Paral·lelisme

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

Grup: par2206
27/03/2020
Curs 2019-2020

Sergi Doce
Ignasi Sant

Índex

Sessió 1

Task decomposition and granularity analysis

Explain the different task decomposition strategies and granularities explored with Tareador for the Mandelbrot code, including in your deliverable the task dependence graphs obtained and clearly indicating the most important characteristics for them (use the small test case -w 8 for this analysis). Explain which section of the code is causing the serialization of all tasks in mandeld-tar and how this section of code should be protected when parallelizing with OpenMP. Reason when each strategy/granularity should be used.

To analyze the potential parallelism we have used two different strategies: the first one creates a task to compute a whole row of the Mandelbrot set, the other one creates a task for each point in the Mandelbrot set. The critical difference we see when we compare both strategies is the workload of each task. However, we have to consider that creating more tasks implies increasing the overhead.
In our examples we have set the number of rows to 8 using the option -w 8.

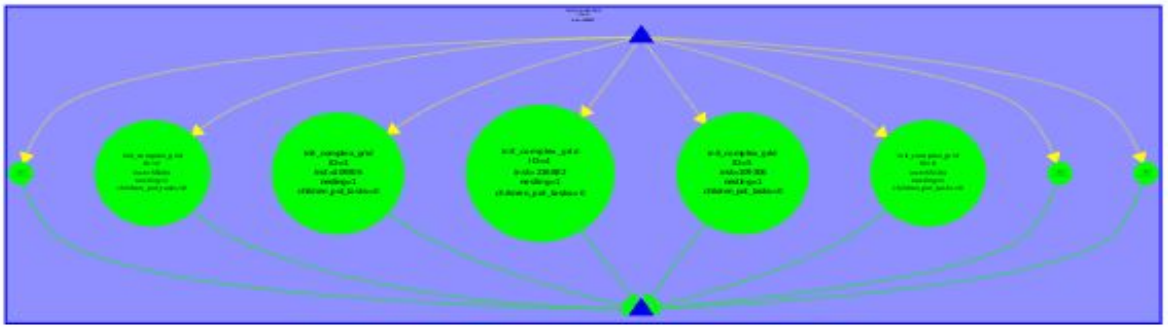
Row granularity strategy

The idea of this strategy is doing a task for each row of the set. Using the option -w 8 , we are setting the number of rows to 8, as there are 8 rows and we are creating a task for row we have 8 tasks. With this strategy we don't too much task to avoid their own overhead.

If we look closely we will see that not all task have the same size due to not all tasks have the same amount of work. All of this is caused by the changing number of iteraions of the loop do-while.

We can see in the images both programs have a similar amount of work to do. The difference resides in the dependencies graphs, Mendel graph is completely parallel while MeIndeld's shows the dependences between tasks that cause sequential execution.

We can see this with Mandel graph because is distributed horizontally, which means that all tasks generated doesn't have any dependence among them. On the other hand Mandeld graph shows us a clear different task decomposition. We observe dependences between nodes.



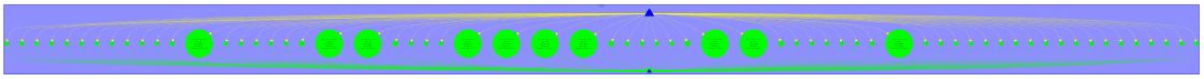
Task decomposition with **Row** Strategy in Mandel -Tar

Task decomposition with **Row** Strategy in Mandeld-Tar

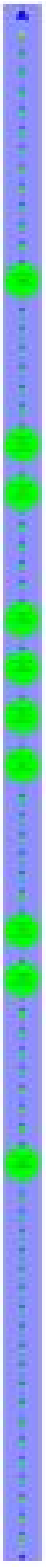
Point granularity strategy

The idea of this strategy is to create a task for each point computed in the set. Again we have used the -w 8 option so we set to 8 the number of rows and colums. The result is that we obtain exacly 8*8=64 tasks, one for each point.
In this case we can also see that not all the tasks of the graphs have the same amount of work. Again this depends on the number of iterations of the loop do-while.
This would be a good parallelism strategy if we have sufficient processors to avoid being penalized with overhead.
The situation is similar as the other strategy

The images below you show the result of Row and Point decomposition on Mandel-tar and Mandeld-tar:



Task decomposition with **Point** Strategy in Mandel -Tar



Task decomposition with **Point** Strategy in Mandeld-Tar

Section of code producing serialization

Mandeld graph shows us a clear different task decomposition. We observe dependences between nodes. The reason of this dependencies occur in the next portion of mandel-tar.c

code. The display variable makes the program execute this piece of code that produces the serialization of all tasks. This happen in both strategies with Mandeld-tar.
The dependences in mandeld that occur in both strategies is caused by a variable named X11_COLOR_fake, and it's used to crate the image generation of the mandled (if_DISPLAY_).

```
#if _DISPLAY_
            /* 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);
            }
#else
            output[row][col]=k;
#endif
```

To protect this region of code in the next sections, we will use the #pragma omp critical directive, with the purpose of defining a region of mutual exclusion where only one thread can be working at the same time.

Conclusion

Once analyzed the difference between both strategies and programes, we end up with the next conclusions: row granularity strategy is a better choice because it don't create so much overhead, on the other hand point granularity could be useful in a system with enough number of processors. In order to say which strategy it's better we have to know the characteristics of the system where the program is executed, including the workload of each task for each version, the task generation overhead and the number of processors of the system.

Sessió 2

Point decomposition in OpenMP

For the Point strategy implemented in OpenMP, describe and reason about how the performance has evolved for the three task versions of the code that you have evaluated, using the speed–up plots obtained and Paraver captures. After that, explain the influence of the granularity control available 9 in the taskloop construct, showing how the execution behaves when setting the number of tasks or iterations per task to 800, 400, 200, 100, 50, 25, 10, 5, 2 and 1, for example. Include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000), again including Paraver captures to help you in the reasoning. Include the

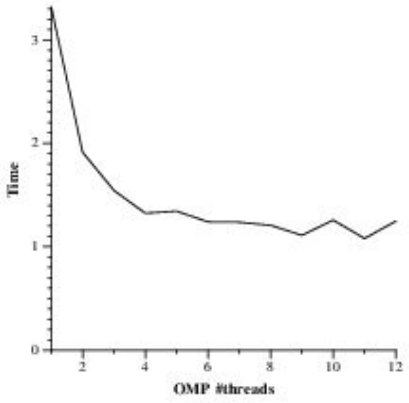
parallel version that makes use of the taskgroup construct. Explain the two fundamental differences with taskwait and if they are relevant in this code. Also include the strong scalability analysis and use tracing to show the differences with taskwait.

In this section we tested three different constructs to parallelize the main program, which are task, task loop and task group.

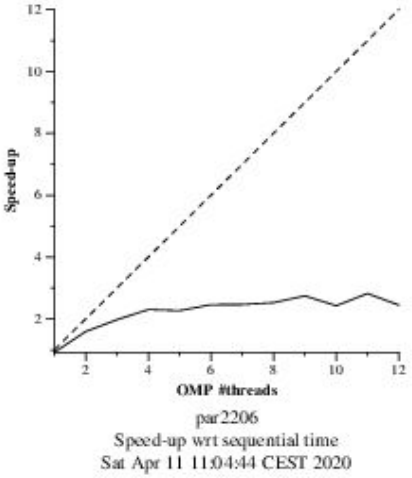
Simplest implementation

At first, we have implemented the point strategy using the **task** clause. We want to do a parallel construct in a row level, the team of threads is created in each iteration of the row loop. To do that we need to use task inside the first for loop, which means that the synchronization after each row is implicit.
Below we have the results of the first version **mandel-omp.c.** We can first notice that the amount of synchronisation overhead is high. However, if we look at the first plot we can see the execution time decreases when we increase the number of threads, furthermore this relation is not linear.
Also the speed-up increases with the amount of threads used but it's far from the ideal. The strong scalability is not performed as expected due to the overhead introduced by the parallel implicit synchronization, which is done after each row.

```
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)
        {
        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
                                 */

        /* Calculate z0, z1, .... until divergence or maximum
erations */
        int k = 0;
        double lengthsq, temp;
        do  {
```



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	84,689	182,400
THREAD 1.1.2	80,258	28,000
THREAD 1.1.3	71,697	56,000
THREAD 1.1.4	76,135	124,000
THREAD 1.1.5	84,160	31,200
THREAD 1.1.6	82,966	3,200
THREAD 1.1.7	76,378	24,000
THREAD 1.1.8	83,717	191,200
Total	640,000	640,000
Average	80,000	80,000
Maximum	84,689	191,200
Minimum	71,697	3,200
StDev	4,457.70	70,252.12
Avg/Max	0.94	0.42



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	41.83 %	-	38.34 %	17.59 %	2.24 %	0.00 %
THREAD 1.1.2	28.73 %	11.16 %	56.30 %	2.95 %	0.86 %	-
THREAD 1.1.3	28.00 %	11.14 %	53.52 %	6.32 %	1.02 %	-
THREAD 1.1.4	26.65 %	11.08 %	46.68 %	14.00 %	1.58 %	-
THREAD 1.1.5	28.65 %	11.16 %	56.29 %	3.02 %	0.88 %	-
THREAD 1.1.6	29.44 %	11.18 %	58.32 %	0.40 %	0.66 %	-
THREAD 1.1.7	28.47 %	11.01 %	55.82 %	2.55 %	2.15 %	-
THREAD 1.1.8	25.67 %	11.23 %	40.97 %	19.18 %	2.94 %	-
Total	237.44 %	77.97 %	406.25 %	66.01 %	12.33 %	0.00 %
Average	29.68 %	11.14 %	50.78 %	8.25 %	1.54 %	0.00 %
Maximum	41.83 %	11.23 %	58.32 %	19.18 %	2.94 %	0.00 %
Minimum	25.67 %	11.01 %	38.34 %	0.40 %	0.66 %	0.00 %
StDev	4.73 %	0.07 %	7.23 %	7.01 %	0.77 %	0 %
Avg/Max	0.71	0.99	0.87	0.43	0.52	1

Results first version of mandel-omp.c

How many times is … now invoked?

Parallel construct -> 800 del primer thread
single worksharing -> 800 per cada thread
taskwait construct -> 0
tasks created -> Imatge “Executed / Instantiated OpenMP task Function”
granularity has changed? No

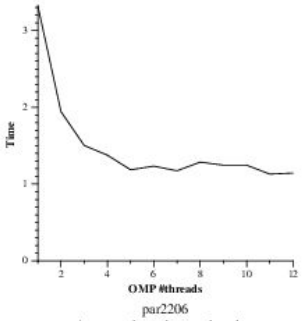
Implementation with taskwait

In this version of the code we will use one parallel region for all the iterations. We won’t have much parallel regions. The thread which creates all the other threads after the creation of each row, will wait them, and then will create them again. If no depend clause is present on the taskwait construct, the current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the taskwait region complete execution.
As we can see the scalability is similar to the one in the first version. Despite we are reducing the overhead we have the problem of waiting the other tasks which is a lose of time. You can notice in paraver that the first thread is doing the creation of the task. The

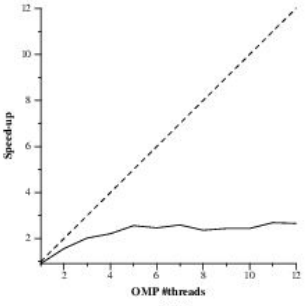
execution time decreases when we increase the number of processadors , but the relation is far from linear.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,636	640,000
THREAD 1.1.2	78,830	-
THREAD 1.1.3	75,860	-
THREAD 1.1.4	75,703	-
THREAD 1.1.5	81,655	-
THREAD 1.1.6	81,227	-
THREAD 1.1.7	75,789	-
THREAD 1.1.8	82,300	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,636	640,000
Minimum	75,703	640,000
StDev	4,165.64	0
Avg/Max	0.90	1

```
    #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)
            {
            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
                                     */

            /* Calculate z0, z1, .... until divergence or maximum iterations */
            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_
            /* Scale color and display point  */
            {
            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
            output[row][col]=k;
#endif
            }
        }
        #pragma omp taskwait
    }
}
```

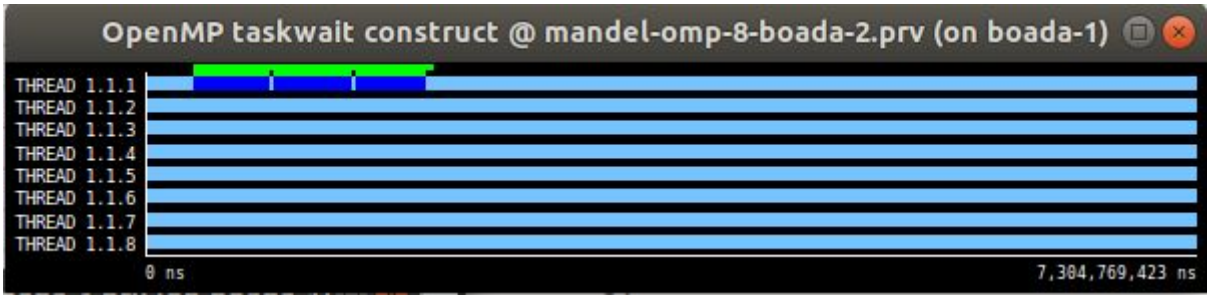


par2206
Average elapsed execution time
Sun Apr 12 09:56:06 CEST 2020



par2206
Speed-up wrt sequential time
Sun Apr 12 09:56:06 CEST 2020

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	81.51 %	-	0.71 %	16.32 %	1.46 %	0.00 %
THREAD 1.1.2	26.43 %	16.87 %	56.12 %	0.00 %	0.59 %	-
THREAD 1.1.3	26.10 %	16.88 %	56.51 %	0.00 %	0.51 %	-
THREAD 1.1.4	26.08 %	16.88 %	56.54 %	0.00 %	0.49 %	-
THREAD 1.1.5	26.39 %	16.87 %	56.18 %	0.00 %	0.56 %	-
THREAD 1.1.6	26.37 %	16.88 %	56.19 %	0.00 %	0.56 %	-
THREAD 1.1.7	26.12 %	16.87 %	56.44 %	0.00 %	0.57 %	-
THREAD 1.1.8	26.31 %	16.79 %	55.87 %	0.00 %	1.03 %	-
Total	265.31 %	118.05 %	394.55 %	16.32 %	5.77 %	0.00 %
Average	33.16 %	16.86 %	49.32 %	2.04 %	0.72 %	0.00 %
Maximum	81.51 %	16.88 %	56.54 %	16.32 %	1.46 %	0.00 %
Minimum	26.08 %	16.79 %	0.71 %	0.00 %	0.49 %	0.00 %
StDev	18.27 %	0.03 %	18.37 %	5.40 %	0.32 %	0 %
Avg/Max	0.41	1.00	0.87	0.13	0.49	1



Results second version of mandel-omp.c

How many times is … now invoked?

Parallel construct -> 1 del primer thread
single worksharing -> 8, un per thread
taskwait contruct -> 800
tasks created -> Imatge “Executed / Instantiated OpenMP task Function”
granularity has changed? No

Implementation with taskgroup

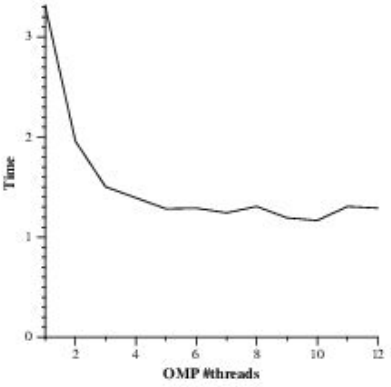
Alternatively to the use of taskwait one can use the #pragma omp taskgroup construct, which defines a region in the program, at the end of which the thread will wait for the termination of all descendant (not only child) tasks. In this version we will use it.

If we take a look on the parver information and the plots, we can see that the situation is similar to the first version. Despite, we can notice an increase of the synchronization time and decrease on the scheduling time, at the end this this version and taskwait version are so similar.

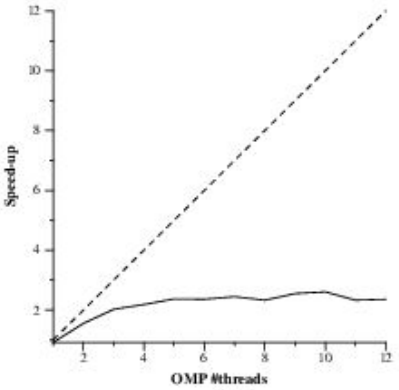
If we compare the taskwait version with this one we get the following conclusion: there’s no need to wait for the termination of all tasks in a row before generating the next row tasks, and with taskwait we are wating some time.

In this versions the threads generating tasks stops task generation, then executes some tasks , and then proceeds generating new tasks. When a thread encounters a taskgroup construct, it starts executing the region. All child tasks generated in the taskgroup region and all of their descendants that bind to the same parallel region as the taskgroup region are part of the taskgroup set associated with the taskgroup region. There is an implicit task scheduling point at the end of the taskgroup region. The current task is suspended at the task scheduling point until all tasks in the taskgroup set complete execution.

```
    #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)
            {
            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
                                     */

            /* Calculate z0, z1, .... until divergence or maximum
iterations */
            int k = 0;
            double lengthsq, temp;
            do  {
```



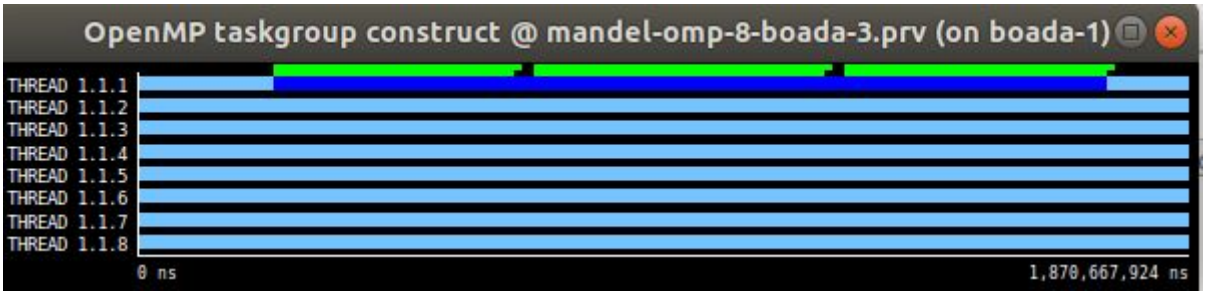
par2206
Average elapsed execution time
Sun Apr 12 10:50:24 CEST 2020



par2206
Speed-up wrt sequential time
Sun Apr 12 10:50:24 CEST 2020

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	34.09 %	-	2.68 %	57.56 %	5.66 %	0.00 %
THREAD 1.1.2	28.54 %	13.92 %	56.93 %	0.00 %	0.61 %	-
THREAD 1.1.3	28.74 %	13.92 %	56.71 %	0.00 %	0.62 %	-
THREAD 1.1.4	28.89 %	13.93 %	56.62 %	0.00 %	0.56 %	-
THREAD 1.1.5	28.70 %	13.92 %	56.75 %	0.00 %	0.63 %	-
THREAD 1.1.6	28.64 %	13.92 %	56.83 %	0.00 %	0.61 %	-
THREAD 1.1.7	28.60 %	13.81 %	56.12 %	0.00 %	1.47 %	-
THREAD 1.1.8	28.62 %	13.83 %	56.22 %	0.00 %	1.33 %	-
Total	234.81 %	97.26 %	398.87 %	57.56 %	11.50 %	0.00 %
Average	29.35 %	13.89 %	49.86 %	7.20 %	1.44 %	0.00 %
Maximum	34.09 %	13.93 %	56.93 %	57.56 %	5.66 %	0.00 %
Minimum	28.54 %	13.81 %	2.68 %	0.00 %	0.56 %	0.00 %
StDev	1.79 %	0.04 %	17.83 %	19.04 %	1.63 %	0 %
Avg/Max	0.86	1.00	0.88	0.13	0.25	1

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,623	640,000
THREAD 1.1.2	81,847	-
THREAD 1.1.3	80,483	-
THREAD 1.1.4	70,885	-
THREAD 1.1.5	84,407	-
THREAD 1.1.6	82,051	-
THREAD 1.1.7	74,300	-
THREAD 1.1.8	76,404	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	89,623	640,000
Minimum	70,885	640,000
StDev	5,572.02	0
Avg/Max	0.89	1



Results third version of mandel-omp.c

How many times is … now invoked?

Parallel construct -> 1 per al primer thread

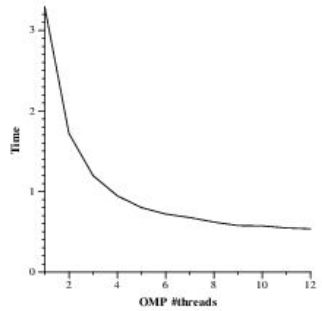
single worksharing -> 8, un per cada thread
taskwait construct -> 0
tasks created -> Imatge “Executed / Instantiated OpenMP task Function”
granularity has changed? NO

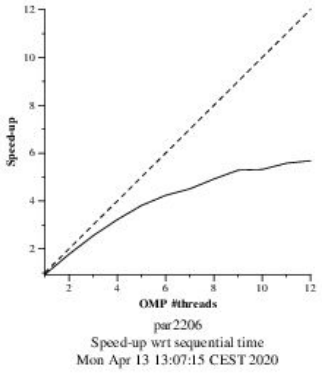
Granularity control with taskloop

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.
The **taskloop** construct is a task generating construct. When a thread encounters a taskloop construct, the construct partitions the iterations of the associated loops into explicit tasks for parallel execution. The data environment of each generated task is created according to the data-sharing attribute clauses on the taskloop construct, per-data environment ICVs, and any defaults that apply. The order of the creation of the loop tasks is unspecified. Programs that rely on any execution order of the logical loop iterations are non-conforming. By default, the taskloop construct executes as if it was enclosed in a taskgroup construct with no statements or directives outside of the taskloop construct. Thus, the taskloop construct creates an implicit taskgroup region.
If the **nogroup** clause is present, no implicit taskgroup region is created.
If a **grainsize** clause is present on the taskloop construct, the number of logical loop iterations assigned to each generated task is greater than or equal to the minimum of the value of the grain-size expression and the number of logical loop iterations, but less than two times the value of the grain-size expression. The parameter of the grainsize clause must be a positive integer expression.
If **num_tasks** is specified, the taskloop construct creates as many tasks as the minimum of the num-tasks expression and the number of logical loop iterations. Each task must have at least one logical loop iteration. The parameter of the num_tasks clause must be a positive integer expression. If neither a grainsize nor num_tasks clause is present, the number of loop tasks generated and the number of logical loop iterations assigned to these tasks is implementation defined.

As we see in the plot

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(64)
    for (int col = 0; col < width; ++col) {
```





	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	72.69 %	-	17.91 %	9.24 %	0.16 %	0.00 %
THREAD 1.1.2	52.68 %	25.92 %	21.30 %	0.00 %	0.10 %	-
THREAD 1.1.3	52.84 %	25.92 %	21.15 %	0.00 %	0.09 %	-
THREAD 1.1.4	53.59 %	25.91 %	20.40 %	0.00 %	0.10 %	-
THREAD 1.1.5	53.76 %	25.92 %	20.22 %	0.00 %	0.10 %	-
THREAD 1.1.6	53.17 %	25.93 %	20.80 %	0.00 %	0.10 %	-
THREAD 1.1.7	52.16 %	25.93 %	21.80 %	0.00 %	0.10 %	-
THREAD 1.1.8	53.59 %	25.93 %	20.38 %	0.00 %	0.10 %	-
Total	444.47 %	181.46 %	163.97 %	9.24 %	0.86 %	0.00 %
Average	55.56 %	25.92 %	20.50 %	1.15 %	0.11 %	0.00 %
Maximum	72.69 %	25.93 %	21.80 %	9.24 %	0.16 %	0.00 %
Minimum	52.16 %	25.91 %	17.91 %	0.00 %	0.09 %	0.00 %
StDev	6.49 %	0.01 %	1.10 %	3.05 %	0.02 %	0 %
Avg/Max	0.76	1.00	0.94	0.13	0.65	1

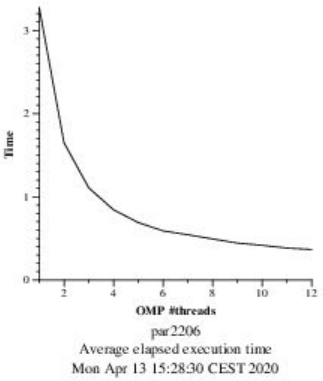
Results fourth version of mandel-omp.c

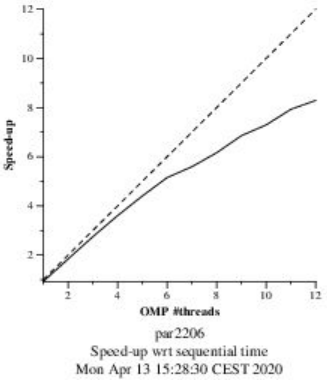
Same taskloop version + nogrup

In this version we just add the nogroup clause. As we can see in the plots we get better results. Using nogrup clause we get a speed-up closer to the ideal. Also the relation between execution elapsed time and cores looks more parallelizable.
All of this happens because we have **no dependencies** between tasks. So waiting until all the group has finished is a waste of time. You can see the Synchronization has decreased a lot, you can also notice that looking at the execution time.
As we just get better results we will be using the nogrup clause in the following versions.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(64) nogroup
    for (int col = 0; col < width; ++col) {
```

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	91.43 %	-	0.00 %	8.39 %	0.18 %	0.00 %
THREAD 1.1.2	62.51 %	29.90 %	7.44 %	0.00 %	0.14 %	-
THREAD 1.1.3	62.56 %	29.92 %	7.42 %	0.00 %	0.10 %	-
THREAD 1.1.4	62.60 %	29.92 %	7.38 %	0.00 %	0.10 %	-
THREAD 1.1.5	62.61 %	29.90 %	7.36 %	0.00 %	0.13 %	-
THREAD 1.1.6	62.68 %	29.98 %	7.22 %	0.00 %	0.12 %	-
THREAD 1.1.7	62.60 %	29.93 %	7.36 %	0.00 %	0.11 %	-
THREAD 1.1.8	62.56 %	29.96 %	7.35 %	0.00 %	0.13 %	-
Total	529.54 %	209.52 %	51.53 %	8.40 %	1.01 %	0.00 %
Average	66.19 %	29.93 %	6.44 %	1.05 %	0.13 %	0.00 %
Maximum	91.43 %	29.98 %	7.44 %	8.39 %	0.18 %	0.00 %
Minimum	62.51 %	29.90 %	0.00 %	0.00 %	0.10 %	0.00 %
StDev	9.54 %	0.03 %	2.43 %	2.78 %	0.02 %	0 %
Avg/Max	0.72	1.00	0.87	0.13	0.72	1



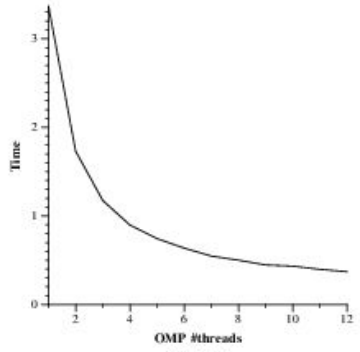


Playing with grainsize

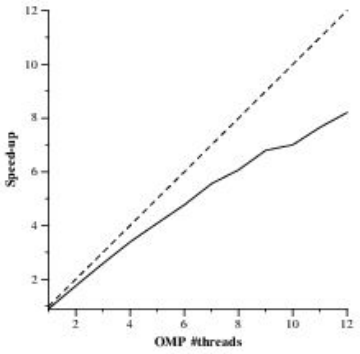
In this versions of mandel-omp.c we will be changeing the grainsize value using 1, 2 , 50, 400, 800.
If a grainsize clause is present on the taskloop construct, the number of logical loop iterations assigned to each generated task is greater than or equal to the minimum of the value of the grain-size expression and the number of logical loop iterations, but less than two times the value of the grain-size expression.

Grainsize 1

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) grainsize(1) nogroup
    for (int col = 0; col < width; ++col) {
```



par2206
Average elapsed execution time
Mon Apr 13 17:44:01 CEST 2020

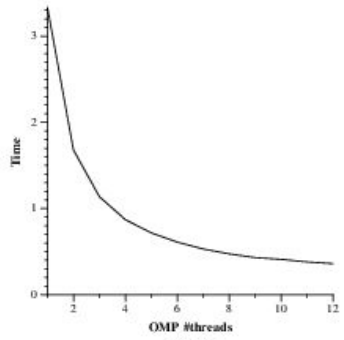


par2206
Speed-up wrt sequential time
Mon Apr 13 17:44:01 CEST 2020

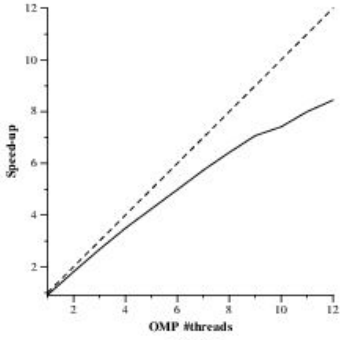
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	93.23 %	-	0.03 %	4.02 %	2.72 %	0.00 %
THREAD 1.1.2	59.50 %	25.81 %	13.18 %	0.10 %	1.40 %	-
THREAD 1.1.3	59.75 %	25.85 %	13.06 %	0.10 %	1.24 %	-
THREAD 1.1.4	60.47 %	26.06 %	12.19 %	0.10 %	1.18 %	-
THREAD 1.1.5	59.65 %	25.80 %	13.00 %	0.11 %	1.44 %	-
THREAD 1.1.6	60.36 %	25.85 %	12.41 %	0.10 %	1.29 %	-
THREAD 1.1.7	60.28 %	25.95 %	12.46 %	0.10 %	1.22 %	-
THREAD 1.1.8	60.14 %	25.88 %	12.48 %	0.11 %	1.40 %	-
Total	513.37 %	181.20 %	88.81 %	4.73 %	11.89 %	0.00 %
Average	64.17 %	25.89 %	11.10 %	0.59 %	1.49 %	0.00 %
Maximum	93.23 %	26.06 %	13.18 %	4.02 %	2.72 %	0.00 %
Minimum	59.50 %	25.80 %	0.03 %	0.10 %	1.18 %	0.00 %
StDev	10.99 %	0.08 %	4.20 %	1.30 %	0.47 %	0 %
Avg/Max	0.69	0.99	0.84	0.15	0.55	1

Grainsize 2

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) grainsize(2) nogroup
    for (int col = 0; col < width; ++col) {
```



par2206
Average elapsed execution time
Mon Apr 13 18:01:10 CEST 2020

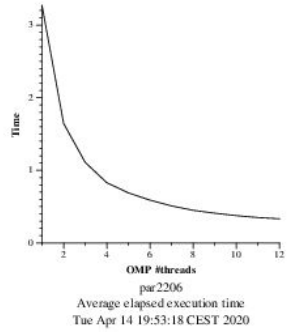


par2206
Speed-up wrt sequential time
Mon Apr 13 18:01:10 CEST 2020

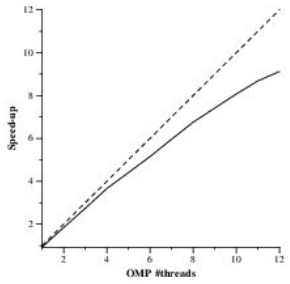
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	93.25 %	-	5.99 %	0.23 %	0.53 %	0.00 %
THREAD 1.1.2	64.27 %	28.96 %	6.14 %	0.03 %	0.60 %	-
THREAD 1.1.3	64.34 %	28.97 %	6.10 %	0.03 %	0.56 %	-
THREAD 1.1.4	66.01 %	28.59 %	0.05 %	3.53 %	1.82 %	-
THREAD 1.1.5	63.42 %	28.95 %	6.95 %	0.04 %	0.65 %	-
THREAD 1.1.6	63.87 %	28.96 %	6.52 %	0.04 %	0.62 %	-
THREAD 1.1.7	64.42 %	28.96 %	5.00 %	0.03 %	0.58 %	-
THREAD 1.1.8	63.97 %	28.97 %	6.45 %	0.03 %	0.57 %	-
Total	543.56 %	202.36 %	44.19 %	3.96 %	5.93 %	0.00 %
Average	67.94 %	28.91 %	5.52 %	0.49 %	0.74 %	0.00 %
Maximum	93.25 %	28.97 %	6.95 %	3.53 %	1.82 %	0.00 %
Minimum	63.42 %	28.59 %	0.05 %	0.03 %	0.53 %	0.00 %
StDev	9.59 %	0.13 %	2.09 %	1.15 %	0.41 %	0 %
Avg/Max	0.73	1.00	0.79	0.14	0.41	1

Grainsize 50

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) grainsize(50) nogroup
    for (int col = 0; col < width; ++col) {
```



par2206
Average elapsed execution time
Tue Apr 14 19:53:18 CEST 2020

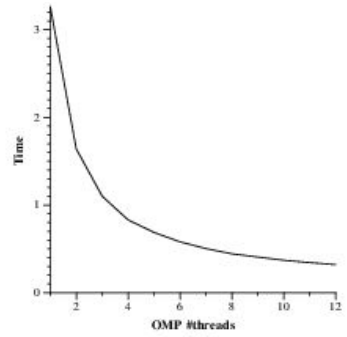


par2206
Speed-up wrt sequential time
Tue Apr 14 19:53:18 CEST 2020

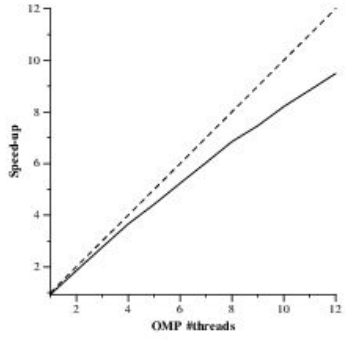
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	98.33 %	-	0.00 %	1.58 %	0.09 %	0.00 %
THREAD 1.1.2	68.44 %	30.61 %	0.93 %	0.00 %	0.03 %	-
THREAD 1.1.3	68.43 %	30.61 %	0.92 %	0.00 %	0.03 %	-
THREAD 1.1.4	68.43 %	30.61 %	0.93 %	0.00 %	0.03 %	-
THREAD 1.1.5	68.45 %	30.61 %	0.91 %	0.00 %	0.03 %	-
THREAD 1.1.6	68.44 %	30.62 %	0.91 %	0.00 %	0.03 %	-
THREAD 1.1.7	68.45 %	30.61 %	0.91 %	0.00 %	0.03 %	-
THREAD 1.1.8	68.46 %	30.64 %	0.87 %	0.00 %	0.03 %	-
Total	577.44 %	214.32 %	6.38 %	1.58 %	0.28 %	0.00 %
Average	72.18 %	30.62 %	0.80 %	0.20 %	0.03 %	0.00 %
Maximum	98.33 %	30.64 %	0.93 %	1.58 %	0.09 %	0.00 %
Minimum	68.43 %	30.61 %	0.00 %	0.00 %	0.03 %	0.00 %
StDev	9.89 %	0.01 %	0.30 %	0.52 %	0.02 %	0 %
Avg/Max	0.73	1.00	0.86	0.13	0.41	1

Grainsize 200

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) grainsize(200) nogroup
    for (int col = 0; col < width; ++col) {
```



par2206
Average elapsed execution time
Tue Apr 14 20:12:55 CEST 2020

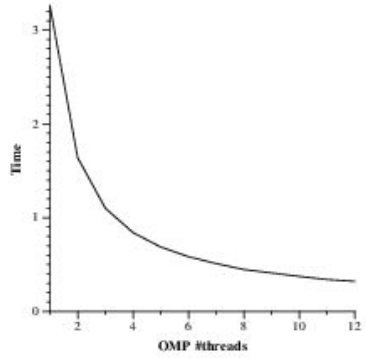


par2206
Speed-up wrt sequential time
Tue Apr 14 20:12:55 CEST 2020

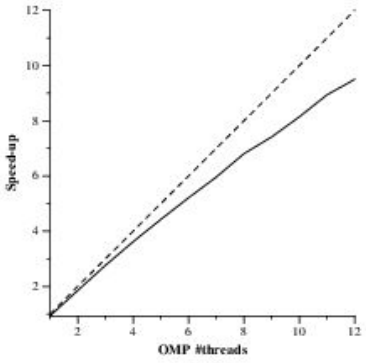
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	98.65 %	-	0.56 %	0.74 %	0.05 %	0.00 %
THREAD 1.1.2	69.70 %	29.98 %	0.31 %	0.00 %	0.01 %	-
THREAD 1.1.3	69.77 %	29.98 %	0.24 %	0.00 %	0.01 %	-
THREAD 1.1.4	69.38 %	29.98 %	0.63 %	0.00 %	0.01 %	-
THREAD 1.1.5	69.45 %	30.03 %	0.51 %	0.00 %	0.01 %	-
THREAD 1.1.6	69.55 %	30.06 %	0.38 %	0.00 %	0.01 %	-
THREAD 1.1.7	69.34 %	30.04 %	0.62 %	0.00 %	0.01 %	-
THREAD 1.1.8	69.52 %	30.05 %	0.43 %	0.00 %	0.01 %	-
Total	585.37 %	210.11 %	3.68 %	0.74 %	0.10 %	0.00 %
Average	73.17 %	30.02 %	0.46 %	0.09 %	0.01 %	0.00 %
Maximum	98.65 %	30.06 %	0.63 %	0.74 %	0.05 %	0.00 %
Minimum	69.34 %	29.98 %	0.24 %	0.00 %	0.01 %	0.00 %
StDev	9.63 %	0.03 %	0.13 %	0.24 %	0.01 %	0 %
Avg/Max	0.74	1.00	0.73	0.13	0.25	1

Grainsize 400

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) grainsize(400) nogroup
    for (int col = 0; col < width; ++col) {
```



par2206
Average elapsed execution time
Tue Apr 14 20:08:43 CEST 2020

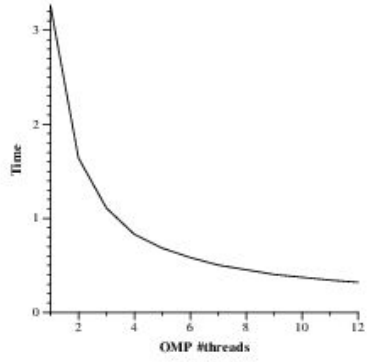


par2206
Speed-up wrt sequential time
Tue Apr 14 20:08:43 CEST 2020

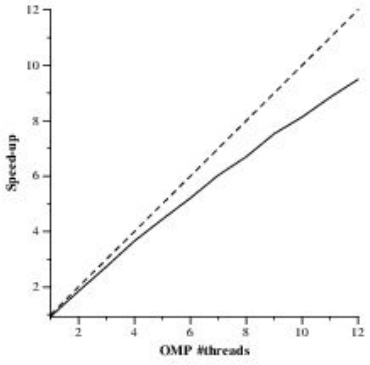
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.13 %	-	0.22 %	0.61 %	0.04 %	0.00 %
THREAD 1.1.2	65.77 %	33.32 %	0.91 %	0.00 %	0.00 %	-
THREAD 1.1.3	66.51 %	33.32 %	0.17 %	0.00 %	0.00 %	-
THREAD 1.1.4	66.04 %	33.37 %	0.59 %	0.00 %	0.00 %	-
THREAD 1.1.5	66.27 %	33.32 %	0.40 %	0.00 %	0.00 %	-
THREAD 1.1.6	65.99 %	33.37 %	0.64 %	0.00 %	0.00 %	-
THREAD 1.1.7	66.16 %	33.38 %	0.46 %	0.00 %	0.00 %	-
THREAD 1.1.8	66.39 %	33.38 %	0.22 %	0.00 %	0.00 %	-
Total	562.27 %	233.46 %	3.60 %	0.61 %	0.06 %	0.00 %
Average	70.28 %	33.35 %	0.45 %	0.08 %	0.01 %	0.00 %
Maximum	99.13 %	33.38 %	0.91 %	0.61 %	0.04 %	0.00 %
Minimum	65.77 %	33.32 %	0.17 %	0.00 %	0.00 %	0.00 %
StDev	10.91 %	0.03 %	0.24 %	0.20 %	0.01 %	0 %
Avg/Max	0.71	1.00	0.50	0.13	0.20	1

Grainsize 800

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) grainsize(800) nogroup
    for (int col = 0; col < width; ++col) {
```



par2206
Average elapsed execution time
Tue Apr 14 20:03:51 CEST 2020



par2206
Speed-up wrt sequential time
Tue Apr 14 20:03:51 CEST 2020

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	98.80 %	-	0.75 %	0.42 %	0.04 %	0.00 %
THREAD 1.1.2	68.71 %	31.15 %	0.14 %	0.00 %	0.00 %	-
THREAD 1.1.3	67.91 %	31.15 %	0.94 %	0.00 %	0.00 %	-
THREAD 1.1.4	68.35 %	31.14 %	0.50 %	0.00 %	0.00 %	-
THREAD 1.1.5	68.78 %	31.19 %	0.03 %	0.00 %	0.00 %	-
THREAD 1.1.6	68.24 %	31.21 %	0.55 %	0.00 %	0.00 %	-
THREAD 1.1.7	68.76 %	31.20 %	0.04 %	0.00 %	0.00 %	-
THREAD 1.1.8	68.05 %	31.21 %	0.74 %	0.00 %	0.00 %	-
Total	577.60 %	218.24 %	3.68 %	0.42 %	0.05 %	0.00 %
Average	72.20 %	31.18 %	0.46 %	0.05 %	0.01 %	0.00 %
Maximum	98.80 %	31.21 %	0.94 %	0.42 %	0.04 %	0.00 %
Minimum	67.91 %	31.14 %	0.03 %	0.00 %	0.00 %	0.00 %
StDev	10.06 %	0.03 %	0.33 %	0.14 %	0.01 %	0 %
Avg/Max	0.73	1.00	0.49	0.13	0.18	1

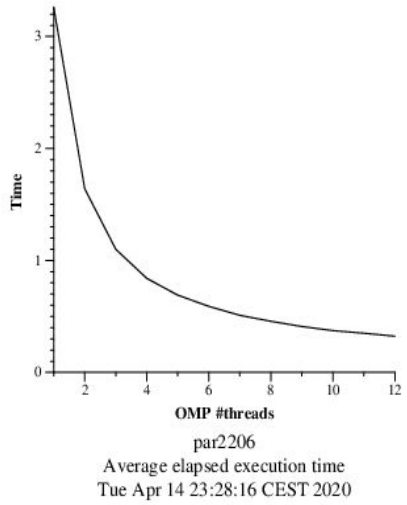
As we can see we get the best results when we use taskloop + nogroup + grainsize(400). The conclusion is that is we use a lower grain size, will decease the performance and the execution time will increase. The same happen if we use a high grainsize value like 800. The best option is an intermediate option like 400 or 200 which give us a very good performance. The plots are so similar but grainsize 400 has given a little better result.

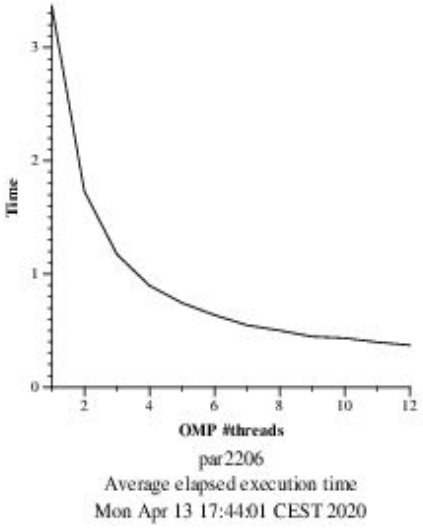
Sessió 3

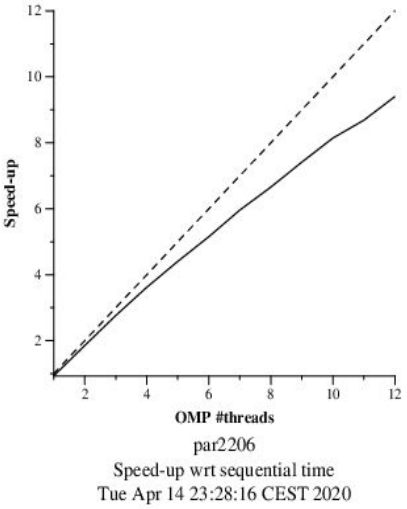
Row decomposition in OpenMP

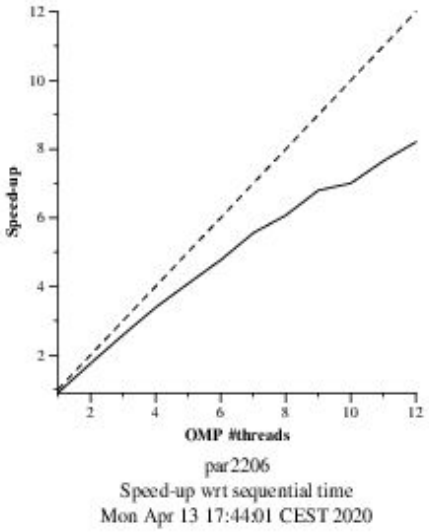
For the Row strategy implemented in OpenMP, describe the parallelization strategy that you have decided to implement. Reason about the performance that is obtained comparing with the results obtained for the best Point implementation, including the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000) and Paraver captures that help you to better explain the results that have been obtained.

Having finished the previous section and having tested all strategies, we can conclude that the best is the taskloop strategy, so that is the strategy that we are going to implement for the row decomposition. In order to create only one task per row, we have to set the grainsize of the taskloop to 1. Having changed and executed the code, what we obtain are this results:









The left plot corresponds to the row decomposition version applying taskloop with a grainsize of 1. The right plots correspond to the same strategy but with point decomposition. As we can see, the execution time plots are almost the same. But we can't say the same thing about the speedup plots, as we can see that the row decomposition plot is slightly better. This may be caused due to overhead, because in the row decomposition version we can see that there is less synchronisation overhead.

So, in conclusion, altough both strategies perform weel when talking about execution time, if we had to choose between one or another, we would choose Row decomposition because as we have seen in this chapter, it offers a better scalability as it requires less synchronisation due to less task creation.