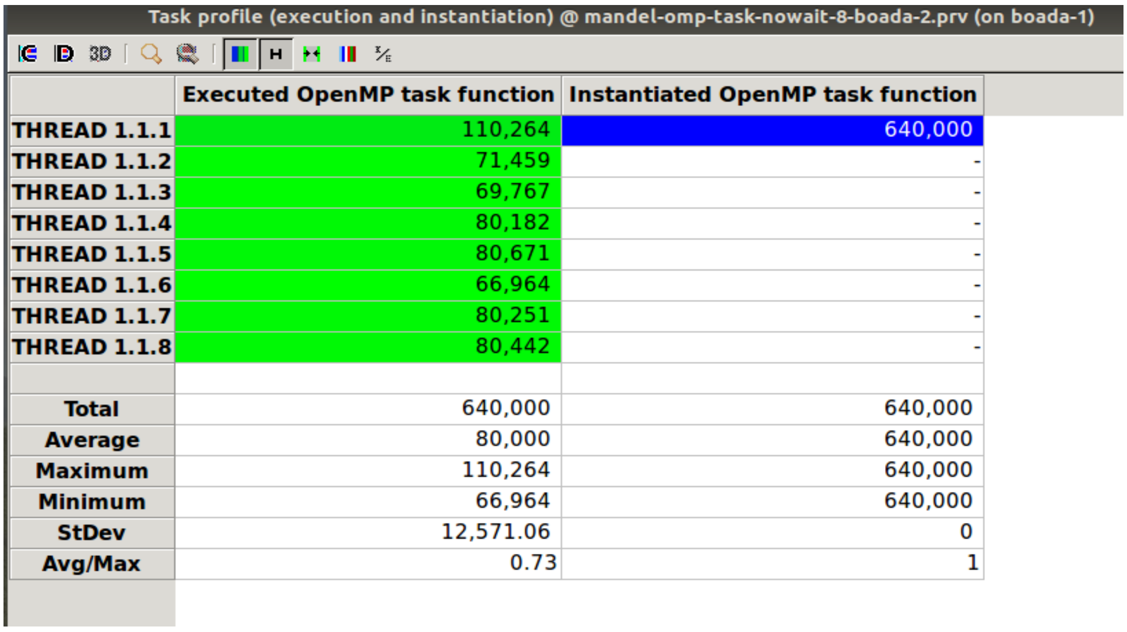


Do you think the taskwait construct in Figure 2.2 is really necessary? Or in other words, is it necessary to wait for the termination of all tasks in a row before generating the tasks for the next rows?

No es necesario, debido a que protegemos la zona crítica para evitar conflictos.

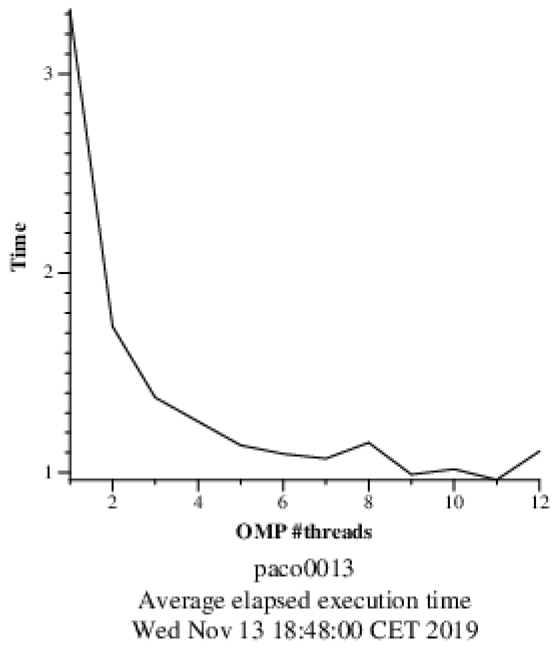
Además lo hemos comprobado mostrando el dibujo y podemos observar que el resultado es el mismo.

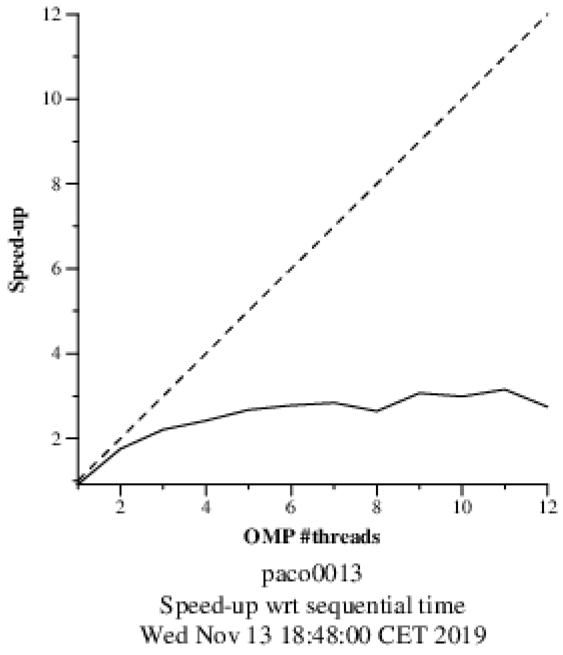
Modify the code in mandel-omp.c to remove this task barrier and repeat the previous evaluation (scalability and tracing).


Task profile (execution and instantiation) @ mandel-omp-task-nowait-8-boada-2.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	110,264	640,000
THREAD 1.1.2	71,459	-
THREAD 1.1.3	69,767	-
THREAD 1.1.4	80,182	-
THREAD 1.1.5	80,671	-
THREAD 1.1.6	66,964	-
THREAD 1.1.7	80,251	-
THREAD 1.1.8	80,442	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	110,264	640,000
Minimum	66,964	640,000
StDev	12,571.06	0
Avg/Max	0.73	1

Figure 4 mandel omp task notwait





Hay una ligera mejora en la escalabilidad.

Has the number of tasks created/executed changed

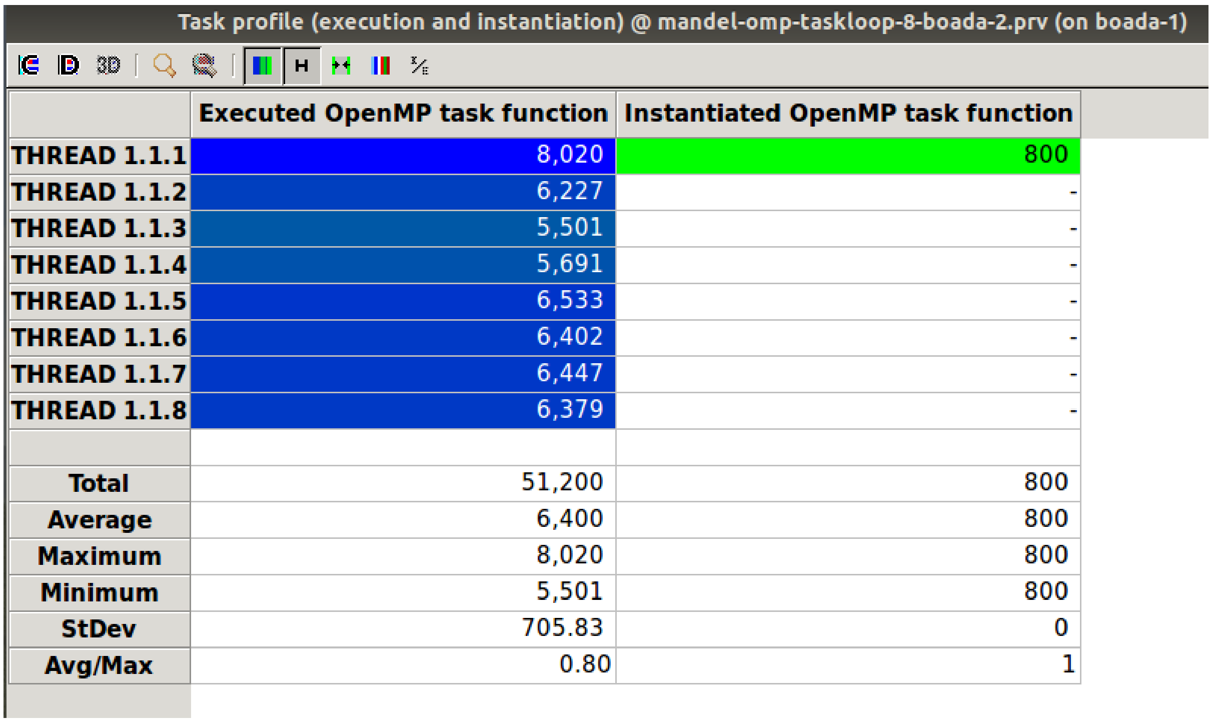
No, obviamente son las mismas, ya que solo hemos eliminado la barrera del taskwait.

Why the threads generating tasks stops task generation, then executes some tasks, and then proceeds generating new tasks?

Esto se debe a que el thread 1.1.1 reparte todas las tareas dentro del segundo for, luego las ejecuta, y hasta que no termina de ejecutar las tareas que el mismo se ha designado no vuelve a repartir las tareas.

2.2 Granularity control with taskloop

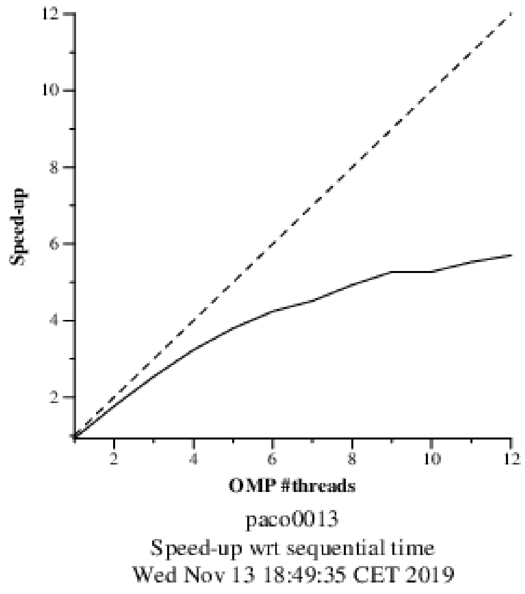
3. Use the mandel-omp target in the Makefile to compile and generate parallel binary for the nongraphical version. For grainsize(1) (or equivalently for num tasks(800)), which is equivalent to the second task version that you analyzed, why the new version based on taskloop performs better than the version based on task? In order to justify your answer, do an Extrae instrumented execution, opening the trace generated with Paraver and using the appropriate configuration files.

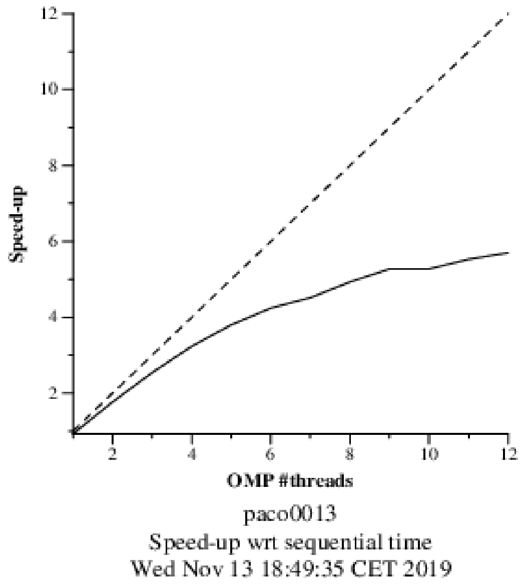

Task profile (execution and instantiation) @ mandel-omp-taskloop-8-boada-2.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	8,020	800
THREAD 1.1.2	6,227	-
THREAD 1.1.3	5,501	-
THREAD 1.1.4	5,691	-
THREAD 1.1.5	6,533	-
THREAD 1.1.6	6,402	-
THREAD 1.1.7	6,447	-
THREAD 1.1.8	6,379	-
Total	51,200	800
Average	6,400	800
Maximum	8,020	800
Minimum	5,501	800
StDev	705.83	0
Avg/Max	0.80	1

Figure 5 mandel omp tanskloop

En los ejemplos anteriores, se creaba una tarea por cada iteración interna del bucle, mientras que el taskloop lo que hace es repartir el bucle directamente entre un grupo determinado de threads, Como podemos observar en la tabla, sólo se generan 800 tareas, es decir, que cada vez que generamos una tarea, le estamos asignando una parte del bucle interno a un determinado thread, en lugar de crear una tarea por cada iteración del bucle interno.
Gracias a esta considerable reducción de generación de tareas se reduce también el overhead de creación de tareas y por lo tanto el tiempo mejora respecto a las versiones anteriores.

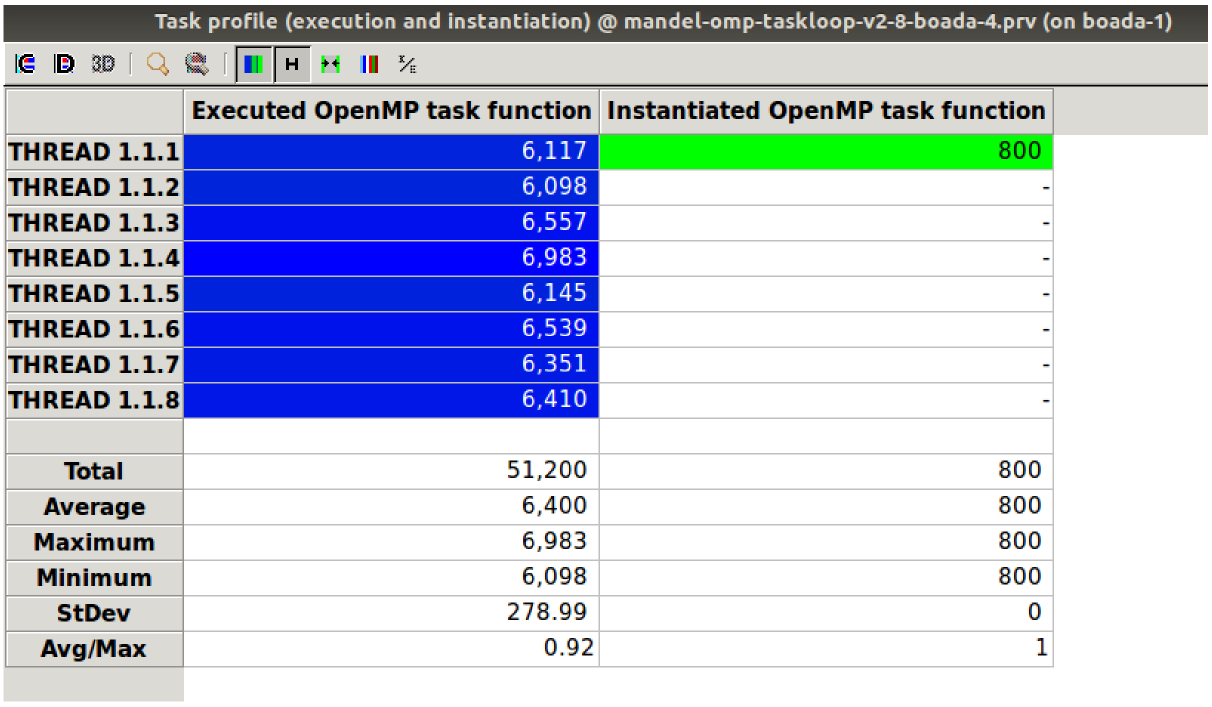




Comparando esta escalabilidad con las demás mejora muchísimo acercándose bastante a la escalabilidad ideal.

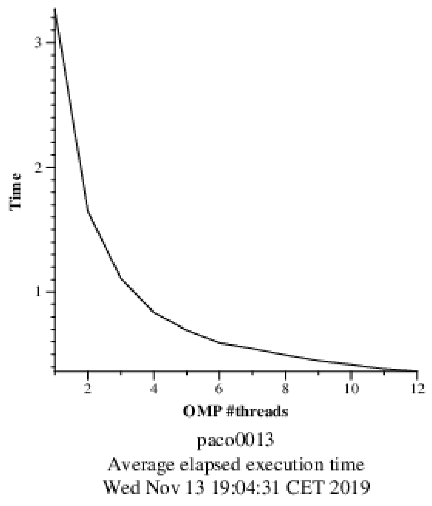
The taskloop construct accepts a nogroup clause that eliminates the implicit task barrier (taskgroup) at the end. Do you think this task barrier can be eliminated? If so, edit the code and insert it before doing the performance evaluation below.

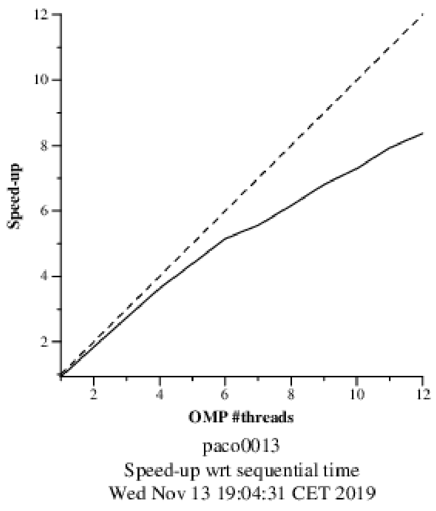
Hemos añadido el nogroup
#pragma omp taskloop nogroup firstprivate(row) num_tasks(8)


Task profile (execution and instantiation) @ mandel-omp-taskloop-v2-8-boada-4.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	6,117	800
THREAD 1.1.2	6,098	-
THREAD 1.1.3	6,557	-
THREAD 1.1.4	6,983	-
THREAD 1.1.5	6,145	-
THREAD 1.1.6	6,539	-
THREAD 1.1.7	6,351	-
THREAD 1.1.8	6,410	-
Total	51,200	800
Average	6,400	800
Maximum	6,983	800
Minimum	6,098	800
StDev	278.99	0
Avg/Max	0.92	1

Figure 6 mandel omp taskloop V2

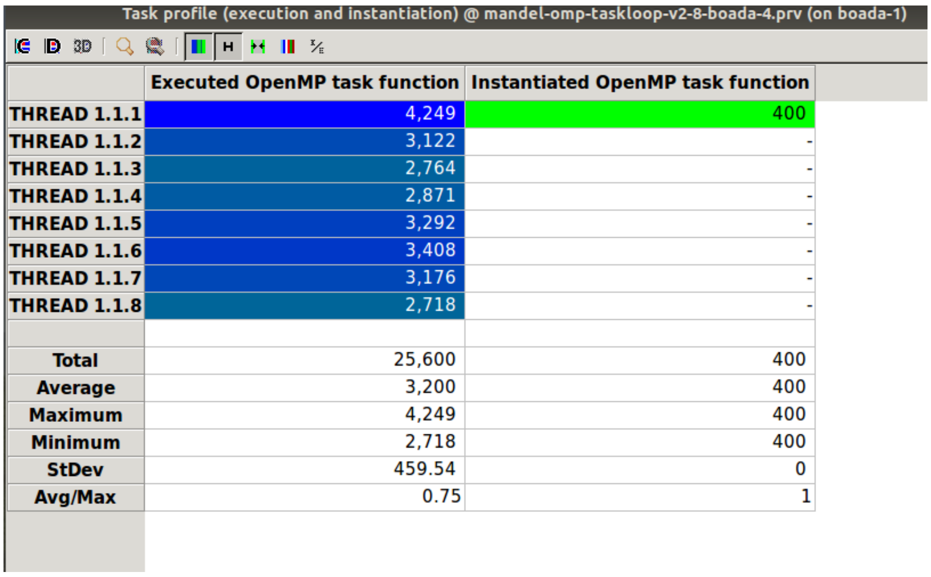




En este aun nos acercamos más a la escalabilidad ideal que el anterior.

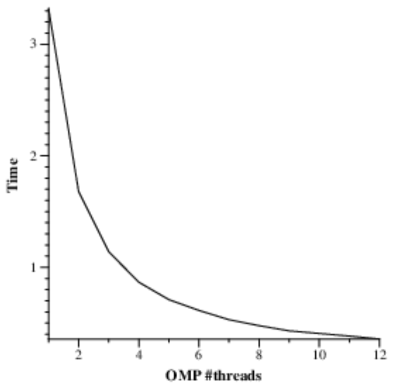
Explore how the new version behaves in terms of performance when using 8 threads and for different task granularities, i.e. setting the number of tasks or iterations per task to 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1, for example. Reason about the results that are obtained.

1. División de la fila en 400 tareas:

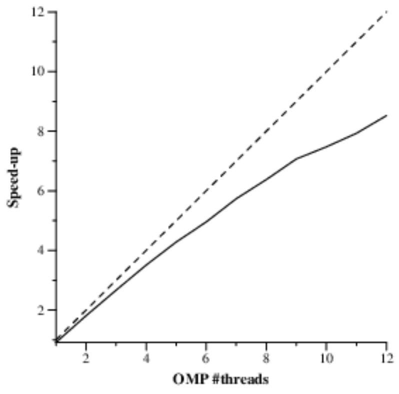


Task profile (execution and instantiation) @ mandel-omp-taskloop-v2-8-boada-4.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	4,249	400
THREAD 1.1.2	3,122	-
THREAD 1.1.3	2,764	-
THREAD 1.1.4	2,871	-
THREAD 1.1.5	3,292	-
THREAD 1.1.6	3,408	-
THREAD 1.1.7	3,176	-
THREAD 1.1.8	2,718	-
Total	25,600	400
Average	3,200	400
Maximum	4,249	400
Minimum	2,718	400
StDev	459.54	0
Avg/Max	0.75	1

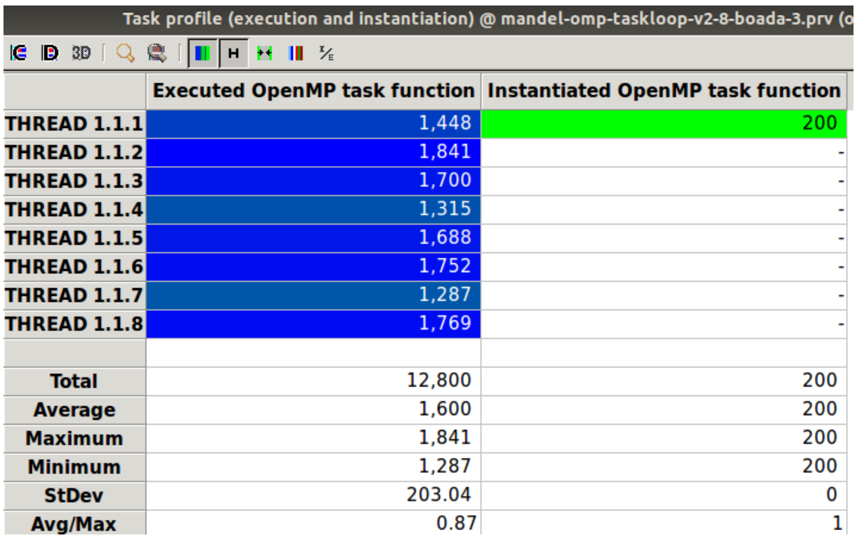


paco0013
Average elapsed execution time
Wed Nov 13 19:36:30 CET 2019



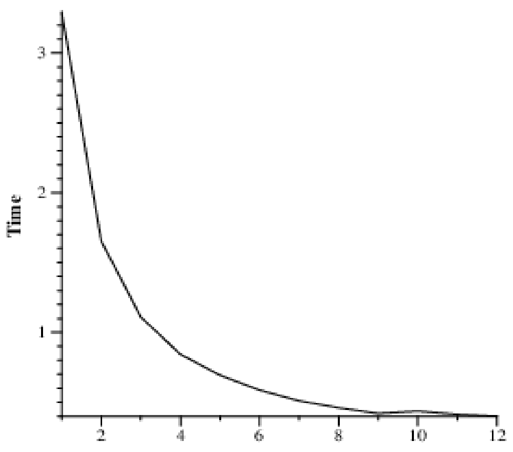
paco0013
Speed-up wrt sequential time
Wed Nov 13 19:36:30 CET 2019

2. División de la fila en 200 tareas:

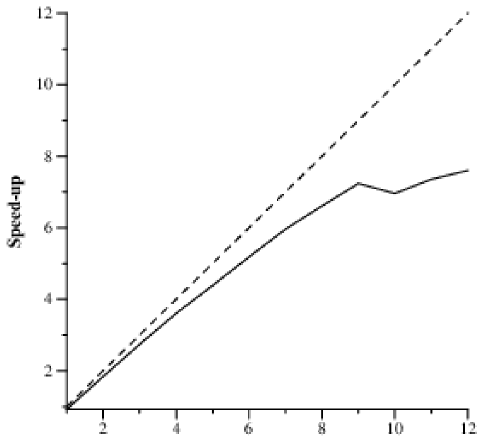


Task profile (execution and instantiation) @ mandel-omp-taskloop-v2-8-boada-3.prv (o

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	1,448	200
THREAD 1.1.2	1,841	-
THREAD 1.1.3	1,700	-
THREAD 1.1.4	1,315	-
THREAD 1.1.5	1,688	-
THREAD 1.1.6	1,752	-
THREAD 1.1.7	1,287	-
THREAD 1.1.8	1,769	-
Total	12,800	200
Average	1,600	200
Maximum	1,841	200
Minimum	1,287	200
StDev	203.04	0
Avg/Max	0.87	1

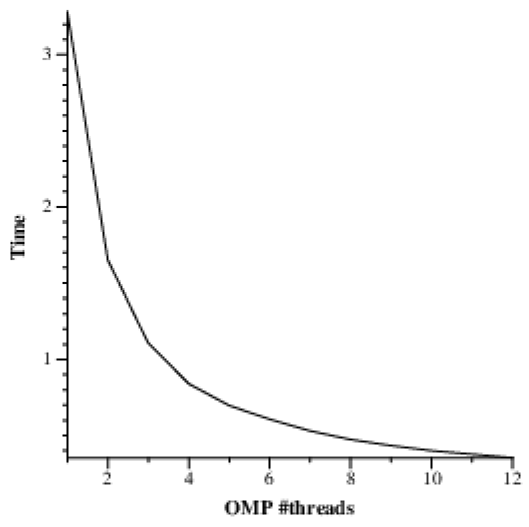


paco0013
Average elapsed execution time
Wed Nov 13 19:32:15 CET 2019

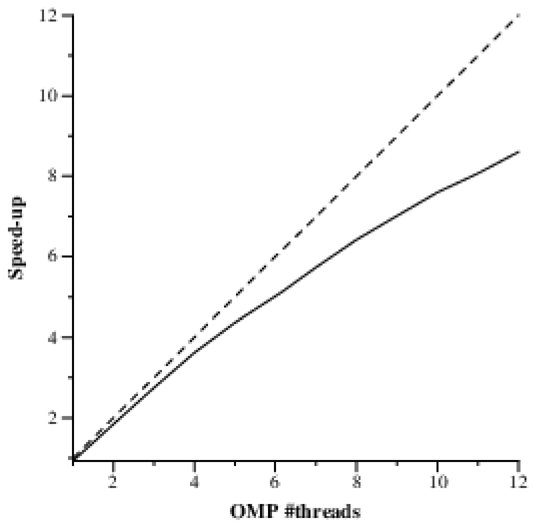


paco0013
Speed-up wrt sequential time
Wed Nov 13 19:32:15 CET 2019

3. División de la fila en 50 tareas:

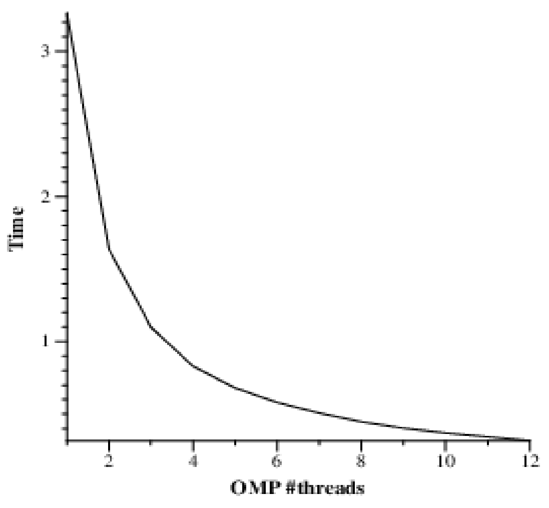


paco0013
Average elapsed execution time
Tue Nov 19 17:14:30 CET 2019

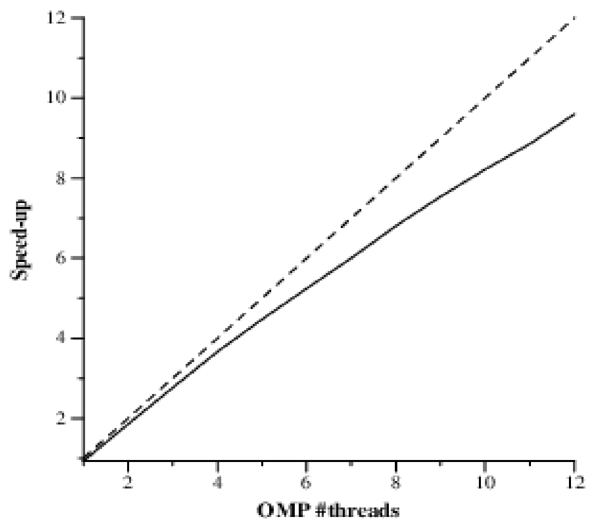


paco0013
Speed-up wrt sequential time
Tue Nov 19 17:14:30 CET 2019

4. División de la fila en 10 tareas:

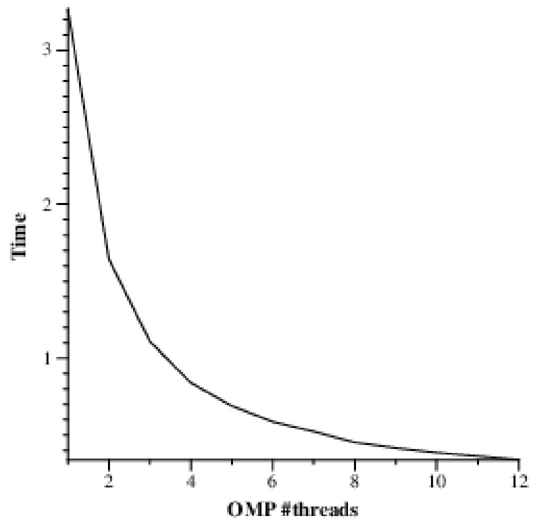


paco0013
Average elapsed execution time
Wed Nov 13 19:44:14 CET 2019

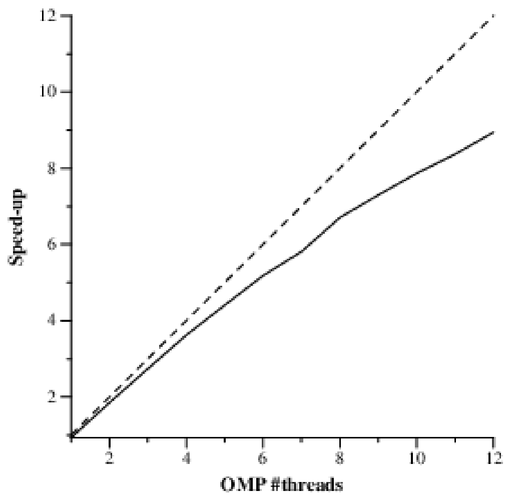


paco0013
Speed-up wrt sequential time
Wed Nov 13 19:44:14 CET 2019

5. División de la fila en 1 sola tarea:

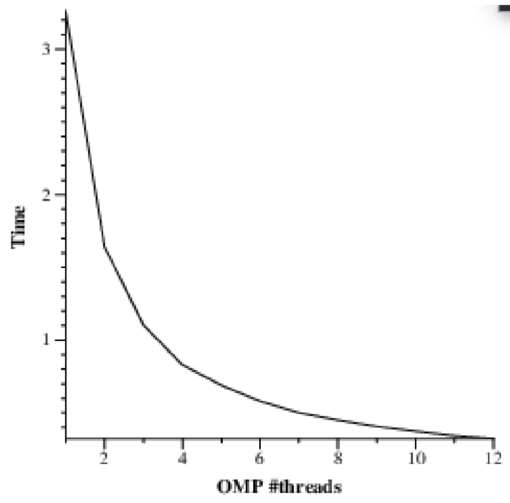


paco0013
Average elapsed execution time
Wed Nov 13 19:47:12 CET 2019

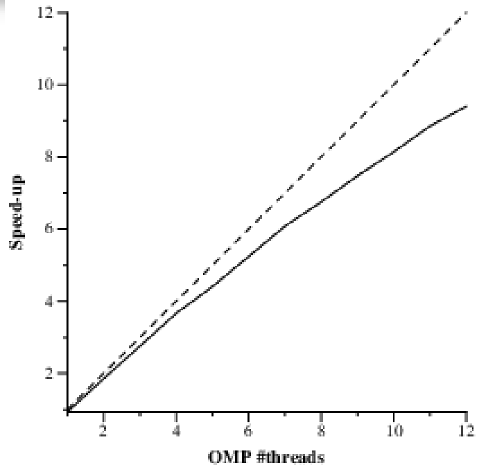


paco0013
Speed-up wrt sequential time
Wed Nov 13 19:47:12 CET 2019

6. División de la fila entre nTreads:



paco0013
Average elapsed execution time
Tue Nov 19 18:16:30 CET 2019



paco0013
Speed-up wrt sequential time
Tue Nov 19 18:16:30 CET 2019

Conclusiones:

Como podemos observar, si dividimos las filas en muchas tareas, si hay muchos threads llega un momento en el que la escalabilidad empieza de reducirse, mientras que si dividimos la fila en menos tareas, es decir, una granularidad más baja, se mantiene más cerca de la escalabilidad ideal al llegar a un nombre mayor threads. Además si nos fijamos en la última gráfica podemos observar que si dividimos la fila entre los threads de cada ejecución el speed up es más estable..

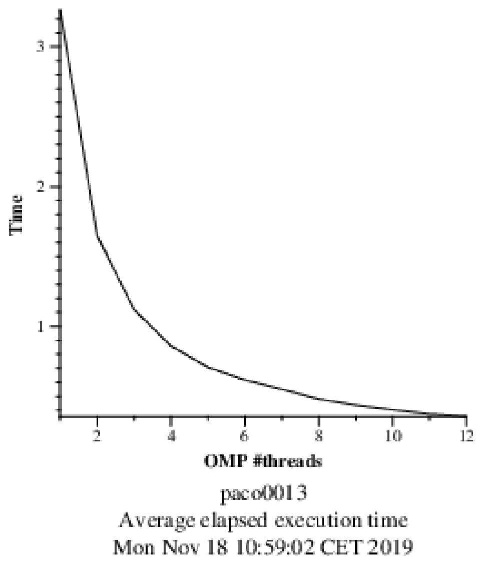
Si observamos las variaciones del nombre de tareas en el que dividimos cada fila podemos llegar a la conclusión de que en quantas menos tareas dividamos cada fila, más estable será la escalabilidad en este caso.

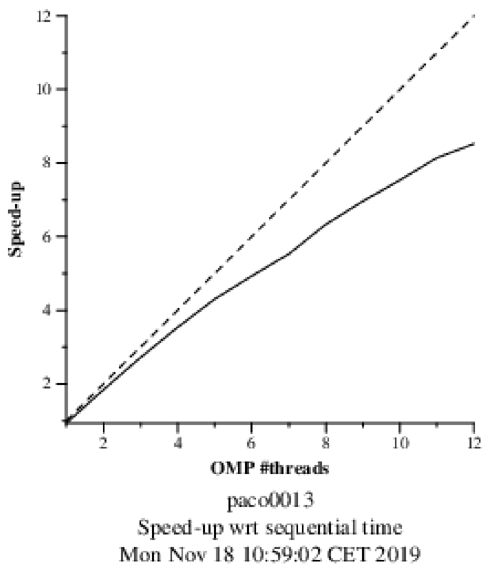
3 Row decomposition in OpenMP

Based on the experience and conclusions you obtained from the previous chapter, in this session you will have to write a new parallel version for mandel-omp.c that obeys to a Row task decomposition. Do the complete analysis of scalability and tracing to conclude about the performance that is obtained with this coarser-grain task decomposition, reasoning about the performance differences that you observe when comparing both strategies (Row and Point).

```
/* Calculate points and save/display */
int num_threads = omp_get_num_threads();
#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(height/num_threads)
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        complex z, c;

        z.real = z.imag = 0;

        /* Scale display coordinates to actual region  */
        c.real = real_min + ((double) col * scale_real);
        c.imag = imag_min + ((double) (height-1-row) * scale_imag);
                                        /* height-1-row so y axis displays
                                         * with larger values at top
                                         */
```

Análisis strong-scalability

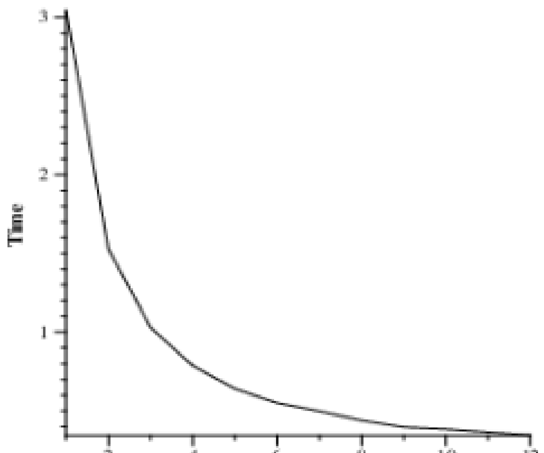




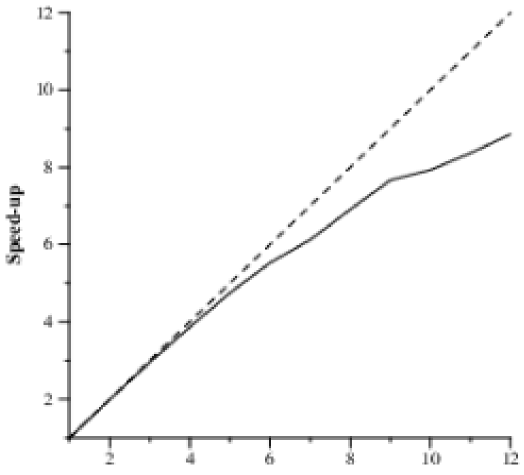
Si comparamos la estrategia de point con row podemos observar en la gráfica de escalabilidad que ambos se acercan bastante al valor ideal, pero si lo miramos con más exactitud podemos ver que en este caso el point con granularidad en taskloop (num_tasks(10)) se aproxima un poco más que el row.

Optional: Write a parallel version for mandel-omp.c making use of the for work–sharing construct instead of tasking to distribute the work among the threads in the team created in the parallel, You should explore the effect of the different schedule kinds in OpenMP for the for work–sharing construct and observe how they appropriate tackle the load balancing problem in the Mandelbrot program.

1. Amb #pragma omp parallel for schedule(dynamic,10)

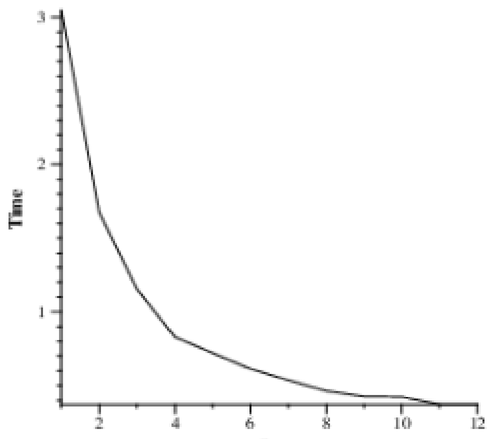


paco0013
Average elapsed execution time
Mon Nov 18 12:03:43 CET 2019

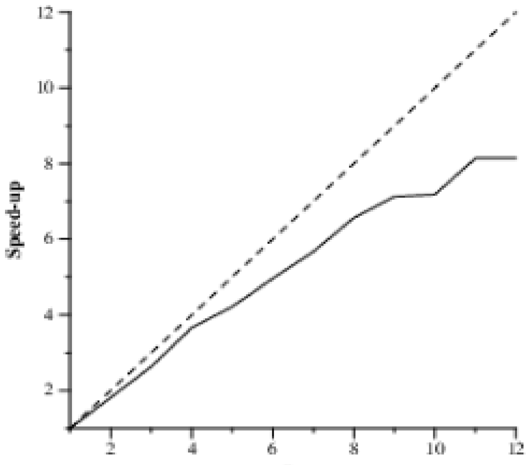


paco0013
Speed-up wrt sequential time
Mon Nov 18 12:03:43 CET 2019

2. Amb #pragma omp parallel for schedule(guided,10)



paco0013
Average elapsed execution time
Mon Nov 18 12:09:23 CET 2019



paco0013
Speed-up wrt sequential time
Mon Nov 18 12:09:23 CET 2019

Si ejecutamos ambos códigos con la versión gráfica podemos observar que ambos son correctos y que se ejecutan a la misma velocidad. Lo que varía de ambos códigos es su análisis de escalabilidad siendo mejor schedule(dynamic,10) por acercarse más al valor ideal, sobretodo en los 5 primeros threads.