Laboratory Assignment 3

PAR - Q2 2019/2020

Marc Monfort Grau
Albert Mercadé Plasencia
Group par2302

March 26th 2020

Index

Introduction

In this document, we explore two task decomposition strategies: point decomposition and row decomposition. We do so by analysing the potential parallelisation of the computation of the Mandelbrot set using both strategies and evaluating their performance to establish which one is more appropriate for this calculation. Furthermore, we also look at another strategy that stems from row decomposition, called for-based decomposition.

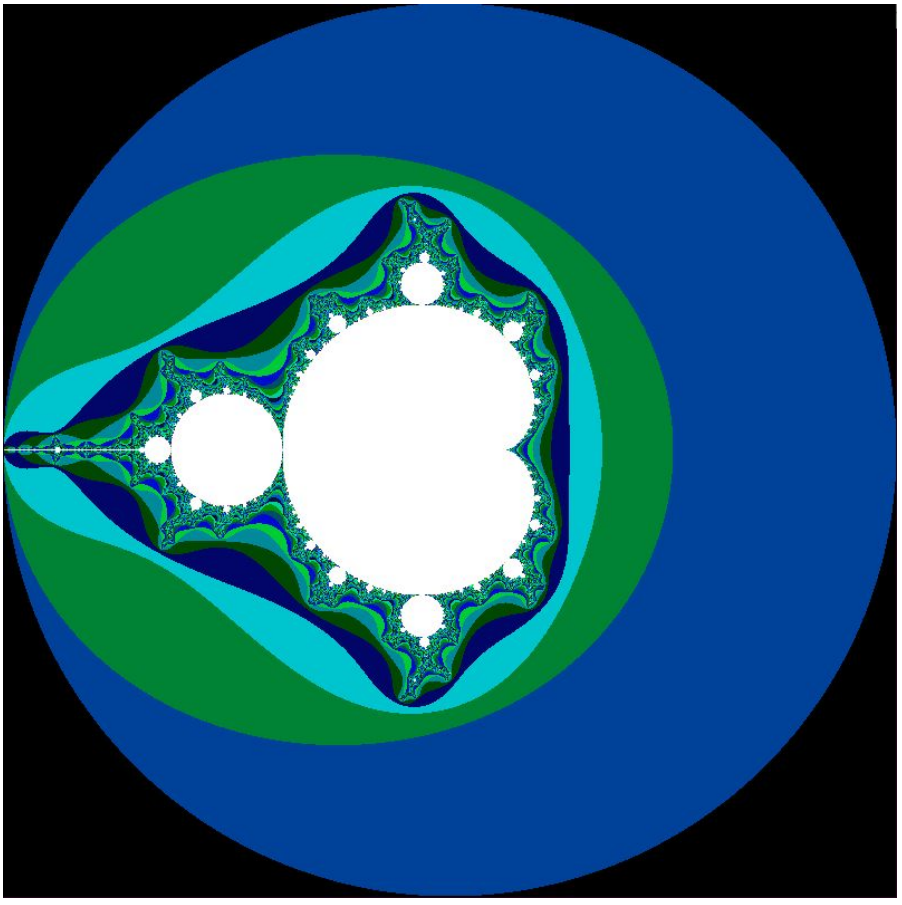


Image 1. Mandelbrot visualization

Parallelization strategies

Task decomposition and granularity analysis

In order to complete `mandel-tar.c` to analyze the potential parallelism of both granularity levels, we need to add `tareador_start_task(...)` and `tareador_end_task(...)` at the desired lines of the code.

```
for (row = 0; row < height; ++row) {
       for (col = 0; col < width; ++col) {
           tareador_start_task("Point_task");
           //code...
           tareador_end_task("Point_task");
       }
}
```

Figure 1. Point task decomposition

For point task decomposition, we need to invoke the previously mentioned *Tareador* task creation functions within the innermost `for` loop, as seen in Figure 1. This way we are signalling *Tareador* that we want to create a task for each point in the Mandelbrot set computation. We'll be able to analyse the dependencies and possible parallelisation with this approach.

```
for (row = 0; row < height; ++row) {
       tareador_start_task("Row_task");
       for (col = 0; col < width; ++col) {
           //code...
       }
       tareador_end_task("Row_task");
}
```

Figure 2. Row task decomposition

Above in Figure 2, we see what the code for row task decomposition looks like for *Tareador* analysis. Since we want to create a task per row, we call *Tareador*'s task creating method before entering the innermost loop, which traverses the rows.

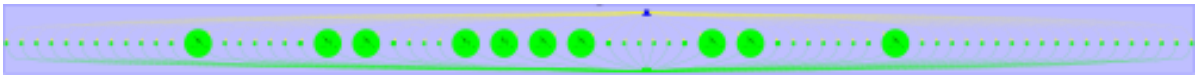


Figure 3. Non-graphical dependency graph (Point)

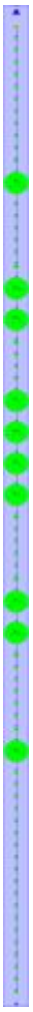


Figure 4. Graphical graph dependency (Point)

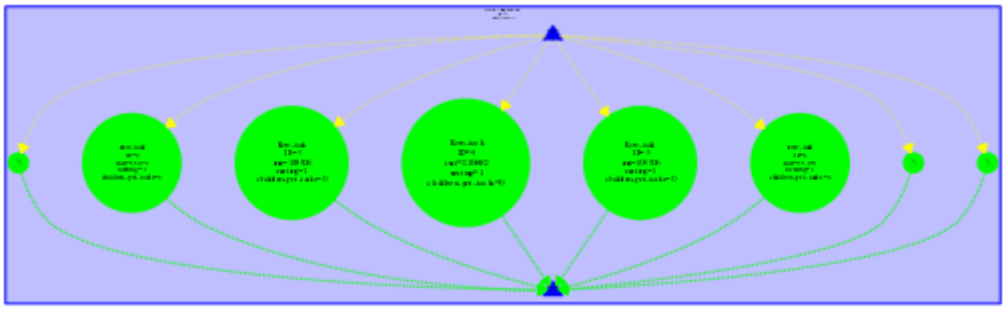


Figure 5. Non-graphical graph dependency (Row)

Figure 6. Graphical graph dependency (Row)

The previous four figures obtained from *Tareador* show us the task dependency graphs for both row and point task decomposition in their graphical and non-graphical variants. In the non-graphical version, we see that for both decomposition strategies the tasks show no dependency between them and thus signalling that this is a very parallelizable program. Secondly, we also observe that in both cases the middle nodes of the dependency graph have more granularity than the nodes on the sides. This could be due to the points/rows calculated in those tasks belong to the fractal shape and hence require to iterate to the maximum number of iterations.

The difference that strikes as most obvious is that the graphical version both for Row and Point decomposition is sequential while the non-graphic version shows no dependency between tasks. As we see in Figure 7, the serialisation of the graphical version is caused by the displaying of the points in the Mandelbrot set using the X11 protocol. Hence, the problem is located in the `XSetForeground` and `XDrawPoint` methods, in Figure 8 we see the region of the code where these are found. To protect this section of the code, we will use `#pragma omp critical` around it, as seen in Figure 9. This way, only one thread at a time will execute this section, as required, but the rest of the threads can continue executing the rest of the code.

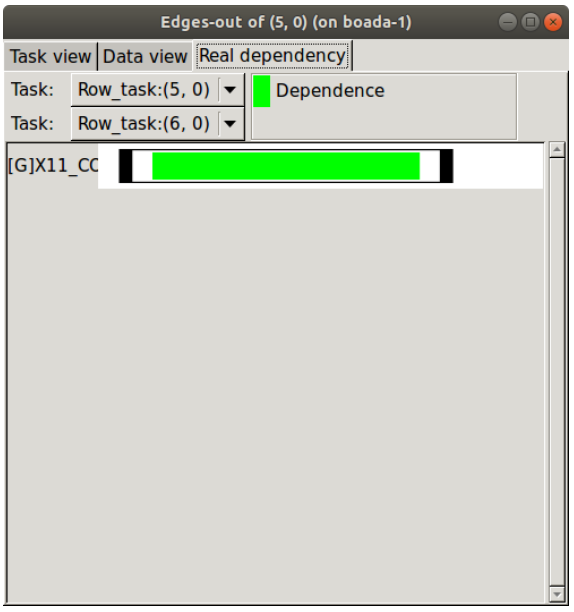


Figure 7. Variable problem dependency

```
#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
```

Figure 8. Dependency problem

```
#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
```

Figure 9. Solution dependency problem

We expect row decomposition to provide a better performance as it will incur in less task creation and synchronisation overhead while still exploiting significant parallelisation.

Point decomposition in OpenMP

Starting from `mandel-omp-v0.c`, which is a copy of the original `mandel-omp.c`, we create `mandel-omp-v1.c` by adding `#pragma omp critical` to honour the detected dependencies in the graphical version. Thus, in v1, the only code section that changes with respect to v0 is the one shown in Figure 9.

After we execute v1 in its graphical version, we observe that the image generated from `mandeld-omp` with only one thread is correct. Its execution time is slightly higher than that of the sequential version, as we can see in Table 1 below, which is explained by the overheads incurred in task creation. Also, as we would expect, the execution time of v0 and v1 are practically the same as the only difference is the `critical` clause, and thus the overheads are the same.

When we execute v1, now with 8 threads, the image is still correct, but the execution time has increased. That's predictable because of the task dependency we saw previously, where the execution is sequential, but now with even more overhead caused by the extra threads. Furthermore, v0 fails to execute due to not having the `#pragma omp critical` clause.

`mandeld` (graphical)	**1 thread**	**8 threads**
`mandel-seq.c`	3.052365s	-
`mandel-omp-v0.c`	3.324979s	FAIL/ERROR
`mandel-omp-v1.c`	3.328787s	4.048273s

Table 1. `mandeld` ***execution times for*** `mandel-seq.c`**,** `mandel-omp-v0.c` ***and*** `mandel-omp-v1.c`

We repeated the same executions as before with 1 and 8 threads but this time with `mandel` (non-graphical) and obtained the results seen in Table 2. As we observe in the table, the execution times with 1 thread don't change at all when compared to the graphical version, which makes sense as the execution is sequential and we aren't exploiting any parallelism. However, when we execute `mandel` using 8 threads, we do see a significant reduction in execution times. This is explained by the fact that with this non-graphical version there is no dependency between tasks and thus we benefit from parallelism. Again, the execution times for v0 and v1 are virtually identical both for 1 and 8 threads, that is because for the non-graphical version they have the exact same code.

`mandel` (non-graphical)	**1 thread**	**8 threads**
`mandel-seq.c`	3.037388s	-

`mandel-omp-v0.c`	3.321079s	1.217835s
`mandel-omp-v1.c`	3.310772s	1.232019s

Table 2. `mandel` ***execution times for*** `mandel-seq.c`, `mandel-omp-v0.c` ***and*** `mandel-omp-v1.c`

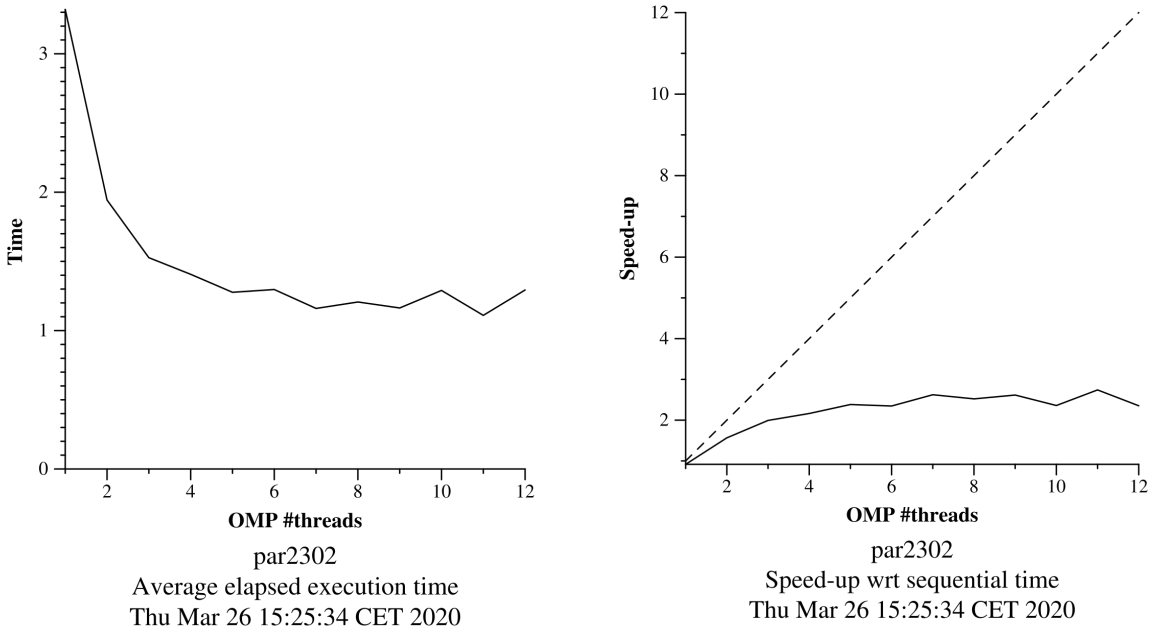


Figure 10. Scalability of v1

In Figure 10 the strong scalability of the mandel-omp version is shown. We see how the execution time after using 4 threads no longer decreases. It even starts increasing when we surpass 10 threads because of the increase in overhead time. Thus, the scalability isn't that great for v1.

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	70,885	248,000
THREAD 1.1.2	83,442	37,600
THREAD 1.1.3	67,622	76,000
THREAD 1.1.4	90,445	151,200
THREAD 1.1.5	83,222	25,600
THREAD 1.1.6	83,531	1,600
THREAD 1.1.7	70,103	10,400
THREAD 1.1.8	90,750	89,600
Total	640,000	640,000
Average	80,000	80,000
Maximum	90,750	248,000
Minimum	67,622	1,600
StDev	8,613.98	78,381.63
Avg/Max	0.88	0.32

Figure 11. Configuration file `OMP_task_profile.cfg` ***for v1***

As we can see in Figure 11, in total there are 640,000 tasks created/executed. In the code of v1, the `parallel` and `single` clauses are inside the first loop, which loops through the rows. Hence, both constructs are called 800 times, once for each of the 800 rows in our calculation of the Mandelbrot set. For the same reason, whichever thread "arrives" first to the `single` construct will create the following 800 tasks, as there are 800 points in each row. In our execution, threads 1, 3, 4 and 8 instantiate the bulk of the tasks created but when it comes to their execution all threads execute a similar same amount, in fact the coefficient of variation is quite low at $8,613.98 \div 80,000 = 0.11$.

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
   for (col = 0; col < width; ++col) {
       #pragma omp task firstprivate(row, col)
       {
           //code ...
       }
   }
   #pragma omp taskwait // waiting point for all child tasks
}
```

Code: `mandel-omp-point-v2.c`

For `mandel-omp-point-v2.c` the three changes in the code are that we move the `parallel` and `single` constructs outside the `for` loop, we intrdouce the `taskwait` synchronisation construct after the innermost `for` loop and we now also make the `row` variable `firstprivate`. This way, only one thread creates the tasks and, due to the taskwait, only after all the tasks of the previous row are executed are the tasks of the next row created.

Below in Figure 12, we can observe that the scalability of v2 is remarkably similar to that of v1, and we see no clear improvement. The execution time stabilises when using more than 4 threads and, hence, so does the speed-up plot.

As a consequence of moving out of the first loop both the `parallel` and `single` constructs, they are only called once before entering the `for` loop. The `taskwait` construct is executed 800 times, once for every row in this Mandelbrot set computation. The total number of tasks created and executed are 640,000, as we can note looking at Figure 13 beneath. Furthermore, as a result of the `single` construct being outside the loop, only one thread will create all tasks, in this case thread 3. The tasks are evenly executed by all threads, even more so than v1 as the coefficient of variation is $4,743.71 \div 80,000 = 0.08$. Since we are still creating a task per each point in the set, the granularity of tasks remains the same.

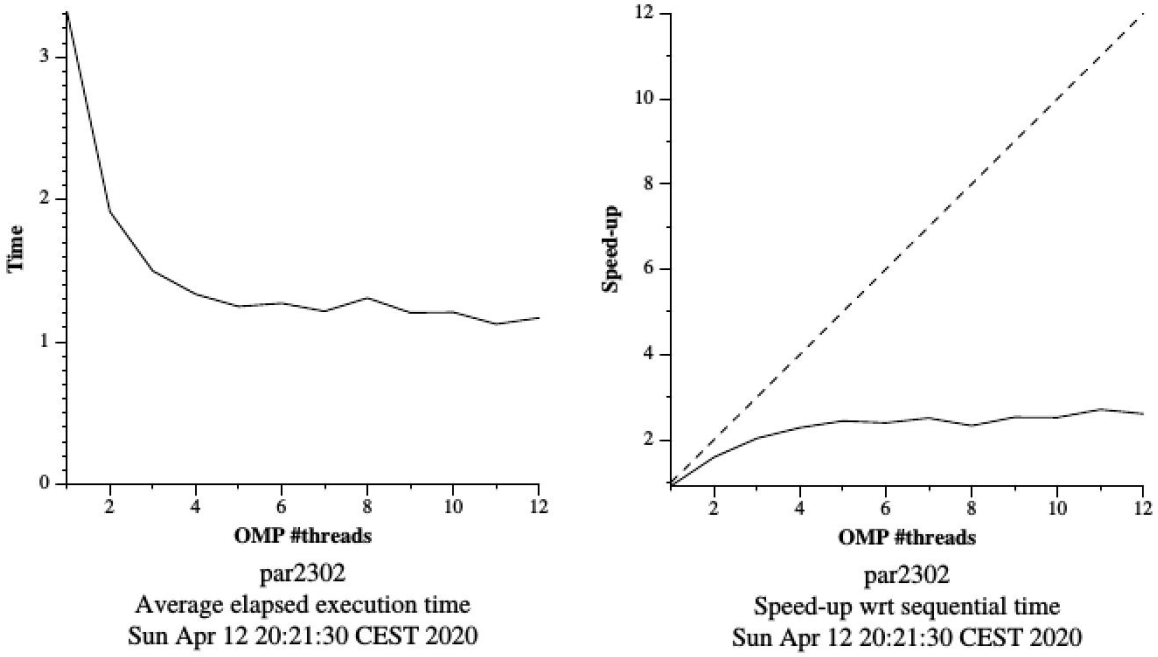


Figure 12. Strong scalability for v2

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	74,009	-
THREAD 1.1.2	83,328	-
THREAD 1.1.3	89,028	640,000
THREAD 1.1.4	68,716	-
THREAD 1.1.5	82,483	-
THREAD 1.1.6	82,886	-
THREAD 1.1.7	84,098	-
THREAD 1.1.8	75,452	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	89,028	640,000
Minimum	68,716	640,000
StDev	6,202.55	0
Avg/Max	0.90	1

Figure 13. Configuration file `OMP_task_profile.cfg` *for v2*

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
   #pragma omp taskgroup
   {
       for (col = 0; col < width; ++col) {
           #pragma omp task firstprivate(row, col)
           {
               //code ...
           }
```

```
        }
    } // waiting point for all descendant tasks in taskgroup region
}
```

Code: `mandel-omp-point-v3.c`

The third variation of point decomposition (`mandel-omp-point-v3.c`) builds on v2 and swaps the `taskwait` for a `taskgroup` synchronisation construct that encloses the innermost loop. Timing-wise we observe that v2 is slightly faster than v3, but marginally so. If we look at behaviour, both versions execute similarly with the only difference being that in v2 all tasks are instantiated by thread 3 and in v3 it is thread 1. We can see this clearly in Figures 14 and 15. This difference in behaviour can be explained by the fact that the `single` construct chooses any thread, not necessarily the master thread, and hence we cannot predict which one will be chosen. Looking at Figure 16, we also observe that task execution is also evenly distributed between all threads just like it was for v2. Thus, this only difference is causeb by the `single` construct and has nothing to do with changing `taskwait` for `taskgroup` in our code.

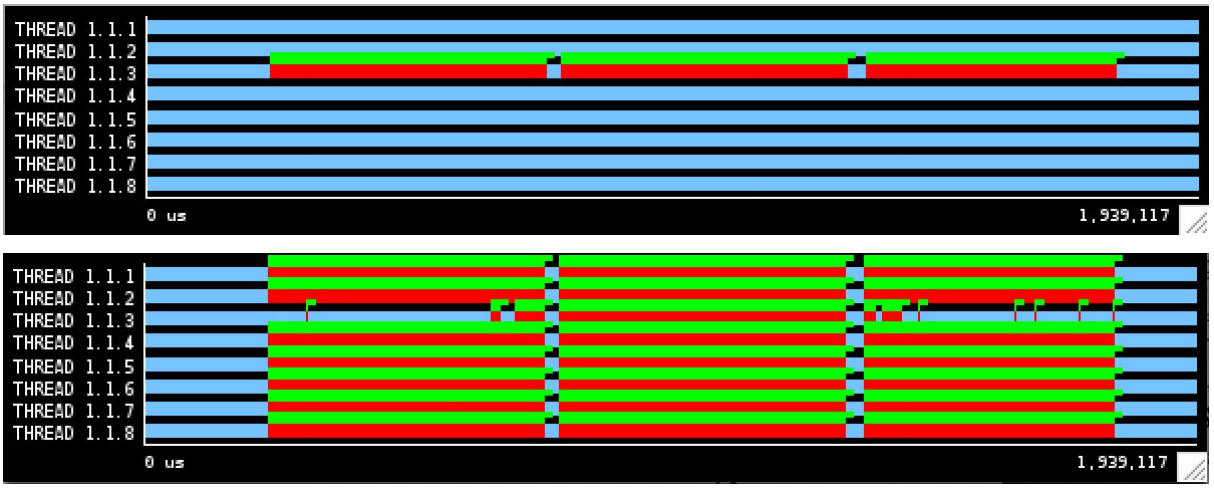
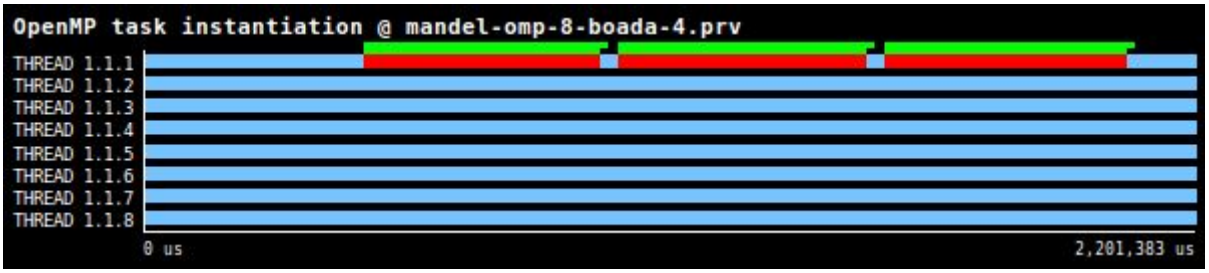


Figure 14. `OMP_tasks.cfg` ***task instantiation (above) and execution (below) for v2***



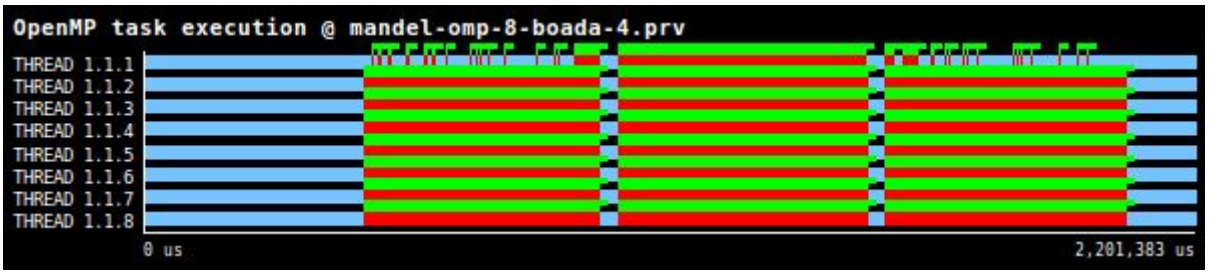


Figure 15. `OMP_tasks.cfg` ***task instantiation (above) and execution (below) for v3***

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,132	640,000
THREAD 1.1.2	83,382	-
THREAD 1.1.3	71,292	-
THREAD 1.1.4	73,907	-
THREAD 1.1.5	84,989	-
THREAD 1.1.6	84,849	-
THREAD 1.1.7	82,475	-
THREAD 1.1.8	70,974	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,132	640,000
Minimum	70,974	640,000
StDev	6,388.65	0
Avg/Max	0.91	1

Figure 16. Configuration file `OMP_task_profile.cfg` ***for v3***

In v4 of our point decomposition strategy, we are asked to modify v2 and remove the `taskwait` we introduced then. It is not necessary at all to have this `taskwait` in our code like we did in v2, as all tasks are completely independent of each other and thus there is no need to wait for all the tasks in a row to continue the execution.

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
   for (col = 0; col < width; ++col) {
       #pragma omp task firstprivate(row, col)
       {
           //code ...
       }
   }
}
```

Code: `mandel-omp-point-v4.c`

The scalability of v4 is again similar to that of v2 and v3. We observe, in Figure 17, a reduction in execution time, and increase in speed-up, up to 4 threads. After that the curve flattens off and the improvement is marginal and far from what we would expect from a

highly parallelizable computation such as this one. The number of tasks created/executed doesn't change, it's still 640,000 like for all other versions of point decomposition, as seen in Figure 18. The thread in charge of creating tasks sometimes stops the instantiation to execute some tasks. That occurs when the rest of the threads are busy executing tasks and there are tasks left in the pool to be executed.

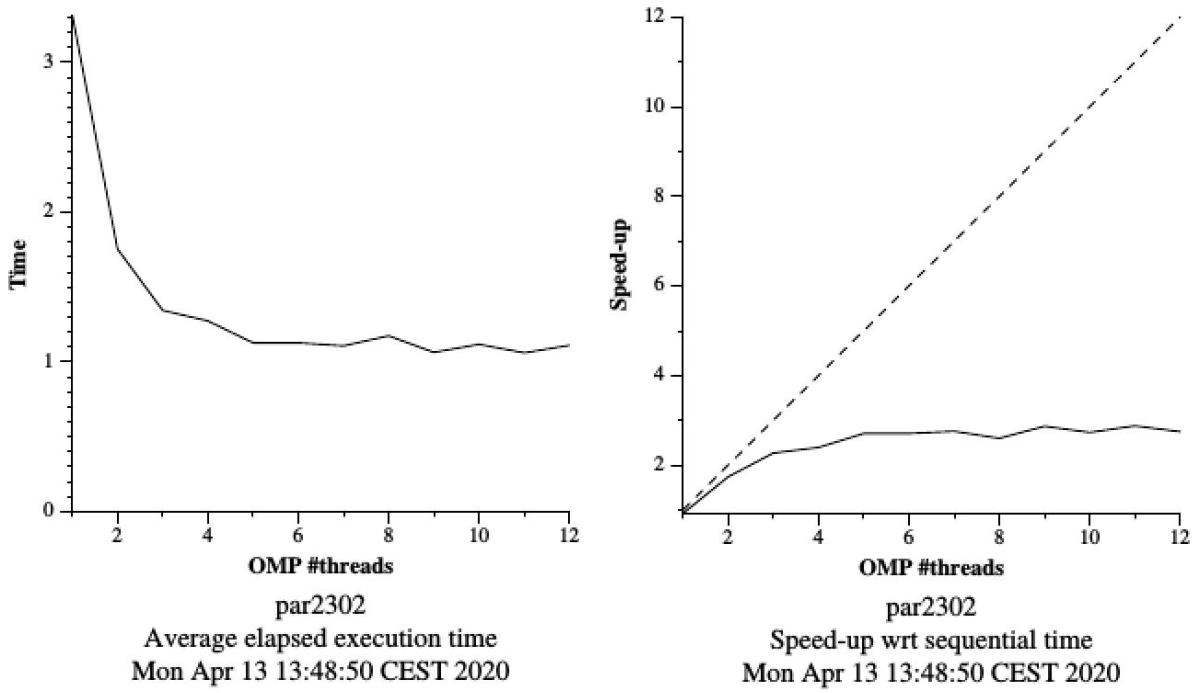


Figure 17. Strong scalability for v4

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	112,159	640,000
THREAD 1.1.2	72,961	-
THREAD 1.1.3	79,590	-
THREAD 1.1.4	78,330	-
THREAD 1.1.5	72,151	-
THREAD 1.1.6	72,208	-
THREAD 1.1.7	78,459	-
THREAD 1.1.8	74,142	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	112,159	640,000
Minimum	72,151	640,000
StDev	12,478.58	0
Avg/Max	0.71	1

Figure 18. Configuration file `OMP_task_profile.cfg` ***for v4***

We attempt to use a new approach with the `taskloop` clause, which allows us to control the granularity of each task. To implement it, we change v4 so it looks like the code below. We set num_tasks to 800 to emulate the behaviour of v2 and to compare which of the two implementations performs better. The graphical version does produce the correct result.

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
   #pragma omp taskloop firstprivate(row) num_tasks(800)
   for (col = 0; col < width; ++col) {
       //code...
   }
}
```

Code: `mandel-omp-point-v5.c`

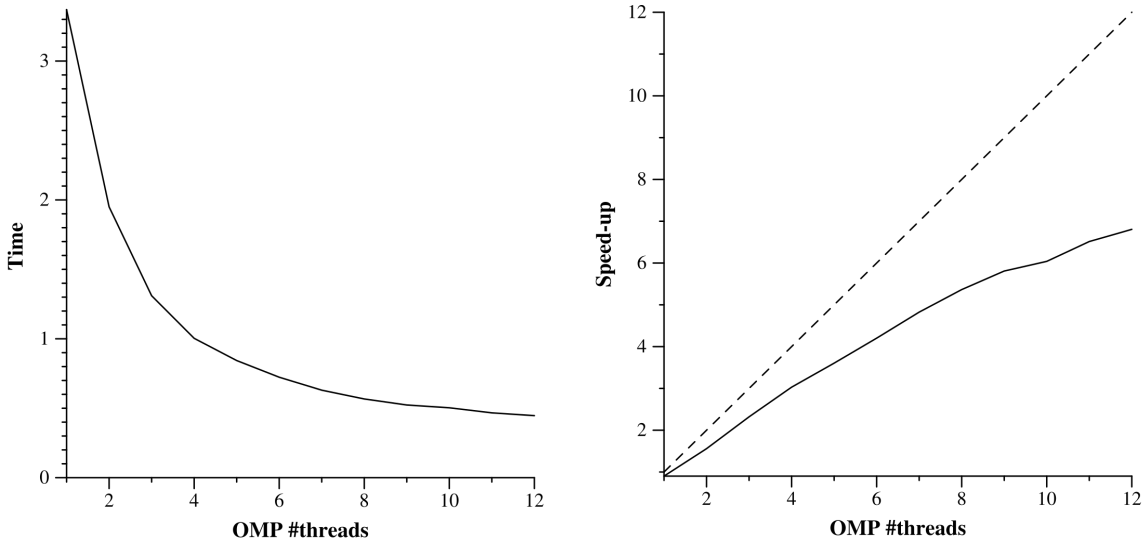
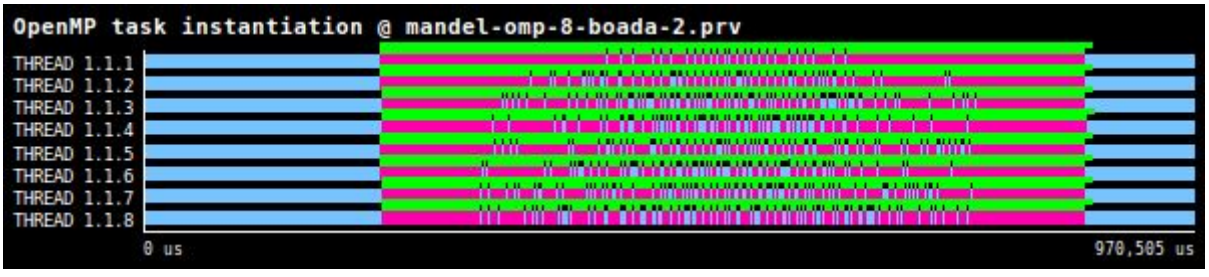


Figure 19. Strong scalability for v5

As we observe in Figure 19, the scalability of v5 is significantly better than that of v2. We see a continuous improvement in execution time and speed-up as we increase the number of threads used. While the speed-up graph shows great improvement, it still has margin for improvement as it is not close to the 45º line. In fact, the execution time when using 8 threads and setting the maximum iterations per point to 10,000 improves from 1.27 seconds for v2 to 0.56 seconds with v5.



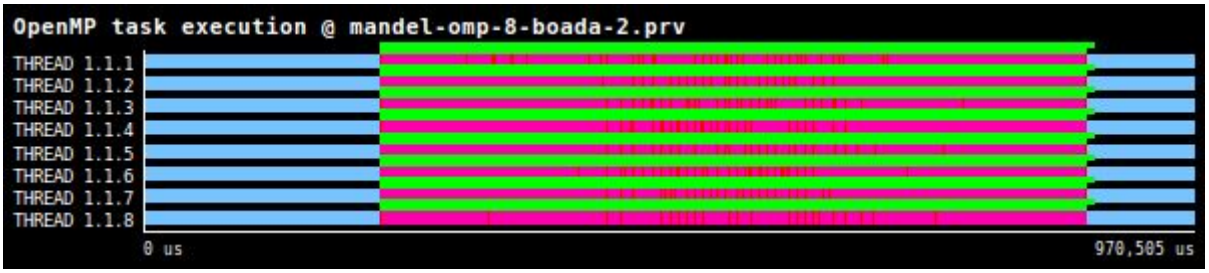


Figure 20. `OMP_tasks.cfg` ***task instantiation (above) and execution (below) for v5***

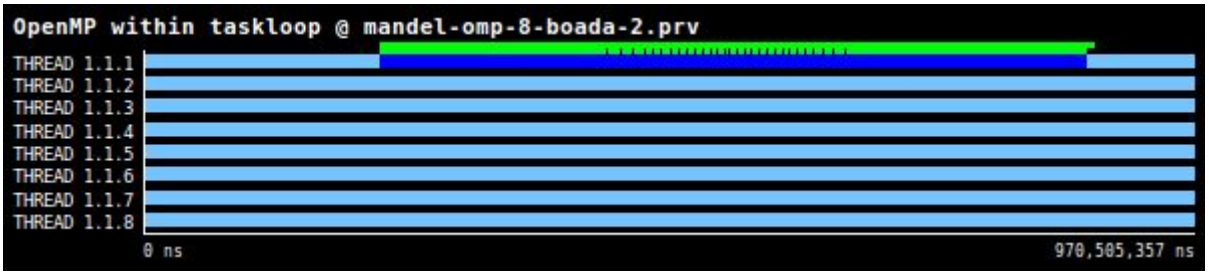


Figure 21. `OMP_taskloop.cfg` ***configuration file for v5***

The main difference with v2, as we see in Figure 20, is that all threads execute tasks the whole time. Hence, as we don't have one thread that barely executes any tasks as we do in Figure 14 for v2, we are able to highly reduce the execution time,

Finally, we have `mandel-omp-point-v6.c`, which builds upon v5 and adds the `nogroup` clause in order to eliminate the implicit task barrier contained in `taskloop`, as we know there are no dependencies between tasks and we are able to remove it harmlessly. This way we avoid having to wait for each row to finish and we can improve even further the execution time of the Mandelbrot set computation.

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
   #pragma omp taskloop firstprivate(row) num_tasks(800)nogroup
   for (col = 0; col < width; ++col) {
       //code...
   }
}
```

Code: `mandel-omp-point-v6.c`

num_tasks	**Execution 1**	**Execution 2**	**Execution 3**	**Average**
800	0.490682s	0.503447s	0.492471s	0.495533s
400	0.469296s	0.478085s	0.476505s	0.474629s

200	0.465352s	0.469155s	0.464876s	0.466461s
100	0.461096s	0.467571s	0.475636s	0.468101s
50	0.474432s	0.478169s	0.481712s	0.478104s
25	0.465461s	0.468073s	0.461681s	0.465072s
10	0.461310s	0.475764s	0.457696s	0.464923s
5	0.455762s	0.452599s	0.475399s	0.461253s
2	0.452436s	0.454445s	0.456381s	0.454421s
1	0.445127s	0.459213s	0.455939s	0.453426s

Table 3. Execution times of v6 for different task granularities

Table 3 shows us the performance of v6 using different task granularities and executed with 8 threads. As we see, we are time execution improves when the number of tasks gets close to one. That outcome is explained by the fact that the higher the number of tasks ew create, the more overhead we incur in their creation. Furthermore, it's interesting to note that when the number of tasks is set to 1, it's equivalent to row decomposition as we are only creating one task for the innermost loop.

`mandel` (non-graphical)	**1 thread**	**8 threads**
`mandel-seq.c`	3.037388s	-
`mandel-omp-v0.c`	3.321079s	1.217835s
`mandel-omp-v1.c`	3.310772s	1.232019s
`mandel-omp-v2.c`	3.317361s	1.273370s
`mandel-omp-v3.c`	3.312031s	1.313591s
`mandel-omp-v4.c`	3.313097s	1.193969s
`mandel-omp-v5.c`	3.376702s	0.561429s
`mandel-omp-v6.c -num_tasks(1)`	3.260485s	0.453426s

Table 4. Execution times for all versions of point decomposition

Observing Table 4 above, we can see a clear comparison of the performance of all 6 versions we have for point decomposition. Clearly, the better performing is v6, as when we use 8 threads we see it achieving the lowest execution time and hence the most parallelism.

Row decomposition in OpenMP

Based on our experience in the previous section, we have decided to implement row decomposition in a similar manner we did for v6 in point decomposition. That is using the `taskloop` construct with the `nogroup` clause, but this time outside the outermost loop and without making `row firstprivate`. Below we can see what our code for row decomposition looks like.

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop num_tasks(800) nogroup
for (row = 0; row < height; ++row) {
      for (col = 0; col < width; ++col) {
          //code...
      }
}
```

Code: `mandel-omp-row.c`

num_tasks	**Execution 1**	**Execution 2**	**Execution 3**	**Average**
800	0.449584s	0.454332s	0.466414s	0.456777s
400	0.467659s	0.461089s	0.456150s	0.461633s
200	0.471000s	0.489332s	0.471703s	0.477345s
100	0.509173s	0.499672s	0.503529s	0.504125s

Table 5. Execution times of row decomposition for different task granularities

In Table 5, we can observe the evolution of execution times as we vary `num_tasks` of the `taskloop` construct when executing with 8 threads. A clear trend can be noted where execution time increases as we reduce the number of tasks to be created. Hence, we conclude that using this approach we obtain the best results when setting num_tasks to 800, which is equivalent to the best result we got for point decomposition, as we are creating a task for each row in our computation.

Optional: for–based parallelization

We also implemented another version of row decomposition, this time using the `pragma omp for` construct. The code below shows what it looks like.

```
#pragma omp parallel for schedule(...)
for (row = 0; row < height; ++row) {
      for (col = 0; col < width; ++col) {
          //code...
      }
}
```

Code: `mandel-omp-for.c`

The following table, shows the different execution times for the three scheduling options available: `static`, `dynamic` and `guided`. All execution were done using 8 threads.

schedule	**Execution 1**	**Execution 2**	**Execution 3**	**Average**
static	1.296615s	1.296964s	1.290148s	1.294576s
dynamic	0.419382s	0.432531s	0.419067s	0.423660s
guided	0.473230s	0.477103s	0.470046s	0.473460s

Table 6. Execution times of for-based parallelization for different schedule options

The results manifest that the best scheduling clause for this Mandelbrot set calculation is `dynamic`, as it achieves a significantly better performance than `guided` and `static` do. `Guided` schedule produced the second best result and `static` came in third with the worst performance, approximately 3 times worse than `dynamic` and `guided` did. The most probable cause for this are load imbalance problems. Therefore, following this method we find `dynamic` to provide the best result.

Performance evaluation and Conclusions

After trying different approaches to this computation of the Mandelbrot set, we obtained the best version for each of them. For point decomposition, the best result came from v6 using `num_tasks(1)` and in the case of row decomposition using `num_tasks(800)` produced the highest performance and using for-based parallelisation with `schedule(dynamic)` did as well. Below in Table 7 we can see a comparison of their execution times with 1 and 8 threads.

`mandel` (non-graphical)	**1 thread**	**8 threads**
Point decomposition `mandel-omp-point-v6.c - num_tasks(1)`	3.260485s	0.453426s
`Row decomposition mandel-omp-row.c - num_tasks(800)`	3.259816s	0.456777s
For-based row decomposition `mandel-omp-for.c - schedule(dynamic)`	3.041804s	0.423660s

Table 7. Execution times of the best performing versions for each approach

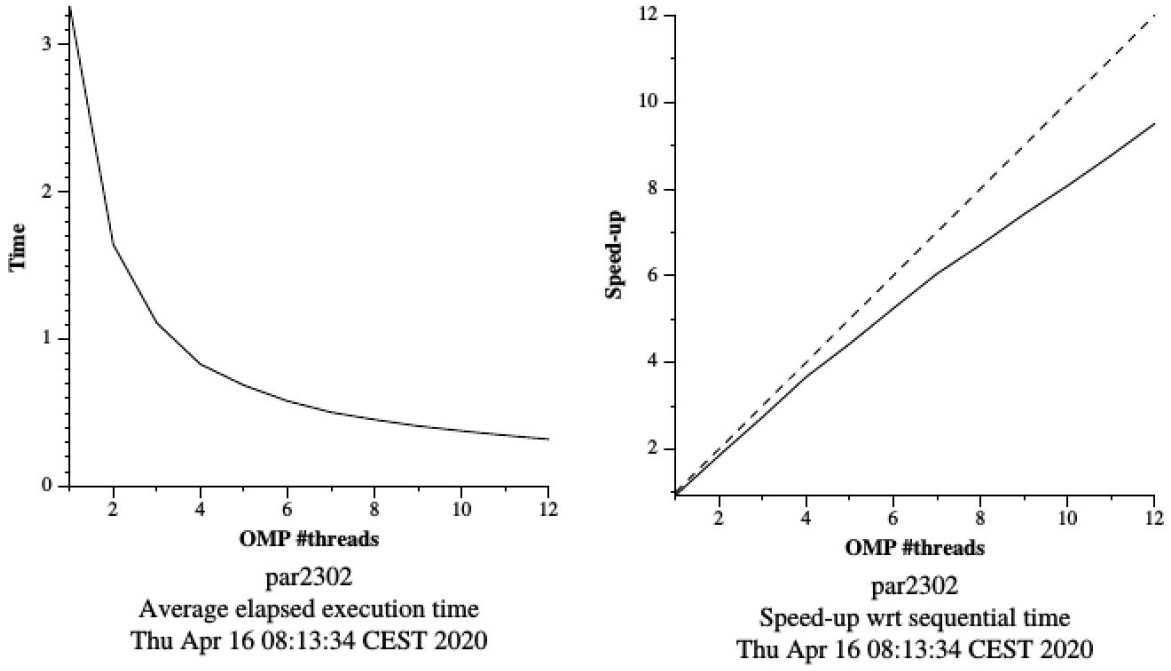


Figure 21. Strong scalability for point decomposition v6 with `num_tasks(1)`

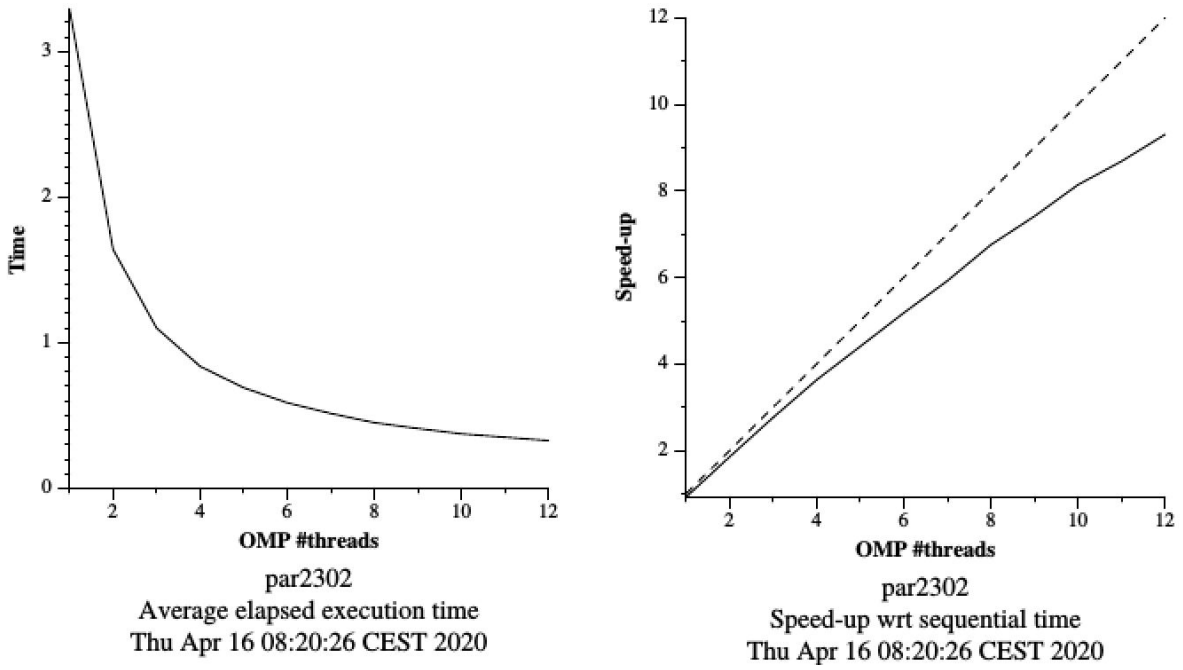


Figure 22. Strong scalability for row decomposition with `num_tasks(800)`

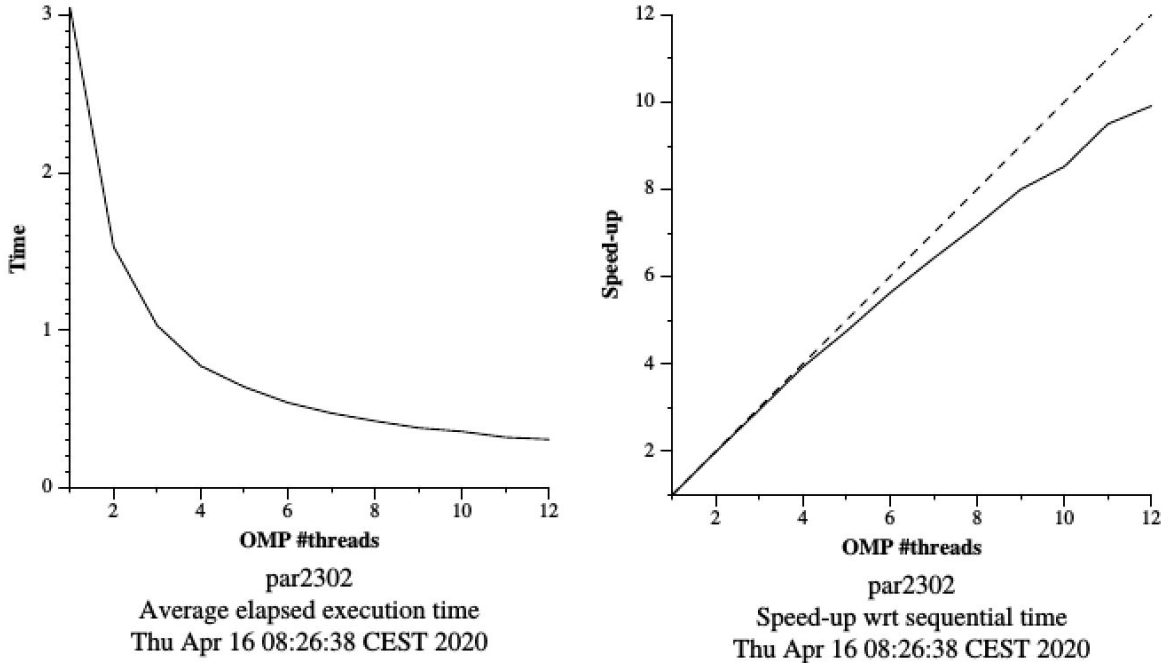


Figure 23. Strong scalability for for-based row decomposition with `schedule(dynamic)`

The plots for their scalability, which can be found below, specifically the speed-up one together with Table 7, allow us to observe that point and row decomposition yield very similar results, and that only if we used for-based parallelisation are we able to slightly improve row decomposition. In fact, we can observe that the speed-up plots in Figures 21 and 22, point v6 and row, looks almost identical.

This shouldn't come as a surprise, as in fact both version are the same in practice. In v6 using `num_tasks(1)` just before the innermost loop we are actually creating one task per row, even though we are calling it point decomposition, and in row decomposition with `num_tasks(800)`, we are also instantiating one task per row with the `taskloop` construct. Therefore, even though the code is different, we are doing exactly the same in both and their similar performance doesn't perplex us. It is important to note, that when using a higher number of tasks in point decomposition or a lower one for row decomposition, we obtained worse results. Hence, we can conclude that the best performance is obtained when issuing one single task per row, not more not less. The reason behind this being that it is probably the approach that gives the best trade-off between parallelisation and overhead due to task creation and synchronisation.

The final version and the one that gives the best result, is row decomposition using the `for` construct with the `schedule(static)` clause. As we see in Table 7, it's execution time is marginally better than that of the other two approaches and looking at its speed-up graph, Figure 23, we do see a slight improvement over the others as well. We note that up to 4 threads, the plot follows the 45^{o} line almost perfectly and after that it stays closer to it than the rest.

For all the reasons above, we conclude that a row decomposition strategy is the better method for the Mandelbrot calculation. Moreover, using `dynamic` scheduling with the `for` work-sharing construct will improve further the performance of row decomposition and provide even better scalability, thus exploiting the parallelisation of this problem even more.