Parallelism

Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Jia Long Ji Qiu: par2212
Jiabo Wang: par2220
Primavera 2021-22
21 de abril de 2022

Index

Introduction

In this laboratory assignment, we are going to study the different task decomposition strategies in OpenMP, which will be implemented to calculate the Mandelbrot set so we can analyze the results obtained and report the conclusions.

Task decomposition and granularity analysis for the Mandelbrot set computation

Task decomposition analysis with Tareador

Analysis of the potential parallelism for the Row strategy of Mandel-tar.c

with no additional options

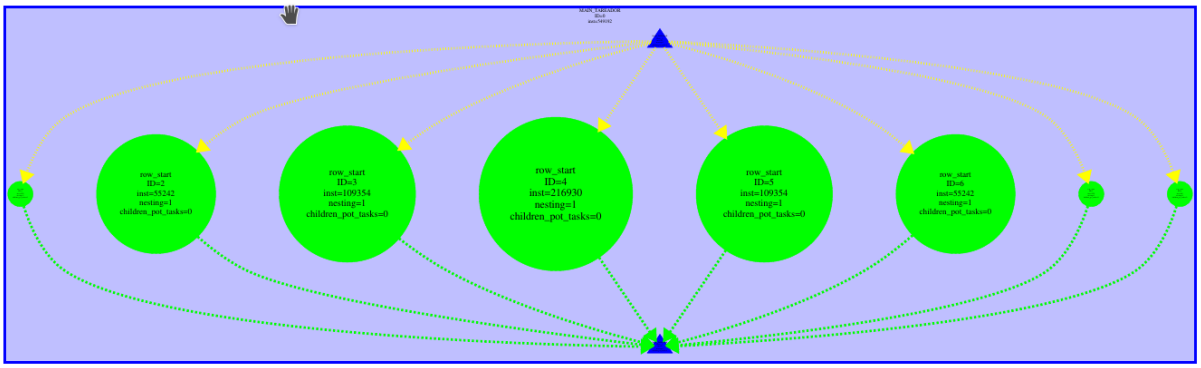


Figure 1: TDG of the program with no additional options

Which are the two most important characteristics for the task graph that is generated?

1. There are no dependencies between the tasks generated, so they are independent. Also we can prove the independence between the tasks by looking at the figure before (Figure 1), as we can see that the tasks are arranged horizontally so there are no dependencies.
2. The sizes of the tasks are different, some require more time than others, this is because some of them execute more iterations than the others (the middle ones execute more iterations than the extreme ones, since they diverge faster).

-d option

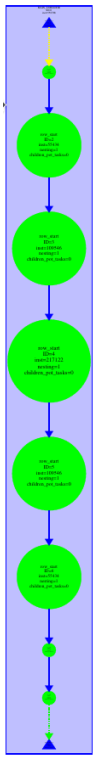


Figure 2: TDG of the program with -d option

Which are the two most important characteristics for the task graph that is generated?

1. As we can see in the dependency graph above (Figure 2), the tasks are arranged vertically, so we can say that the tasks are dependent. This is because the piece of code being added is executed dependently, XSetForeground (sets the color) and XDrawPoint (uses the color that has been set by XSetForeground). So a task starts when the previous one finishes.
2. The sizes of the tasks are different, some require more time than others, this is because some of them execute more iterations than the others (the middle ones execute more iterations than the extreme ones, since they diverge faster).

Which part of the code is making the big difference with the previous case?

```
if (output2display) {
    /* 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);
    }
}
```

Figure 3: code snippet of the program

The instructions XSetForeground and XDrawPoint are the ones that create the dependencies between the tasks.

How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions?

There are different ways to fix this problem. The first option to work around this dependency, i.e. to try to prevent datarace from occurring, is to use the #pragma omp critical constructor. And the other way to solve this problem (more efficient, but more complicated), is to do the calculation first and paint the pixels last.

-h option

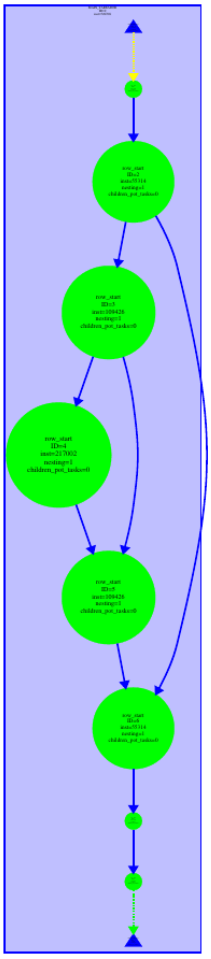


Figure 4: TDG of the program with -h option

What does each chain of tasks in the task graph represents?

An iteration of the outer for (one iteration of the row).

Which part of the code is making the big difference with the two previous cases?

```
if (output2histogram) histogram[k-1]++;
```

Figure 5: code snippet of the program

This part of the code is producing the dependencies between the tasks, as we can see in the TDG above.

How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions?

We can solve the problem by using the #pragma omp atomic, #pragma omp critical or #pragma omp reduction constructors.

Analysis of the potential parallelism for the Point strategy of Mandel-tar.c with no additional options

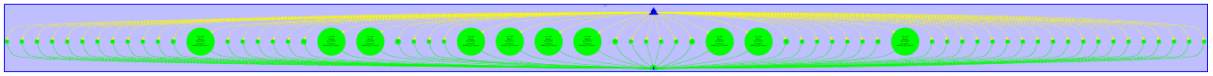


Figure 6: TDG of the program with no additional options

Which are the two most important characteristics for the task graph that is generated?

It has the same characteristics as the row strategy, but the main difference is that there are more tasks created (more granularity).

1. There are no dependencies between the tasks generated, so they are independent. Also we can prove the independence between the tasks by looking at the image before (Figure 6), as we can see that the tasks are arranged horizontally, there are no dependencies.
2. The sizes of the tasks are different, some require more time than others, this is because some of them execute more iterations than the others (the middle ones execute more iterations than the extreme ones, since they diverge faster).

-d option

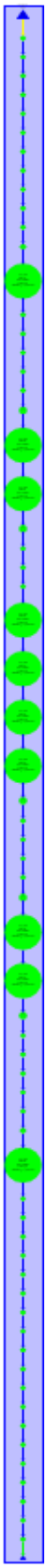


Figure 7: TDG of the program with -d option

Which are the two most important characteristics for the task graph that is generated?

It has the same characteristics as the row strategy, but the main difference is that there are more tasks created (more granularity).

1. As we can see in the dependency graph above (Figure 7), the tasks are arranged vertically, so we can say that the tasks are dependent. This is because the piece of code being added is executed dependently, XSetForeground (sets the color) and XDrawPoint (uses the color that has been set by XSetForeground). So a task starts when the previous one finishes.
2. The sizes of the tasks are different, some require more time than others, this is because some of them execute more iterations than the others (the middle ones execute more iterations than the extreme ones, since they diverge faster).

Which part of the code is making the big difference with the previous case?

```
if (output2display) {
    /* 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);
    }
}
```

Figure 8: code snippet of the program

The instructions XSetForeground and XDrawPoint are the ones that create the dependencies between the tasks.

How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions?

There are different ways to fix this problem. The first option to work around this dependency, i.e. to try to prevent datarace from occurring, is to use the #pragma omp critical constructor. And the other way to solve this problem (more efficient, but more complicated), is to do the calculation first and paint the pixels last.

-h option

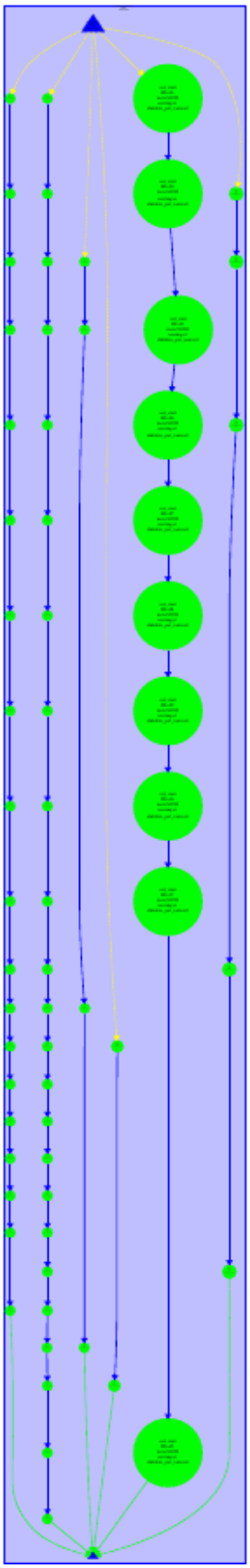


Figure 9: TDG of the program with -h option

What does each chain of tasks in the task graph represents?

An iteration of the inner for (one iteration of the col).

Which part of the code is making the big difference with the two previous cases?

```
if (output2histogram) histogram[k-1]++;
```

Figure 10: code snippet of the program

This part of the code is producing the dependencies between the tasks, as we can see in the TDG above.

How will you protect this section of code in the parallel OpenMP code that you will program in the next sessions?

We can solve the problem by using the #pragma omp atomic, #pragma omp critical or #pragma omp reduction constructors.

After doing the executions for the Row and Point strategies, are conclusions the same?

Yes, we can see that they have the same characteristics and produce the same problems.

Which is the main change that you observe with respect to the Row strategy?

The main difference is that the granularity of the Point strategy is bigger than the Row strategy, so it creates more tasks (a task for each inner iteration of the for), whereas in the Row strategy it only creates a task for each outer iteration of the for, so it has fewer tasks.

Reason when each strategy/granularity should be used.

As mentioned before, the Point strategy creates a high number of tasks compared to the Row strategy. Also we have seen that there are dependencies between the tasks, so some tasks have to wait much time for the others to finish, and the result of this is that the execution time will be increased, apart from the synchronization time and the overheads that may produce. So in conclusion, in this case the Row strategy may be more adequate. The Point strategy can be used if there are no dependencies between the tasks and/or the synchronization time and overheads are few.

Point decomposition in OMP

Point strategy implementation using task

The implementation of the point strategy with granularity control using task consists in defining the corresponding directive right after the inner loop, as shown in the figure below (Figure 11).

```
// Calculate points and generate appropriate output
#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);

        output[row][col]=k;

        if (output2histogram) {
            #pragma omp atomic
            histogram[k-1]++;
        }

        if (output2display) {
            /* 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);
                }
            }
        }
        }
    }
}
```

Figure 11: code snippet of the program

In order to verify the correctness of this version for the display option, a comparison between the display images shown for 1 and 2 threads has been done, and there were indeed no differences. As for the histogram option, it was ensured as well that the output files for 1 and 8 threads were identical to the output file computed with the sequential version, using the

same options in all executions. Since both options were correct, we can assume that the parallelism strategy implemented in this version is also correct.

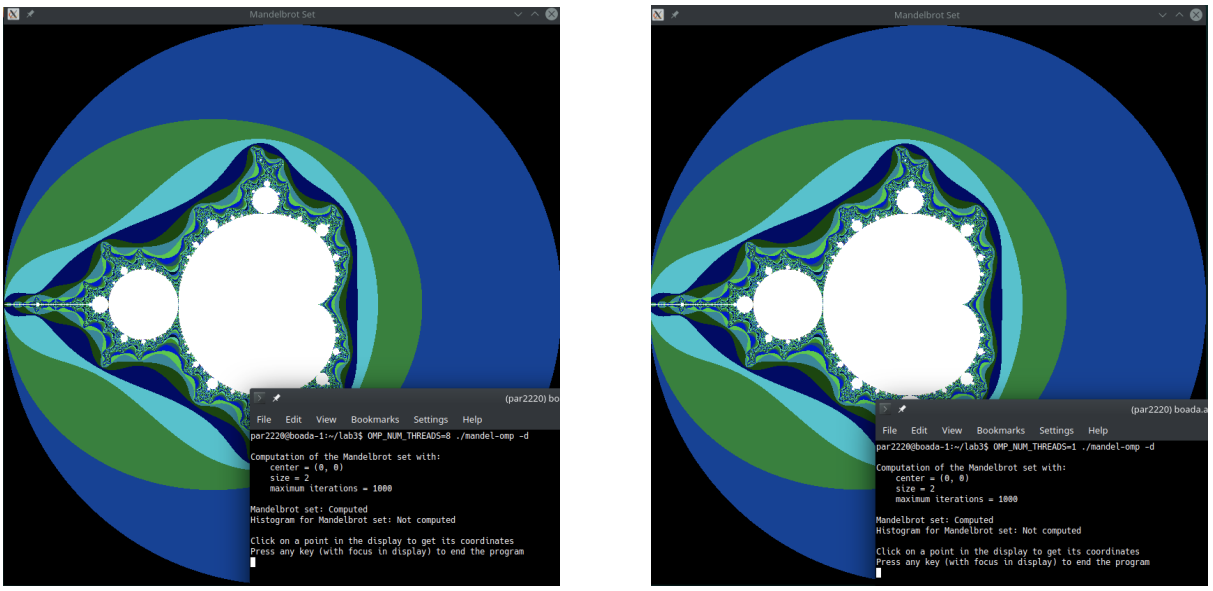


Figure 12: comparison between the display images with 1 and 2 threads

Moving on now to the analysis of performance, the execution time for 1 and 8 threads were:

T_1 = 4.08 s T_8 = 2.74 s

Therefore, the speed-up is:

$S_8 = T_1/T_8 = 2.74$

Now, if we take a look at the scalability plots (Figure 13), it can be observed that the performance is still far from being constant and the scalability is not appropriate, since as can be seen in the scalability graphs below, the speed-up can only reach 2 and remains constant. Therefore, a good result is not obtained.

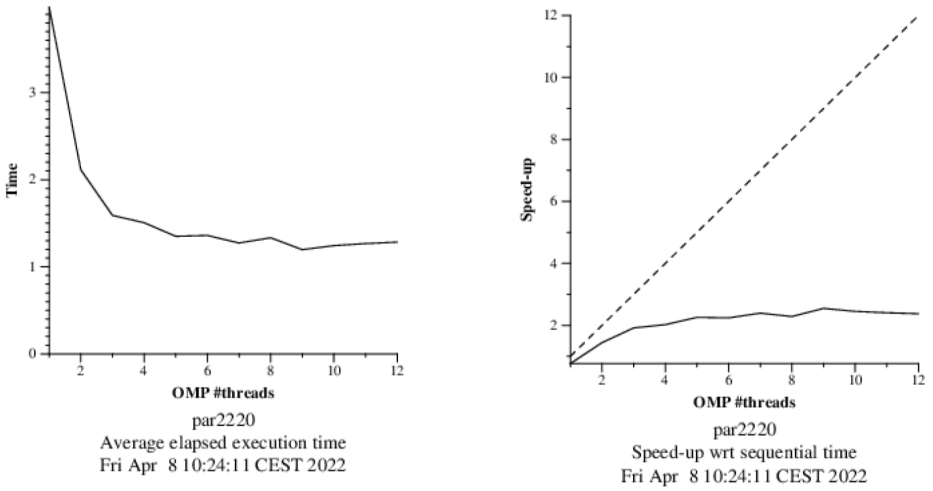


Figure 13: Time and Speed-up graphs depending on the number of threads

Next, we are going to analyze the trace generated (Figure 14) for the execution of this version.

By using the configuration “Explicit tasks function created and executed”, we can identify the thread 1.1.1 as the one who has created all the explicit tasks, which were executed by all the threads created.

Figure 14: Explicit task function execution and explicit task function creation graphs

Since that previous configuration doesn’t give any exact number of tasks generated and executed, we need to check the profile configuration (Figure 15) in order to get further details.

Histogram task instantiation @ mandel-omp-8-bc

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	102,400
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	102,400
Average	102,400
Maximum	102,400
Minimum	102,400
StDev	0
Avg/Max	1

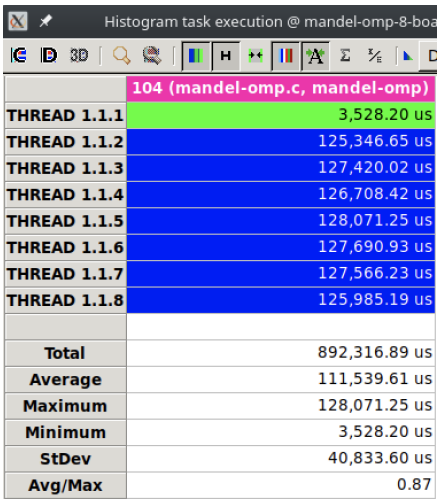
Histogram task execution @ mandel-omp-8-boa

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	64
THREAD 1.1.2	13,764
THREAD 1.1.3	15,200
THREAD 1.1.4	14,113
THREAD 1.1.5	15,418
THREAD 1.1.6	14,150
THREAD 1.1.7	15,450
THREAD 1.1.8	14,241
Total	102,400
Average	12,800
Maximum	15,450
Minimum	64
StDev	4,852.76
Avg/Max	0.83

Figure 15: Histogram task instantiation and histogram task execution

We can observe that the number of tasks instantiated by thread 1.1.1 is 102400 and the total number of tasks executed is also 102400. This means each task consists of 1 iteration.

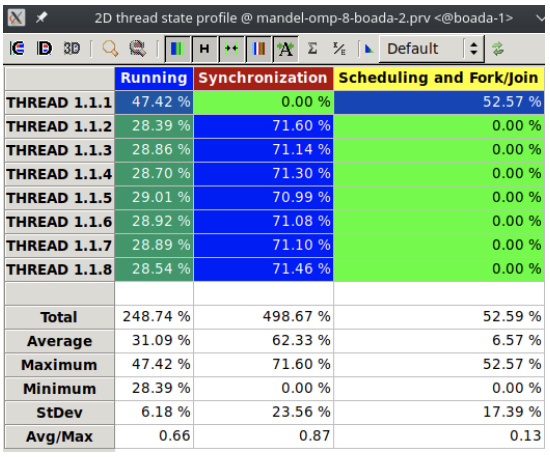
With regard to load balancing, this execution was well balanced both in terms of number of tasks and in terms of total time executing tasks except for thread 1.1.1, which had fewer number of tasks (64) and about 1,2 less seconds than the average for both cases. As we can see, the number of tasks executed is not proportional to the execution time, which leads us to conclude that the measurement which is actually relevant is the execution time, because no matter how balanced the number of tasks executed is, if a certain thread needs much more time for executing the same amount of tasks as the others, the load of the whole parallel region will end up being unbalanced.


Histogram task execution @ mandel-omp-8-boa

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	3,528.20 us
THREAD 1.1.2	125,346.65 us
THREAD 1.1.3	127,420.02 us
THREAD 1.1.4	126,708.42 us
THREAD 1.1.5	128,071.25 us
THREAD 1.1.6	127,690.93 us
THREAD 1.1.7	127,566.23 us
THREAD 1.1.8	125,985.19 us
Total	892,316.89 us
Average	111,539.61 us
Maximum	128,071.25 us
Minimum	3,528.20 us
StDev	40,833.60 us
Avg/Max	0.87

Figure 16: Histogram task execution

Talking now about the overheads related with synchronization and task scheduling, in the configuration “Thread state profile” we can check the percentage of runtime where synchronization and scheduling had been done for each thread. On average, about 70% of runtime was used for synchronization, and over 52% for scheduling in the case of thread 1.1.1. This means that the task granularity is not appropriate, because the synchronization time lasts longer than the running time. Moreover, changing this granularity is easy because we can change task for taskloop for example, and the granularity will be increased.


2D thread state profile @ mandel-omp-8-boada-2.prv <@boada-1>

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	47.42 %	0.00 %	52.57 %
THREAD 1.1.2	28.39 %	71.60 %	0.00 %
THREAD 1.1.3	28.86 %	71.14 %	0.00 %
THREAD 1.1.4	28.70 %	71.30 %	0.00 %
THREAD 1.1.5	29.01 %	70.99 %	0.00 %
THREAD 1.1.6	28.92 %	71.08 %	0.00 %
THREAD 1.1.7	28.89 %	71.10 %	0.00 %
THREAD 1.1.8	28.54 %	71.46 %	0.00 %
Total	248.74 %	498.67 %	52.59 %
Average	31.09 %	62.33 %	6.57 %
Maximum	47.42 %	71.60 %	52.57 %
Minimum	28.39 %	0.00 %	0.00 %
StDev	6.18 %	23.56 %	17.39 %
Avg/Max	0.66	0.87	0.13

Figure 17: 2D thread state profile

Point strategy with granularity control using taskloop

The implementation of the point strategy with granularity control using taskloop consists in defining the corresponding directive right before the inner loop, as shown in the figure below.

```
// Calculate points and generate appropriate output
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop
    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
                                   */

        // 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);

        output[row][col]=k;

        if (output2histogram) {
            #pragma omp atomic
            histogram[k-1]++;
        }

        if (output2display) {
            /* 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);
                }
            }
        }
    }
}
```

Figure 18: code snippet of the program

In order to verify the correctness of this version for the display option, a comparison between the display images shown for 1 and 2 threads has been done, and there were indeed no differences. As for the histogram option, it was ensured as well that the output files for 1 and 8 threads were identical to the output file computed with the sequential version, using the same options in all executions. Since both options were correct, we can assume that the parallelism strategy implemented in this version is also correct.

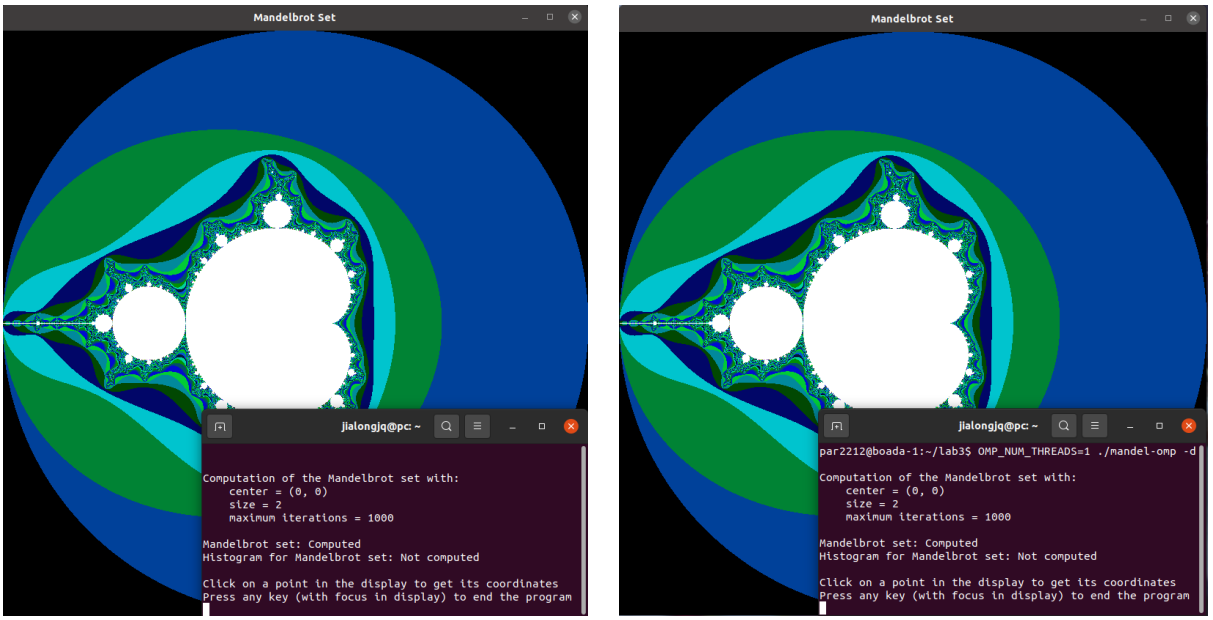


Figure 19: comparison between the display images with 1 and 2 threads

Moving on now to the analysis of performance, the execution time for 1 and 8 threads were:

T1 = 3.16 s T8 = 0.70 s

Therefore, the speed-up is:

S8 = T1/T8 = 4.5

At a first glance, we can say that the speed-up in this case is more appropriate than the one calculated for the previous version, which had an implementation using the task directive. Now, if we take a look at the scalability plots, it can be observed that although the improvement in performance is still far from being constant, the taskloop version is already performing much better than the task version.

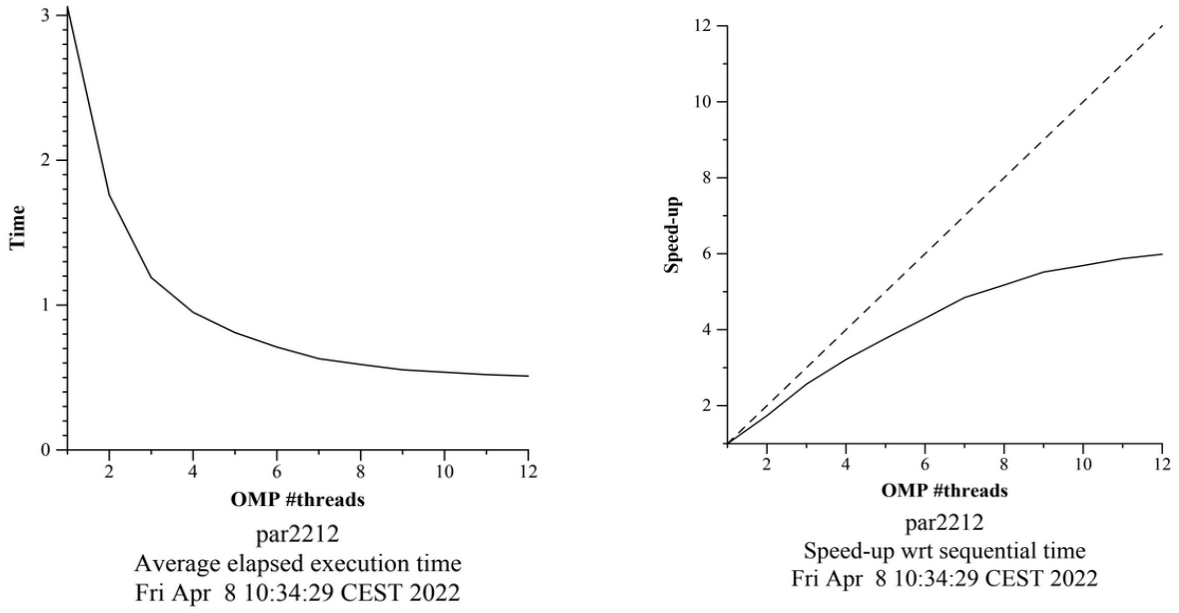


Figure 20: Time and Speed-up graphs depending on the number of threads

Next, we will analyze the trace generated for the execution of this version.

By using the configuration “Explicit tasks function created and executed”, we can identify the thread 1.1.1 as the one who has created all the explicit tasks, which were executed by all the threads created.

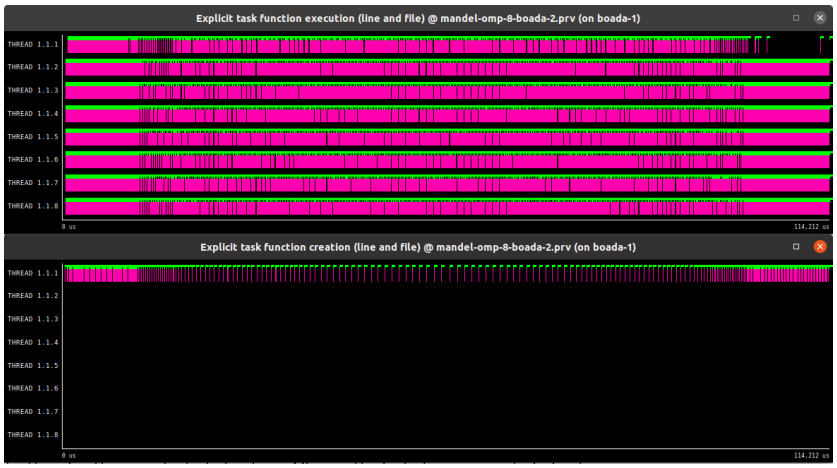


Figure 21: Explicit task function execution and explicit task function creation graphs

Since that previous configuration doesn’t give any exact number of tasks generated and executed, we need to check the profile configuration in order to get further details.

Histogram task instantiation @ mandel-omp-8-boada-2.pr

	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	320
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	320
Average	320
Maximum	320
Minimum	320
StDev	0
Avg/Max	1

Histogram task execution @ mandel-omp-8-boada-2.prv ..

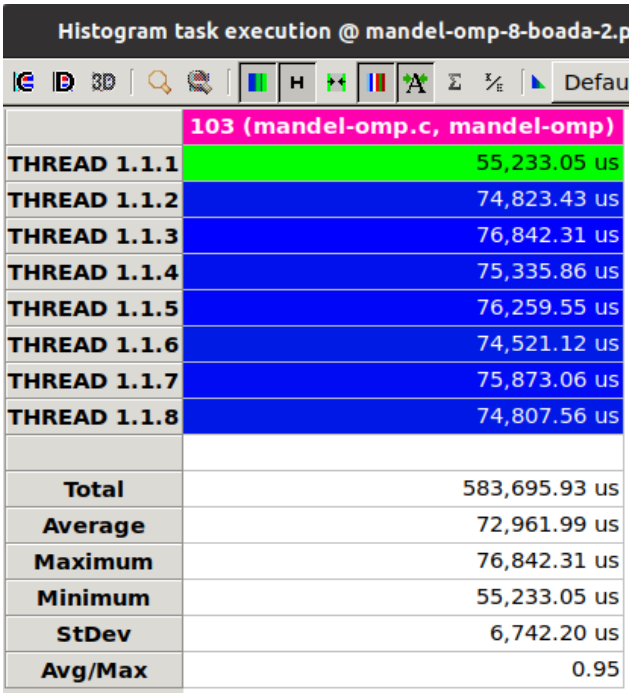
	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	4,231
THREAD 1.1.2	3,046
THREAD 1.1.3	3,021
THREAD 1.1.4	3,073
THREAD 1.1.5	3,112
THREAD 1.1.6	3,023
THREAD 1.1.7	3,056
THREAD 1.1.8	3,038
Total	25,600
Average	3,200
Maximum	4,231
Minimum	3,021
StDev	390.66
Avg/Max	0.76

Figure 22: Histogram task instantiation and histogram task execution

We can observe that the number of tasks instantiated by thread 1.1.1 is 320 (since there is one task instantiation per taskloop, and a total of 320 rows) and the total number of tasks executed is 25600. This means that 25600/320 = 80 tasks are executed per taskloop (num_tasks = 80), each task consisting of 320*320/25600 = 4 iterations (grainsize = 4).

Since we did not specify neither num_tasks nor grainsize, the granularity was arbitrarily specified depending on the number of threads in the parallel region. To check this, we generated different traces with 1 and 4 threads, and as a result, 3200/320 = 10 tasks per taskloop were generated with 1 thread, and 12800/320 = 40 tasks per taskloop in the case of 4 threads. Thus, we can conclude that with no granularity specified, num_tasks was implicitly specified as the number of threads in the parallel region multiplied by 10.

With regard to load balancing, this execution was well balanced both in terms of number of tasks and in terms of total time executing tasks except for thread 1.1.1, which had approximately 1000 more tasks executed and about 0,02 less seconds than the average for both cases. As we can see, the number of tasks executed is not proportional to the execution time, which leads us to conclude that the measurement which is actually relevant is the execution time, because no matter how balanced the number of tasks executed is, if a certain thread needs much more time for executing the same amount of tasks as the others, the load of the whole parallel region will end up being unbalanced.



	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	55,233.05 us
THREAD 1.1.2	74,823.43 us
THREAD 1.1.3	76,842.31 us
THREAD 1.1.4	75,335.86 us
THREAD 1.1.5	76,259.55 us
THREAD 1.1.6	74,521.12 us
THREAD 1.1.7	75,873.06 us
THREAD 1.1.8	74,807.56 us
Total	583,695.93 us
Average	72,961.99 us
Maximum	76,842.31 us
Minimum	55,233.05 us
StDev	6,742.20 us
Avg/Max	0.95

Figure 23: Histogram task task execution

Talking now about the overheads related with synchronization and task scheduling, in the configuration “Thread state profile” we can check the percentage of runtime where synchronization and scheduling had been done for each thread. On average, over 33% of runtime was used for synchronization, and over 24% for scheduling in the case of thread 1.1.1. This means that the task granularity has helped to reduce drastically the overhead produced, being now much more appropriate than the previous version. Moreover, changing this granularity is easy since it can be directly specified in the taskloop constructor.

2D thread state profile @ mandel-omp-8-boada-2.prv (on boada-1)

Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	49.84 %	26.65 %	23.51 %
THREAD 1.1.2	65.75 %	34.24 %	0.01 %
THREAD 1.1.3	67.52 %	32.47 %	0.01 %
THREAD 1.1.4	66.20 %	33.79 %	0.01 %
THREAD 1.1.5	67.01 %	32.98 %	0.01 %
THREAD 1.1.6	65.48 %	34.51 %	0.01 %
THREAD 1.1.7	66.67 %	33.32 %	0.01 %
THREAD 1.1.8	65.73 %	34.26 %	0.01 %
Total	514.20 %	262.23 %	23.57 %
Average	64.28 %	32.78 %	2.95 %
Maximum	67.52 %	34.51 %	23.51 %
Minimum	49.84 %	26.65 %	0.01 %
StDev	5.49 %	2.41 %	7.77 %
Avg/Max	0.95	0.95	0.13

Figure 24: 2D thread state profile

To determine where the overhead caused by synchronization is coming from, we can zoom in into the timelines of “In taskgroup construct” and “Explicit task function execution”. As we can observe, since a taskgroup is implicitly being created in each instantiation of taskloop, the timelines consist of groups of different tasks, where a certain group cannot start its execution if the tasks in the previous one have not ended yet. Therefore, the task synchronization is happening right after the inner loop, so the next taskloop won’t be executed until all threads have arrived at this point.

Figure 25: Taskgroup construct and explicit task function execution

These task barriers, however, are not really useful, since all dependencies between tasks are already being protected. For this reason, we have tested how the performance would change if we remove them. This implementation can be easily done by modifying the taskloop constructor, being now “#pragma omp taskloop firstprivate(row) nogroup” (privatizing the row variable is necessary since dataracing might be caused now). As a result, the efficiency per thread has improved up to 25% thanks to the reduction of task synchronization needed, as can be seen in Figure 27. In Figure 28, we can observe that the scalability has slightly improved as well.

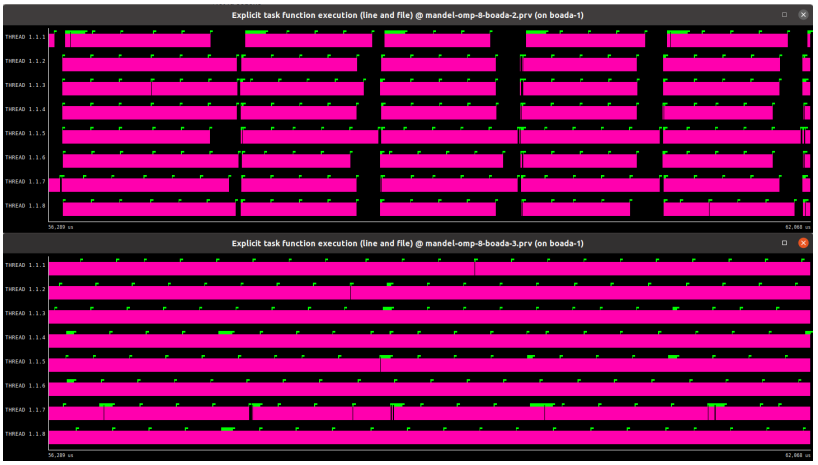


Figure 26: Explicit task function execution

2D thread state profile @ mandel-omp-8-boada-3.prv (on boada-1)

Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	89.33 %	10.65 %	0.02 %
THREAD 1.1.2	89.12 %	10.87 %	0.01 %
THREAD 1.1.3	88.95 %	11.04 %	0.01 %
THREAD 1.1.4	89.21 %	10.79 %	0.00 %
THREAD 1.1.5	89.14 %	10.86 %	0.01 %
THREAD 1.1.6	89.15 %	10.85 %	0.00 %
THREAD 1.1.7	79.95 %	0.10 %	19.96 %
THREAD 1.1.8	89.14 %	10.85 %	0.00 %
Total	703.99 %	76.00 %	20.01 %
Average	88.00 %	9.50 %	2.50 %
Maximum	89.33 %	11.04 %	19.96 %
Minimum	79.95 %	0.10 %	0.00 %
StDev	3.04 %	3.56 %	6.60 %
Avg/Max	0.99	0.86	0.13

Figure 27: 2D thread state profile

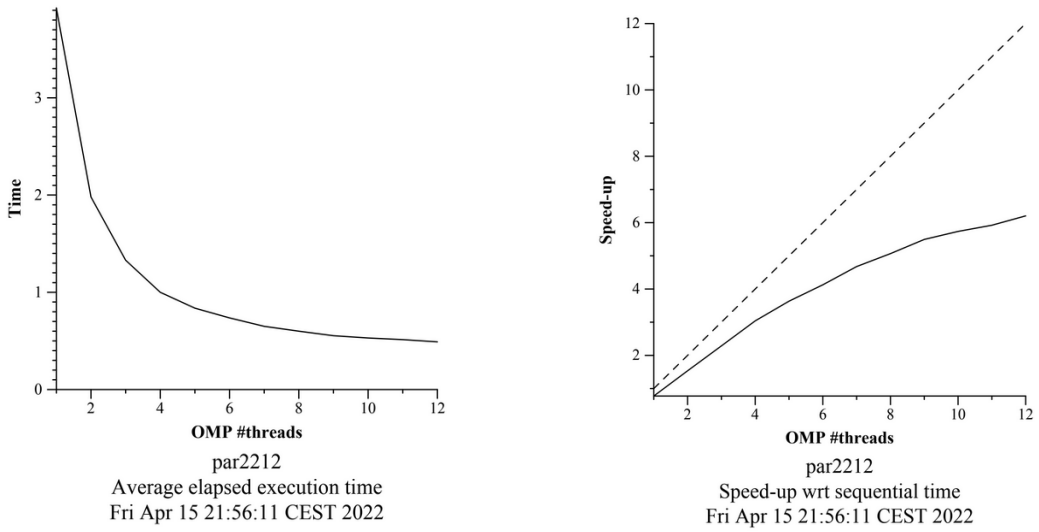


Figure 28: Time and Speed-up graphs depending on the number of threads

Row strategy implementation

As we can observe in the graphics below of time and speed-up, although there is not any big difference between both versions, by implementing the task decomposition strategy we can obtain better results than using the taskloop strategy, since for the task strategy the scalability is more uniform.

If we compare the row and point strategies, it is clear that the row strategy has a much better scalability, since there is a closer approach to the ideal scalability.

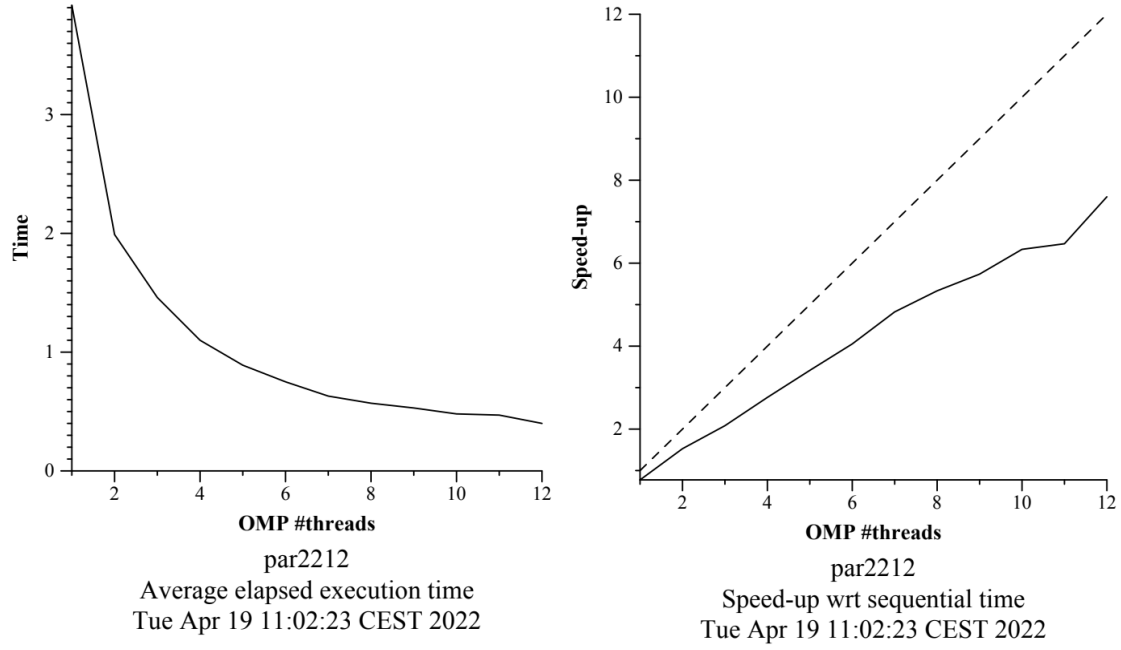


Figure 29: Time and Speed-up graphs depending on the number of threads with taskloop

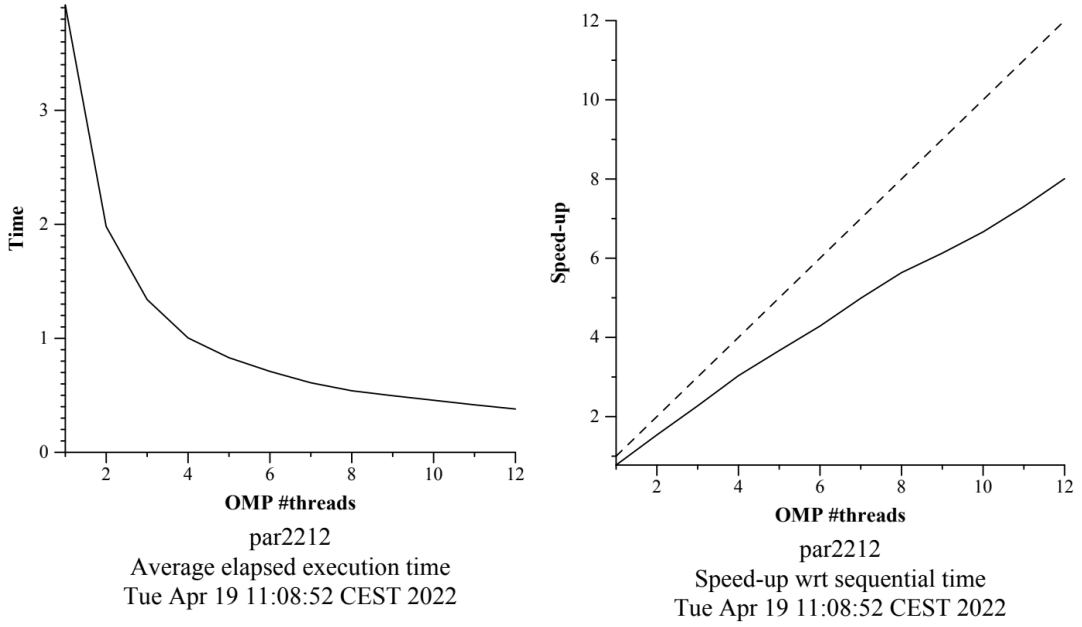


Figure 30: Time and Speed-up graphs depending on the number of threads with task

This variation of growth in the scalability of the taskloop version is due mainly to the inefficiency caused by the unbalance in workload. Since the granularity of tasks is arbitrarily assigned and the cost of completing each row iteration is not always the same, the load balance cannot be controlled and there is a chance where some threads get more workload than the others. For the task version, there is not such a problem because it uses a finer granularity so better efficiency can be achieved. This performance difference can be visualized in the traces shown in Figures 31 and 33, where the regions in black are regions where a thread is waiting for the others to finish their tasks.

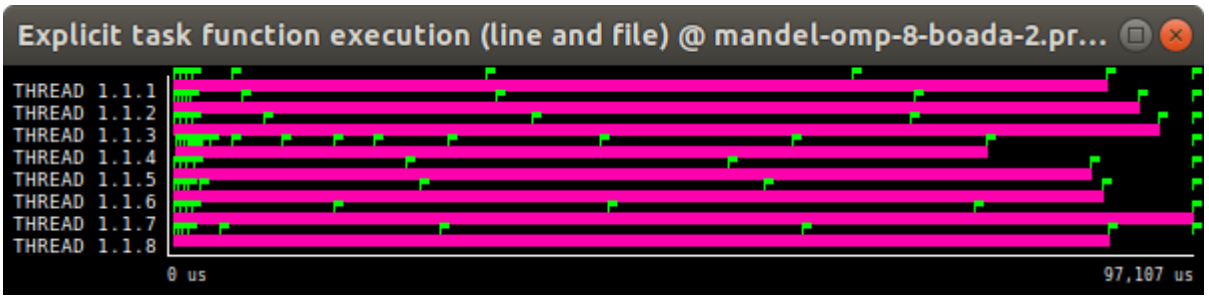


Figure 31: Explicit task function execution

2D thread state profile @ mandel-omp-8-boada-2.prv #3 (on boada-1)

Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	91.36 %	8.62 %	0.02 %
THREAD 1.1.2	94.55 %	5.44 %	0.01 %
THREAD 1.1.3	96.47 %	3.52 %	0.01 %
THREAD 1.1.4	79.33 %	20.48 %	0.19 %
THREAD 1.1.5	89.84 %	10.15 %	0.01 %
THREAD 1.1.6	90.91 %	9.09 %	0.01 %
THREAD 1.1.7	99.83 %	0.16 %	0.01 %
THREAD 1.1.8	91.63 %	8.36 %	0.01 %
Total	733.92 %	65.82 %	0.26 %
Average	91.74 %	8.23 %	0.03 %
Maximum	99.83 %	20.48 %	0.19 %
Minimum	79.33 %	0.16 %	0.01 %
StDev	5.64 %	5.59 %	0.06 %
Avg/Max	0.92	0.40	0.17

Figure 32: 2D thread state profile

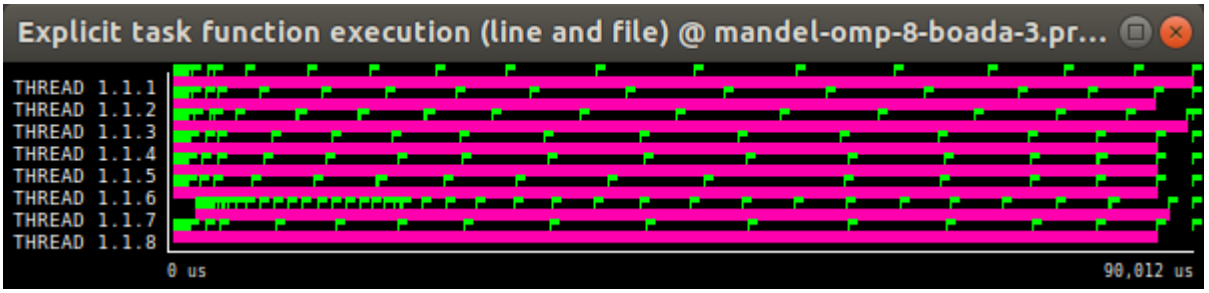


Figure 33: Explicit task function execution

2D thread state profile @ mandel-omp-8-boada-3.prv #1 (on boada-1)

Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	99.73 %	0.24 %	0.03 %
THREAD 1.1.2	96.03 %	3.96 %	0.01 %
THREAD 1.1.3	99.12 %	0.87 %	0.01 %
THREAD 1.1.4	96.11 %	3.88 %	0.01 %
THREAD 1.1.5	96.18 %	3.81 %	0.01 %
THREAD 1.1.6	96.28 %	3.72 %	0.01 %
THREAD 1.1.7	95.67 %	2.95 %	1.38 %
THREAD 1.1.8	96.16 %	3.84 %	0.01 %
Total	775.27 %	23.27 %	1.46 %
Average	96.91 %	2.91 %	0.18 %
Maximum	99.73 %	3.96 %	1.38 %
Minimum	95.67 %	0.24 %	0.01 %
StDev	1.47 %	1.40 %	0.45 %
Avg/Max	0.97	0.73	0.13

Figure 34: 2D thread state profile

From these comparisons between taskloop and task versions, we can conclude that grouping iterations depending on the number of threads using taskloop can cause a workload unbalance, which can be solved by reducing the grain size.

Optional

First of all, after looking at the script submit-numtasks-omp.sh, we have discovered that it passes the value of the number of tasks after the option -u, so in the mandel-omp.c we have added the next constructor: #pragma omp taskloop num_tasks(user_param), since the argument after the -u parameter is assigned to the global variable user_param.

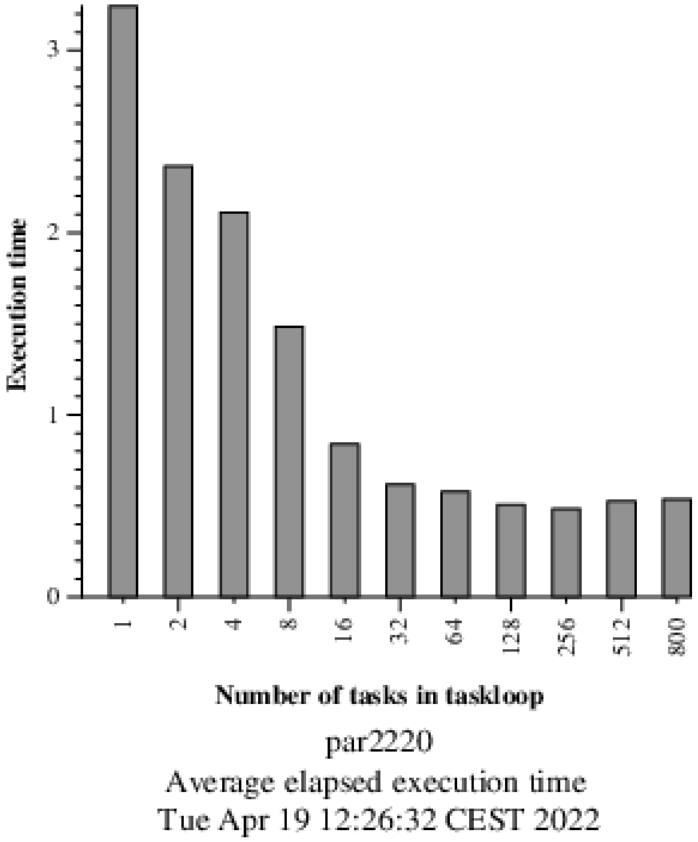


Figure 35: graph of granularities with point strategy

We can see in figure 35 that by increasing the number of tasks in taskloop, the execution time decreases, but only to a certain point (in this case 256). After that, the execution time rises up again, due to overheads caused by synchronization time.

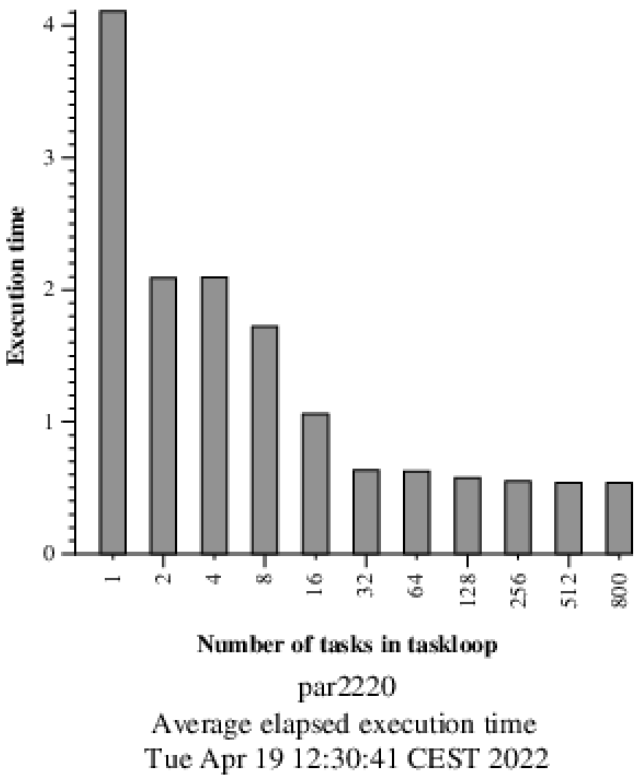


Figure 36: graph of granularities with point strategy

In this case, we can observe in Figure 36 that the execution time always decrease by increasing the number of tasks in taskloop and after 256 it doesn't increase. This is because there are only 320 iterations in the outer loop, so it is the same having 256, 512 or 800 number of tasks assigned to a taskloop, because of that the execution time keeps constant.

Conclusions

Through this assignment we have explored the different ways to achieve parallelism for the Mandelbrot set computation, from a point strategy to a row strategy.

First, we have studied the dependencies between tasks and the potential parallelism for both strategies using Tareador, which has allowed us to detect the instructions where datarace could be produced, in order to protect them later for the implementation in OpenMP using the most suitable constructs.

In the implementation of the point strategy, three versions have been implemented, one with task and the other two with taskloop. For the task version we have seen that a lot of overhead was produced due to synchronization and scheduling of tasks, which was later reduced with the taskloop version, achieving a better efficiency per thread. Since no grouping was required in this program, for the third version the barriers were removed to improve it even more.

For the row strategy, first we have observed that implementing task decomposition is better than taskloop decomposition strategy, due to the inefficiency caused by the unbalanced workload. After that, we have compared the best option of row strategy (task decomposition) with the best option of point strategy (the one with taskloop and without the barriers). By analyzing the results, we have seen that the row strategy has a better performance and its scalability is nearer to the ideal.

Finally, we have also tested the influence of the number of tasks in both versions, which lead us to conclude that better performance can be achieved if more tasks are instantiated, but there is a point where it can end up being counterproductive, due to the overhead that may be produced.

The most important conclusion to which we have arrived is that it is important to take into account the space locality while implementing parallelism for a program. For the Mandelbrot set, we have seen that by implementing the point strategy with task, a lot of overhead was caused mainly because of false sharing, since there was a possibility where multiple threads needed to access data close to each other. By contrast, the taskloop constructor allowed us to assign groups of iterations to different tasks, which accomplished a better management of space locality.