Parallelism

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Group: par2206
1/05/2020
Course 2019-2020
2nd semester

Sergi Doce
Ignasi Sant

Table of contents

Introduction **3**

Analysis with Tareador **3**

Leaf Strategy 3

Tree Strategy 7

Number of tasks table 12

Parallelisation and performance analysis with tasks **12**

Parallelisation and performance analysis with dependent tasks **22**

Conclusions **26**

Introduction

When we are working with computer programs one of the important aspects that help you realize when a program is good or not is efficiency. When we use sorting algorithms this aspect is critical, especially when the size of the problem gets bigger.

In this project we will be comparing different task decompositions methods in sorting algorithms like MergeSort. We are going to use different tools like Tareador and Paraver to know exactly what is happening. The two strategies we will use are tree and leaf.

The main difference between both strategies is **who**, **where** and **when** tasks are created. While in the leaf strategy all tasks are created in the last level of recursion of a method, in the tree strategy tasks are created everytime we call the method. We will explain all of this in the following sections.

Analysis with Tareador

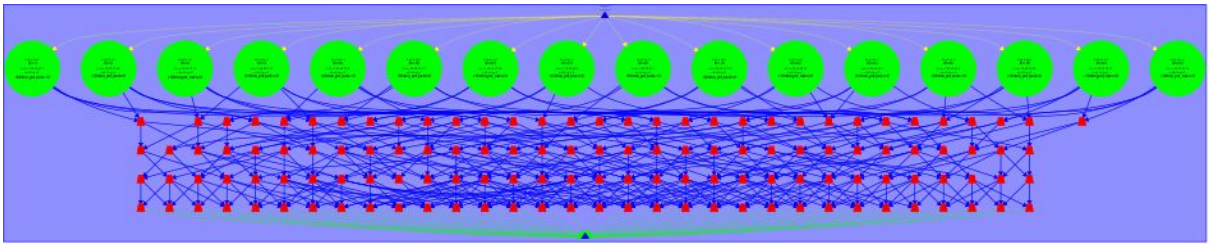
In this first part we will write two different codes: **multisort-tareador-leaf.c** which follows a **leaf strategy** of task generation, and **multisort-tareador-tree.c** which follows the **tree strategy**. Then, we just used Tareador API to get all the information we need to complete this part.

Leaf Strategy

First we will analyze the leaf strategy program. In the following images you can see how tasks are only created in the last level of recursion, in other words, in the leafs of task dependence graph. A task corresponds with each invocation of **merge** and **multisort** once the recursive invocations **stop.**

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                tareador_start_task("basicmerge");
                basicmerge(n, left, right, result, start, length);
                tareador_end_task("basicmerge");
        } else {
                // Recursive decomposition
                merge(n, left, right, result, start, length/2);
                merge(n, left, right, result, start + length/2, length/2);
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                multisort(n/4L, &data[0], &tmp[0]);
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
                // Base case
                tareador_start_task("basicsort");
                basicsort(n, data);
                tareador_end_task("basicsort");
        }
}
```

multisort-tareador-leaf.c code

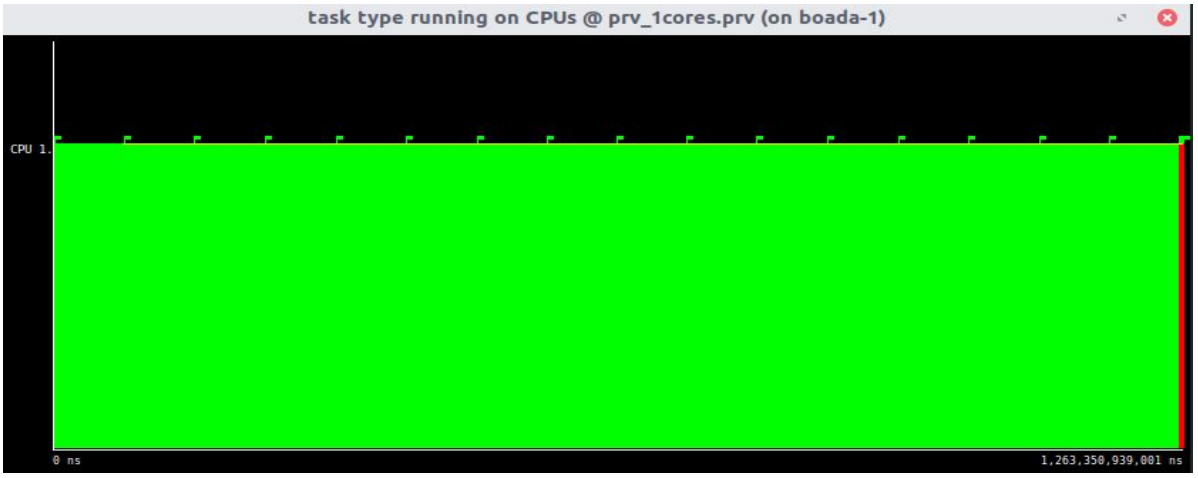


Task decomposition Graph - multisort-tareador-leaf.c

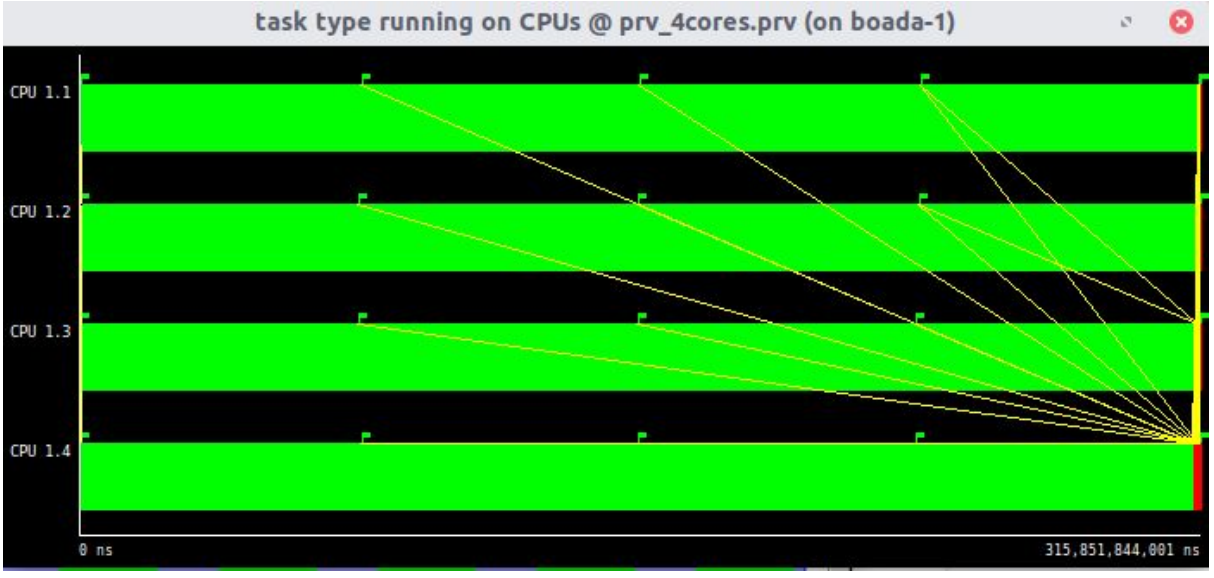
	MAIN_TAREADOR	basicsort	basicmerge
CPU 1.1	1	4	16
CPU 1.2	-	4	26
CPU 1.3	-	4	33
CPU 1.4	-	4	53
Total	1	16	128
Average	1	4	32
Maximum	1	4	53
Minimum	1	4	16
StDev	0	0	13.55
Avg/Max	1	1	0.60

Num Tasks - multisort-tareador-leaf.c

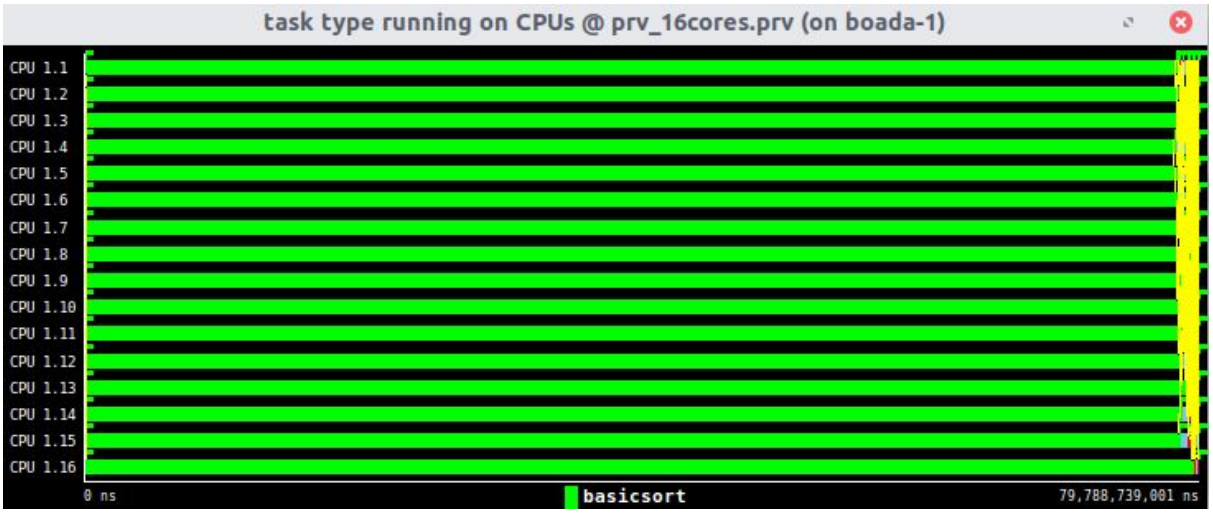
As we see we get 145 tasks from tareador which corresponds with the number of tasks appearing in the decomposition graph. We are just creating tasks in the base case of the functions **multisort** and **merge** in order to implement the leaf strategy. This can be traduced as we don't have any intermediate tasks which create other tasks as we could have in the tree strategy.



Leaf strategy with 1 thread

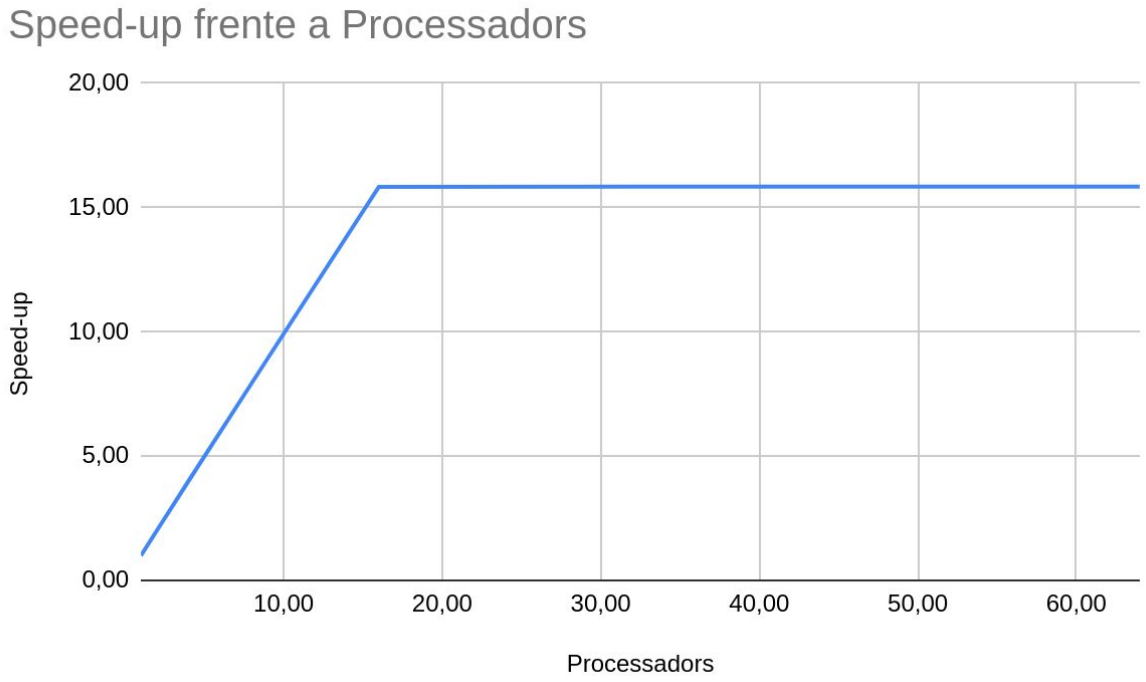


Leaf strategy with 4 threads



Leaf strategy with 16 threads

Processors	Execution Time	Speed-up
1	1263,350s	1
2	631,688s	1,99
4	315,851s	3,99
8	158,381s	7,976
16	79,788s	15,833
32	79,746s	15,84
64	79,746s	15,84



Leaf strategy - Processors vs Speed-up plot

If we take a look to the plot and the table from above we can realize that the speed-up tends to 15.84. Having the maximum speed-up we can calculate which part of the program can be parallelized. We will use **Amdahl's law**:
When P -> infinite, the expression of the speed-up becomes:

Sp = 1 / (1-α)
15.84 = 1/ (1-α) α = 0.9368

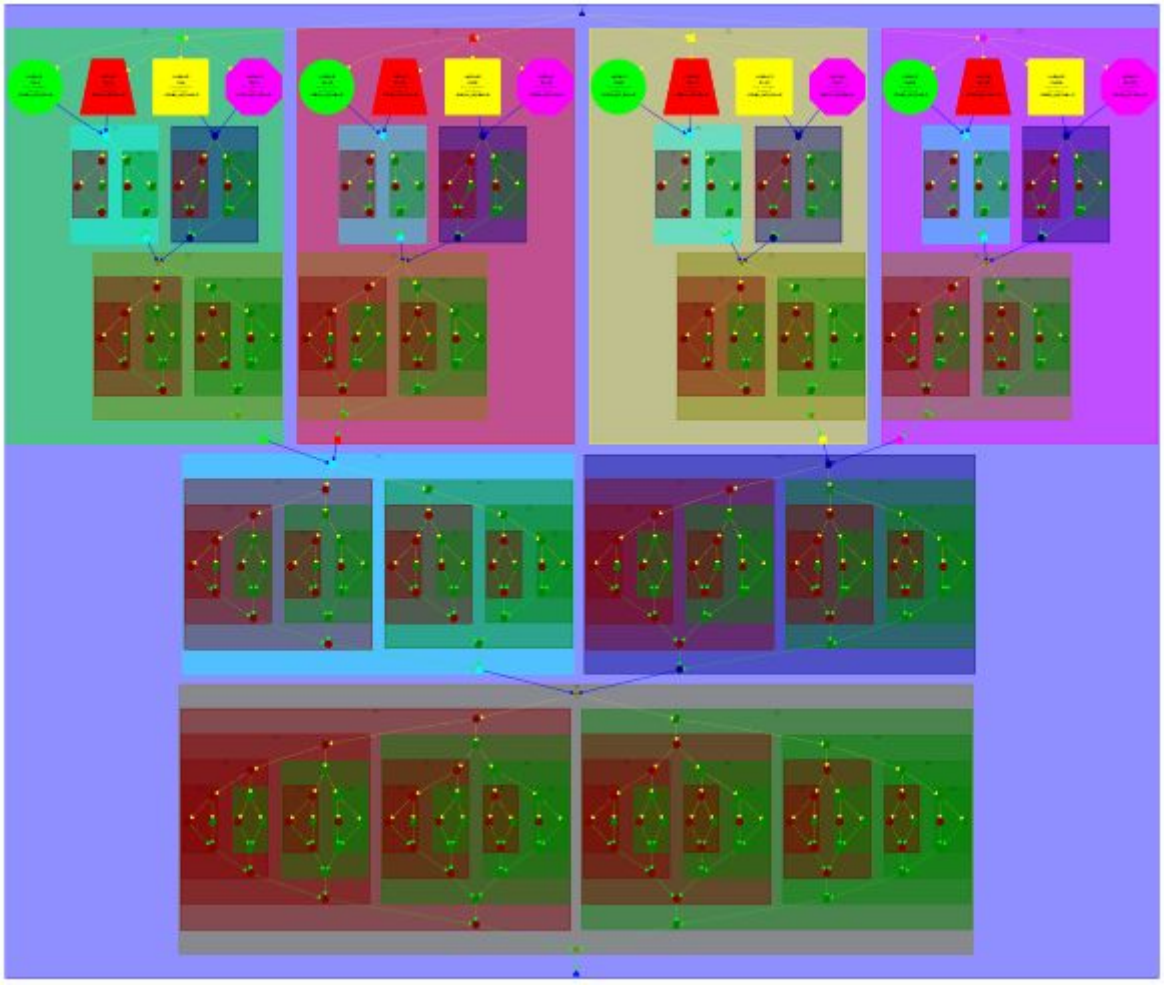
This means that 93.68% of the program is parallelizable. On the other side we have a 6.32% which is sequential and limit our speed-up from increasing linearly.

Tree Strategy

Now we will analyze tree strategy task generation. In the following images you can clearly see the difference between both strategies. Notice that a tasks corresponds with each invocation of **multisort** and **merge** methods so tasks generate other tasks.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                tareador_start_task("merge1");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("merge1");
                tareador_start_task("merge2");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("merge2");
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("multisort1");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("multisort1");

                tareador_start_task("multisort2");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("multisort2");

                tareador_start_task("multisort3");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("multisort3");

                tareador_start_task("multisort4");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("multisort4");

                tareador_start_task("merge3");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("merge3");

                tareador_start_task("merge4");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("merge4");

                tareador_start_task("merge5");
                merge(n/2L,&tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("merge5");
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

multisort-tareador-tree.c code

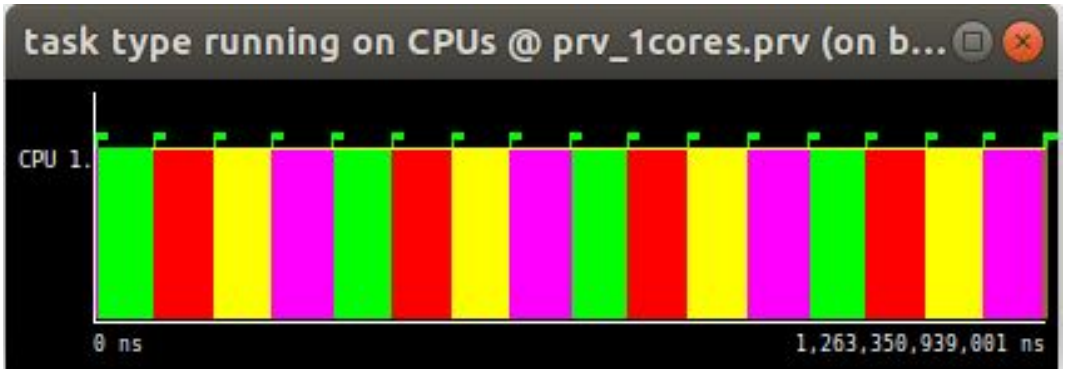


Task decomposition Graph - multisort-tareador-tree.c

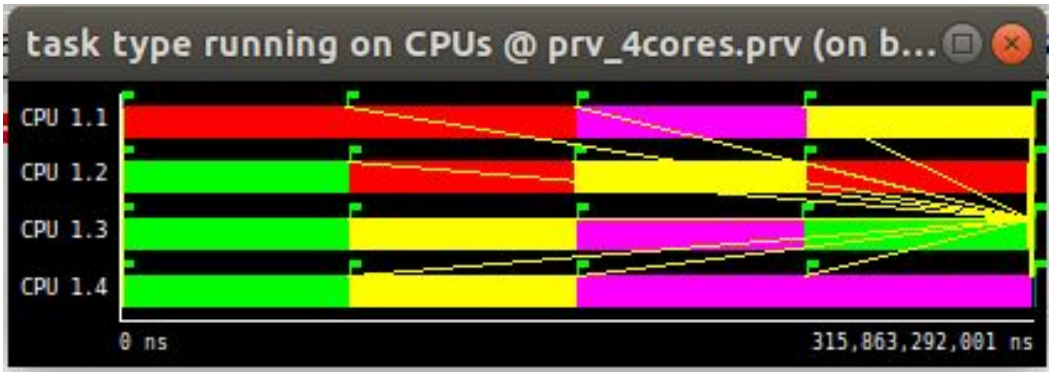
	MAIN_TAREADOR	multisort1	multisort2	multisort3	multisort4	merge3	merge1	merge2	merge4	merge5
CPU 1.1	1	5	5	5	5	5	113	113	5	5
Total	1	5	5	5	5	5	113	113	5	5
Average	1	5	5	5	5	5	113	113	5	5
Maximum	1	5	5	5	5	5	113	113	5	5
Minimum	1	5	5	5	5	5	113	113	5	5
StDev	0	0	0	0	0	0	0	0	0	0
Avg/Max	1	1	1	1	1	1	1	1	1	1

Num Tasks - multisort-tareador-tree.c

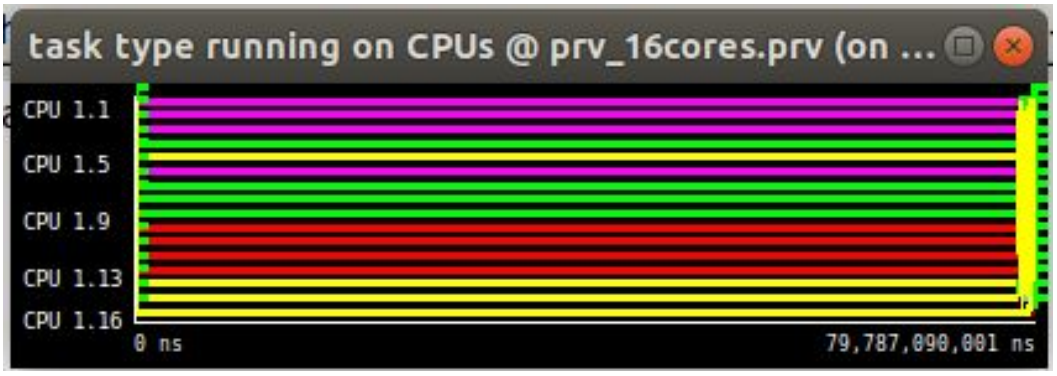
As we see we get 262 tasks from tareador. We are creating tasks in each level of the recursion of the methods **multisort** and **merge** in order to implement the tree strategy. This can be traduced as we have intermediate tasks which create other tasks .



1 processors tree strategy

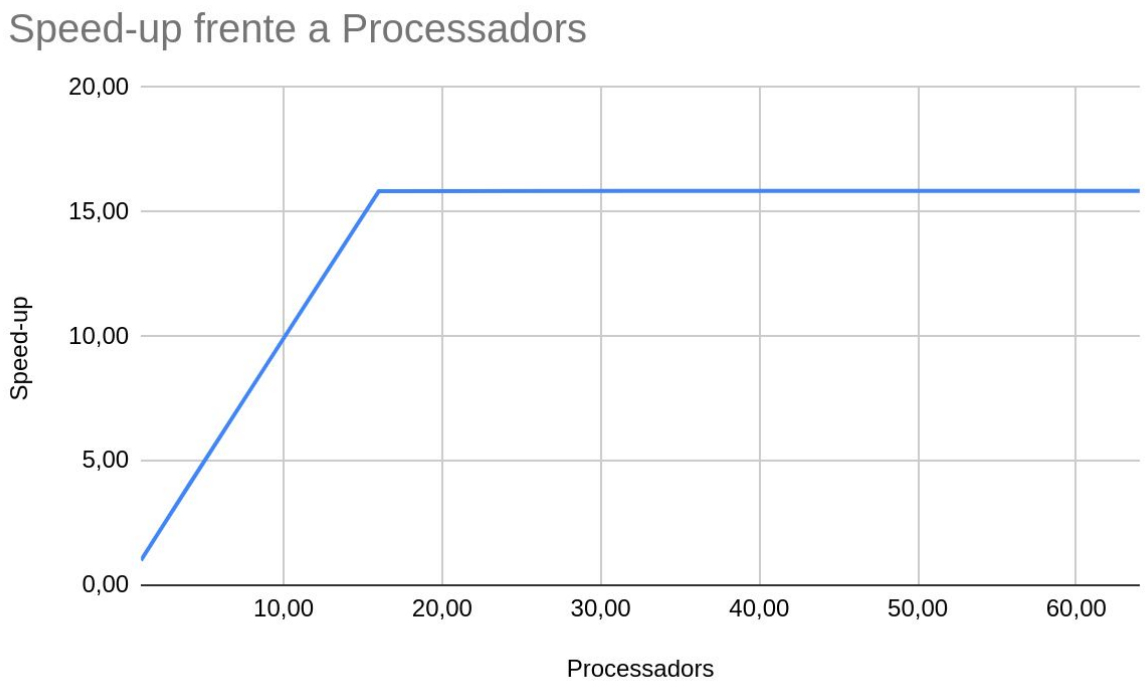


4 processors tree strategy



16 processors tree strategy

Processors	Execution Time	Speed-up
1	1263,350 seg	1
2	631,692 seg	1,999
4	315,863 seg	3.9996
8	158,824 seg	7,954
16	79,787 seg	15,834
32	79,744 seg	15,842
64	79,744 seg	15,842



Tree strategy - Processors vs Speed-up plot

If we take a look to the plot and the table from above we can realize that the speed-up tends to 15.842. Having the maximum speed-up we can calculate which part of the program can be parallelized. We will use **Amdahl's law**:
When P -> infinite, the expression of the speed-up becomes:

Sp = 1 / (1-α)
15.842 = 1/ (1-α) α = 0.9368

This means that 93.68% of the program is parallelizable. On the other side we have a 6.32% which is sequential and limit our speed-up from increasing linearly.

As we can see the speed-up obtained using this two strategies is almost the same so we can not conclude which of the strategies is better.

Number of tasks table

	Computation tasks	Internal tasks	Total tasks
Leaf strategy	144	1	145
Tree strategy	0	262	262

As in the **leaf strategy** tasks are only created with each invocation of **merge** and **multisort** once the recursive invocations **stop**, tasks always coincide with a computational task, despite, we have an exception, the “MAIN_TAREADOR” task which is not computational. For this reason, we have no intermediate task crating other tasks, because during the recursion no tasks are created.

On the other hand, in the **tree strategy** tasks corresponds with each invocation of **multisort** and **merge** methods so tasks generate other tasks. This is the reason of getting more tasks, because we have the leaf tasks plus all the intermediate tasks.

Parallelisation and performance analysis with tasks

In this section, we are going to parallelise the original code with OpenMP tasks, run it and extract some conclusions looking at the different plots and data generated. Like in the first section, we are going to analyze two different strategies of task decomposition, leaf and tree decomposition.

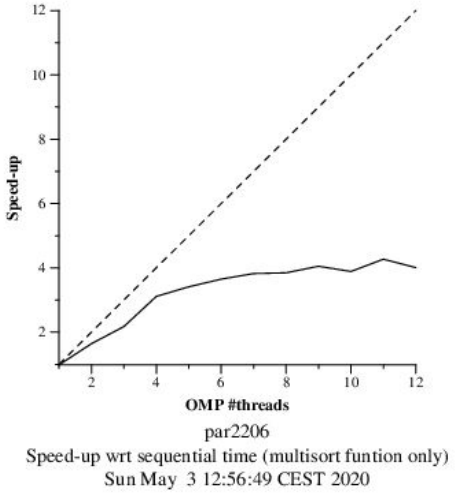
Leaf decomposition

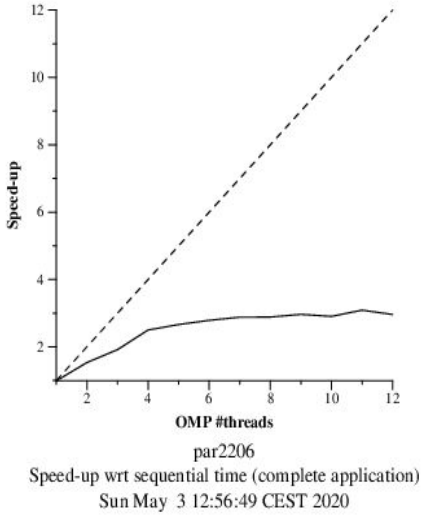
We are going to begin with the leaf version. To implement it, we have inserted OMP tasks only when the tree’s leafs are reached, that means that tasks are created only in the “base case” part of the code for both, multisort and merge, functions. However, it is needed a mechanism to synchronize the tasks to avoid data races. The mechanism we have chosen is the pragma omp taskwait construct, which waits

for all the tasks to be completed before continuing with the execution of the code. This taskwait construct has been inserted between the multisort and merge calls and before the last merge, all in the multisort function. This has been this way because for example, the merge functions, need correct values generated from the multisort functions, so they have to receive the correct data before going on.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                #pragma omp task
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                merge(n, left, right, result, start, length/2);
                merge(n, left, right, result, start + length/2, length/2);
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                multisort(n/4L, &data[0], &tmp[0]);
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                #pragma omp taskwait
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                #pragma omp taskwait
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
                // Base case
                #pragma omp task
                basicsort(n, data);
        }
}
```

The **taskwait** construct specifies a wait on the completion of child tasks of the current task.

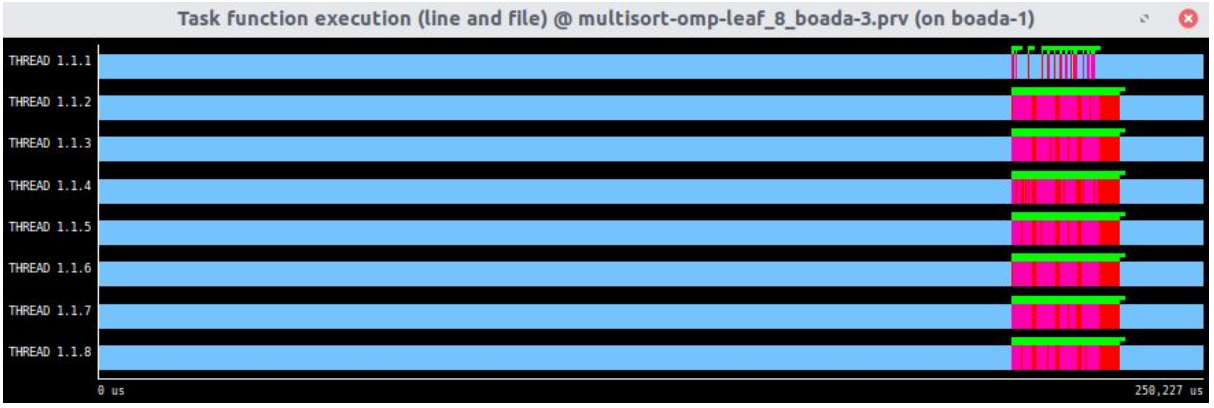




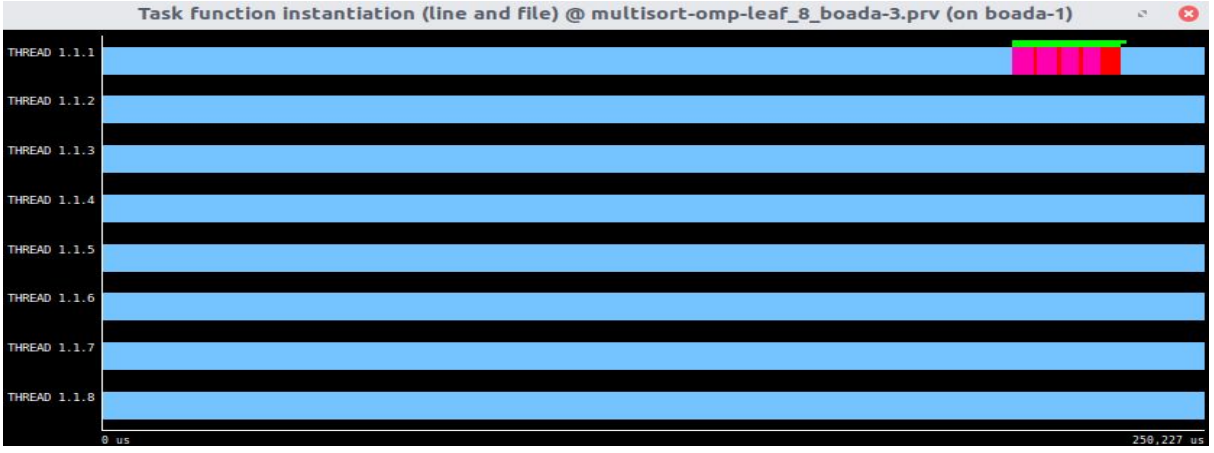
multisort-omp speed-up plots using leaf strategy

As we can see, we get non-convincing results from the speed-up plots, which show us that there is a problem concerning the parallelisation of the code. What we could think is that there is a big sequential portion of the code, which makes it not very good for parallelism because although we use more threads to execute the code, the execution time will not have an important decrement due to the big portion of the code that is still executing sequentially.

This hypothesis is confirmed by the following Paraver captures.



multisort-omp task execution timeline using leaf strategy with 8 threads.



multisort-omp task instantiation timeline using leaf strategy with 8 threads.

As we can see, the parallel work is preceded by a sequential part that is big enough to affect to our speed-up plots. This sequential part corresponds to the initialization function which we have not been asked to parallelise, so we will carry this problem through the rest of this laboratory.

Although we have a justification for the disastrous speed-up englobing all the code, we have to be aware that the speed-up of the multisort function is not affected by the sequential part of the code mentioned before, so we have to search for the problem inside the function.

The problem here is related to the strategy that we have chosen to analyze, the leaf decomposition strategy. As mentioned before, the leaf strategy only creates the tasks when we reach the base case, so the code is only parallelised when this happens. This means that the multisort function is not parallelised from the beginning, adding more sequential code to the program. This makes the code less parallelizable, and, as we have seen, this results in a disastrous speed-up again, as the code is unable to make profit from the resources it has been given.

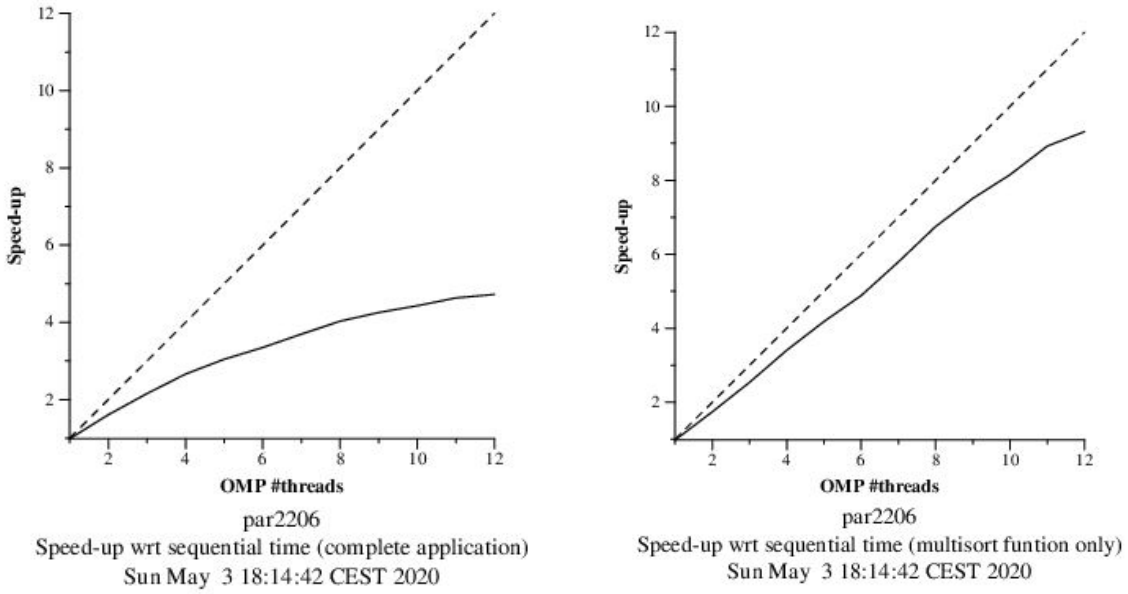
Tree decomposition

Now, we are going to analyze the tree decomposition strategy. With this strategy, every node of the tree is given a task, so the code is parallelised from the beginning. This means that to implement the strategy we have to create a task every time a recursive function is called. So, in this case, we create a task before every multisort and merge calling.

As we still have to deal with the data races problem, we still have to use the taskwait construct to ensure that the data is generated correctly. The taskwait constructs are positioned the same way as in the leaf strategy.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case

                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                #pragma omp task
                merge(n, left, right, result, start, length/2);
                #pragma omp task
                merge(n, left, right, result, start + length/2, length/2);
                #pragma omp taskwait
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task
                multisort(n/4L, &data[0], &tmp[0]);
                #pragma omp task
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                #pragma omp task
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                #pragma omp task
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                #pragma omp taskwait
                #pragma omp task
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                #pragma omp task
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

                #pragma omp taskwait
                #pragma omp task
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                #pragma omp taskwait
        } else {
                // Base case

                basicsort(n, data);
        }
}
```

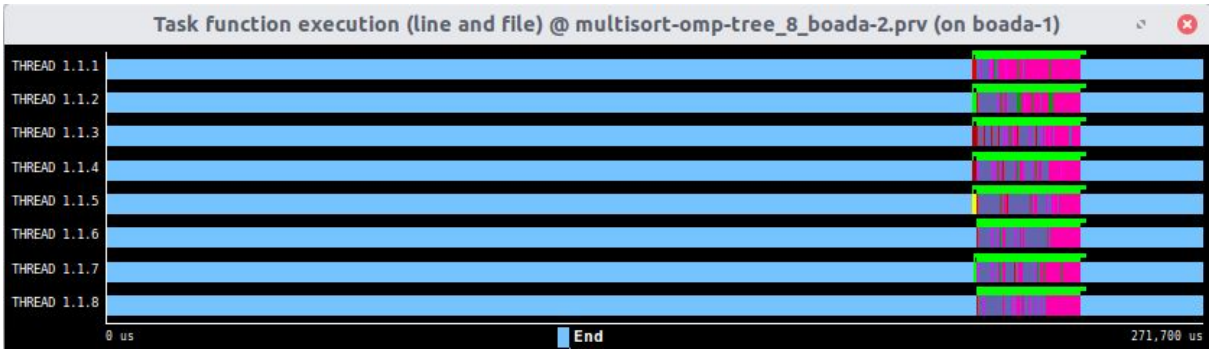
The **taskwait** construct specifies a wait on the completion of child tasks of the current task. If no depend clause is present on the taskwait construct, the current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the taskwait region complete execution.



multisort-omp speed-up plots using tree strategy

In this occasion, we can see almost no variation in the overall speed-up plot due to the big sequential part of the code because of the initialize function. However, we see a great improvement in the multisort function speed-up.

This is caused by the new tree decomposition strategy, which creates tasks from the beginning, making the multisort function more parallelizable than before and more capable to use the resources that is given.



multisort-omp task execution timeline using tree strategy with 8 threads.

multisort-omp task instantiation timeline using tree strategy with 8 threads.

Including a cut-off mechanism

In this section we are going to include a cut-off mechanism. This kind of mechanism makes the program stop generating tasks at a certain level of recursion in order to improve optimization.

To implement this mechanism, we have followed the scheme showed at the theory classes, using a new variable called depth to control the recursion level we're at.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case

                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                if(!omp_in_final()){
                        #pragma omp task final(depth >= CUTOFF)
                        merge(n, left, right, result, start, length/2, depth + 1);
                        #pragma omp task final(depth >= CUTOFF)
                        merge(n, left, right, result, start + length/2, length/2, depth + 1);
                        #pragma omp taskwait
                }
                else{
                        merge(n, left, right, result, start, length/2, depth + 1);
                        merge(n, left, right, result, start + length/2, length/2, depth + 1);
                }
        }
}
```

```
void multisort(long n, T data[n], T tmp[n], int depth) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                if(!omp_in_final()){
                        #pragma omp task final(depth >= CUTOFF)
                        multisort(n/4L, &data[0], &tmp[0], depth + 1);
                        #pragma omp task final(depth >= CUTOFF)
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth + 1);
                        #pragma omp task final(depth >= CUTOFF)
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth + 1);
                        #pragma omp task final(depth >= CUTOFF)
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth + 1);

                        #pragma omp taskwait
                        #pragma omp task final(depth >= CUTOFF)
                        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth + 1);
                        #pragma omp task final(depth >= CUTOFF)
                        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth + 1);

                        #pragma omp taskwait
                        #pragma omp task final(depth >= CUTOFF)
                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth + 1);
                        #pragma omp taskwait
                }
                else{

                        multisort(n/4L, &data[0], &tmp[0], depth + 1);
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth + 1);
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth + 1);
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth + 1);

                        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth + 1);
                        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth + 1);

                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth + 1);

                }
        } else {
                // Base case

                basicsort(n, data);
        }
}
```

When a final clause is present on a task construct and the final clause expression evaluates to true, the generated task will be a final task. All task constructs encountered during execution of a final task will generate final and included tasks.

In the following paraver captures, we can see how the mechanism works. It creates tasks until the level of recursion defined at CUTOFF(included) is reached and then the execution continues but without creating any more tasks. We have tried the mechanism with two different examples:

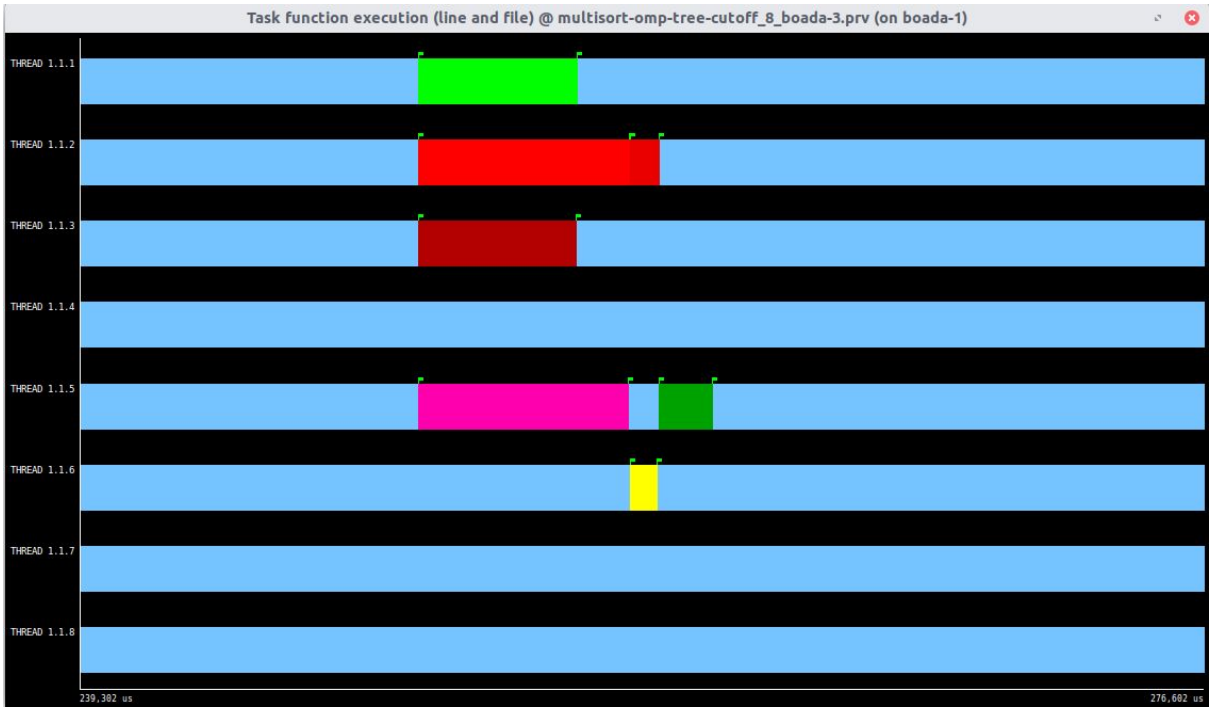
Task generation until level 0

In this example, task generation is only permitted in the first level of the recursion. That means, that in this case, we will create just 7 tasks because the execution of the function multisort creates 7 tasks.

- 4 tasks for the multisort functions

- 3 tasks for the merge functions

We can see this example at the following Paraver capture.



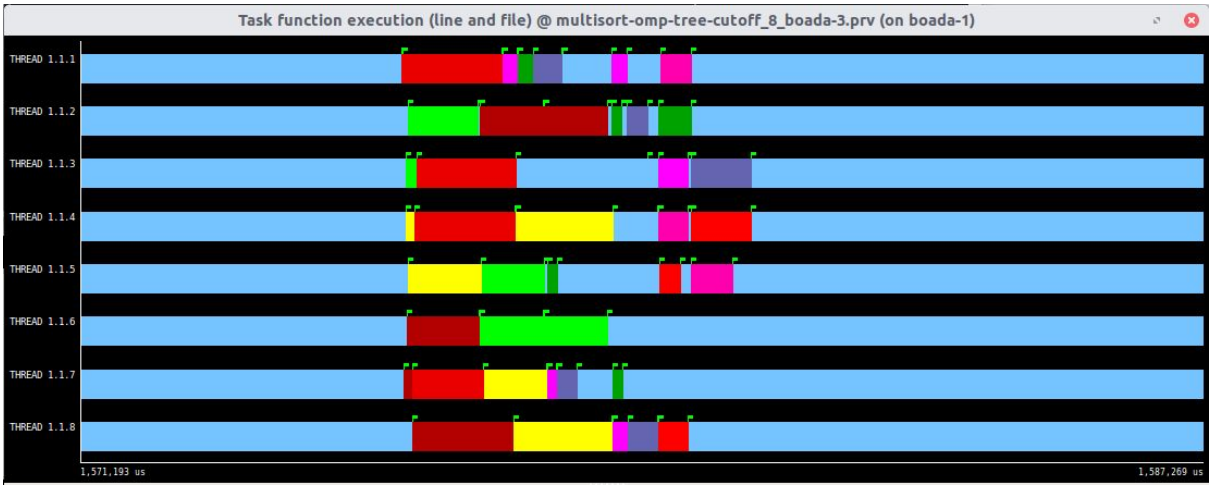
multisort-omp following tree strategy with task generation until level 0

Task generation until level 1

In this second example, task generation is only allowed until the second level of recursion. Having that in mind, we can understand why the total number of tasks created is 41. As we have said before, every multisort function generates 7 tasks. We also know that every merge task generates 2 more tasks, but in this case, the merge function will only create tasks if it is derivated from the first multisort, so we have to add 6 more tasks.

- 7 * 5 = 35
- 2 * 3 = 6

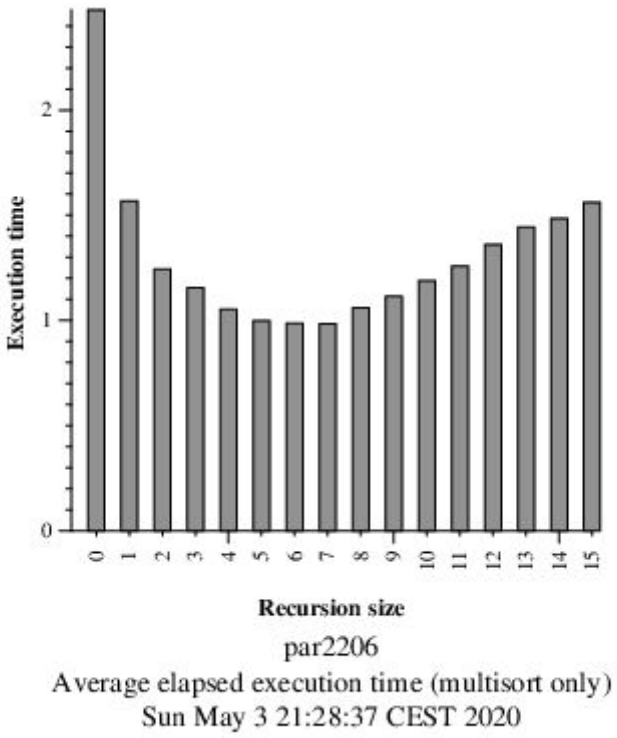
We can see this example in the following Paraver capture.



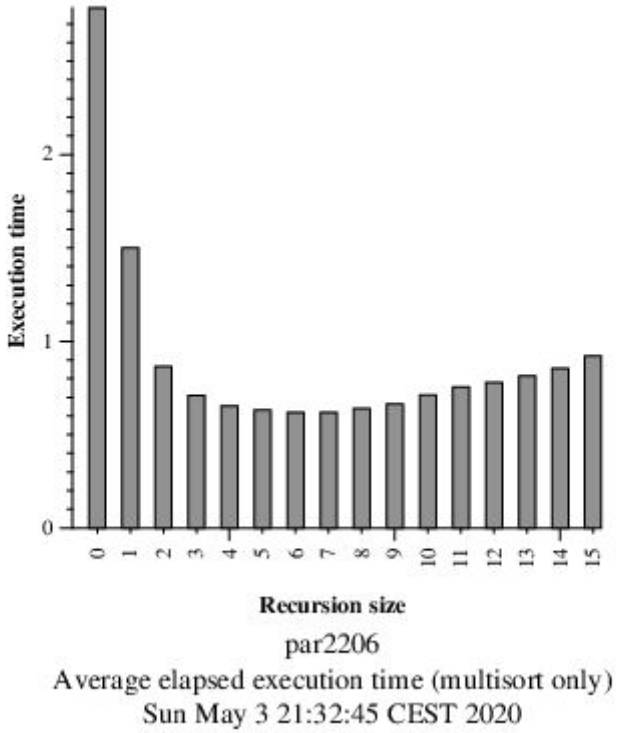
multisort-omp following tree strategy with task generation until level 1

Now, we shall take a look at the execution time plots in order to extract some conclusion about which is the best cut-off to get the best perform. We have executed two test, one executing the program with 8 threads and another one executing it with 16 threads. This has been done in order to see if the number of threads played a significant role in selecting the best cut-off level.

We have seen that no matter the number of threads, the best cut-off level is 7.



Using 8 threads



Using 16 threads

Parallelisation and performance analysis with dependent tasks

In this last session we have changed our synchronisation mechanism based on the taskwait construct for the point by point task dependencies. This should make our program more efficient because new tasks will only have to wait for the data they depend on, instead of the ending of all the tasks in the group.

In order to implement this mechanism, we have used the depend clause in all of the task creation constructs except in the tasks created in the merge function, as they don't have any data dependency.

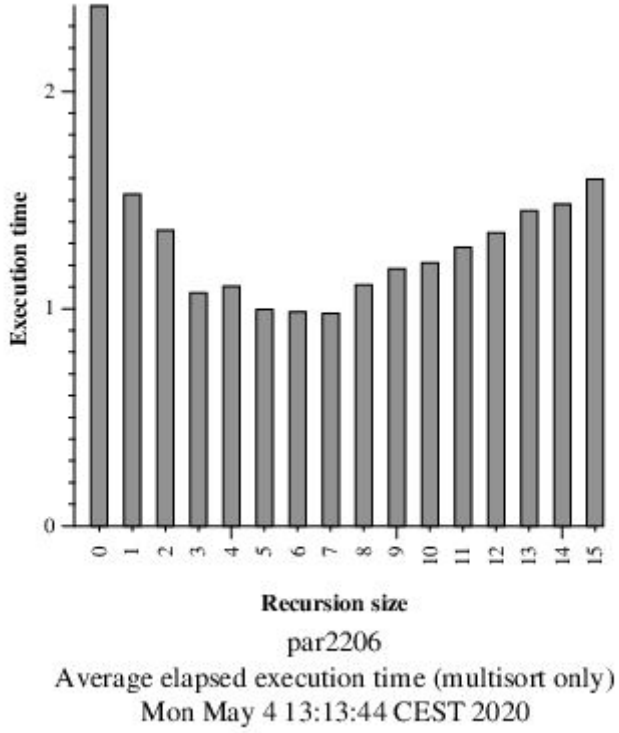
The depend clause enforces additional constraints on the task scheduling by establishing dependencies between sibling tasks. However, we still have to use the taskwait construct in order to avoid erroneous data generation. These taskwait have been positioned at the end of the task regions.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case

                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                if(!omp_in_final()){
                        #pragma omp task final(depth >= CUTOFF)
                        merge(n, left, right, result, start, length/2, depth + 1);
                        #pragma omp task final(depth >= CUTOFF)
                        merge(n, left, right, result, start + length/2, length/2, depth + 1);
                        #pragma omp taskwait
                }
                else{
                        merge(n, left, right, result, start, length/2, depth + 1);
                        merge(n, left, right, result, start + length/2, length/2, depth + 1);
                }
        }
}

void multisort(long n, T data[n], T tmp[n], int depth) {
       if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                if(!omp_in_final()){
                        #pragma omp task final(depth >= CUTOFF) depend(out:data[0])
                        multisort(n/4L, &data[0], &tmp[0], depth + 1);
                        #pragma omp task final(depth >= CUTOFF) depend(out:data[n/4L])
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth + 1);
                        #pragma omp task final(depth >= CUTOFF) depend(out:data[n/2L])
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth + 1);
                        #pragma omp task final(depth >= CUTOFF) depend(out:data[3L*n/4L])
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth + 1);


                        #pragma omp task final(depth >= CUTOFF) depend(in:data[0], data[n/4L]) depend(out:tmp[0])
                        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth + 1);
                        #pragma omp task final(depth >= CUTOFF) depend(in:data[n/2L], data[3L*n/4L]) depend(out:tmp[n/2L])
                        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth + 1);


                        #pragma omp task final(depth >= CUTOFF) depend(in:tmp[0], tmp[n/2L])
                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth + 1);
                        #pragma omp taskwait
                }
                else{

                        multisort(n/4L, &data[0], &tmp[0], depth + 1);
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth + 1);
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth + 1);
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth + 1);

                        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth + 1);
                        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth + 1);

                        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth + 1);
```

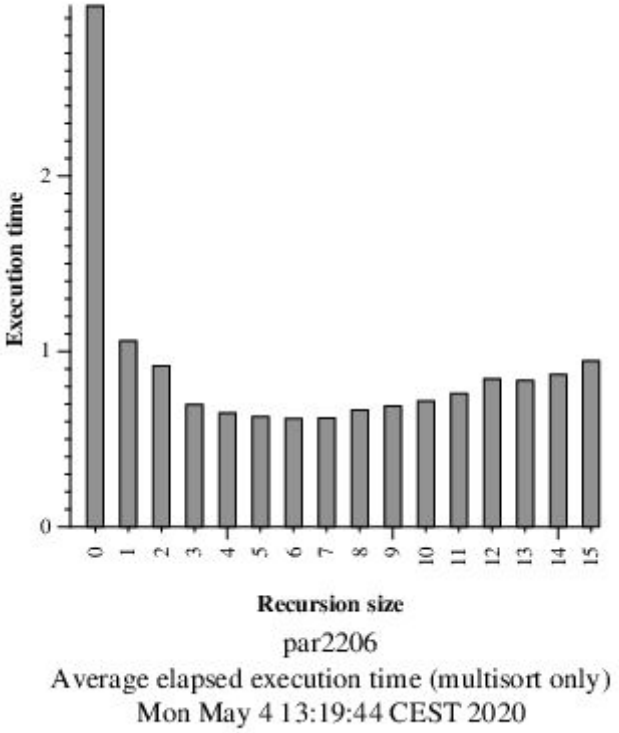
Now, we will take a look at the plots and Paraver captures obtained after executing some tests.

As we still have the cut-off mechanism, we have performed some tests to see if the depend clauses have had any effects on the most optimal recursion level that we discovered in the previous section. The results are negative. As we can see, after performing the test with different cut-off values and different number of threads, the best value is still 7.

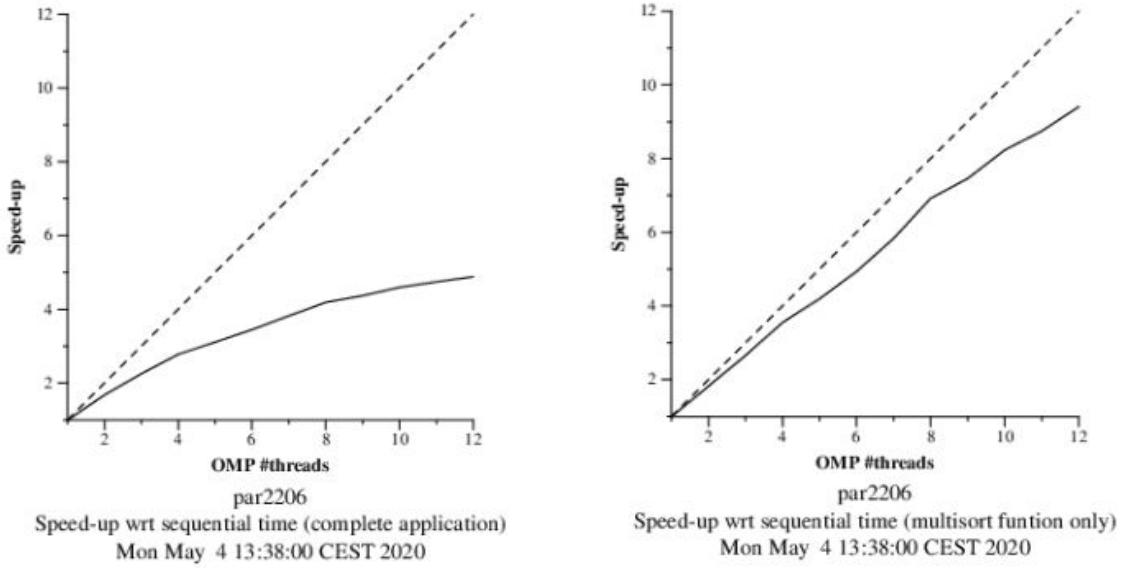
After concluding that 7 was still the best value for the cut-off, we tested the program with 8 threads to extract the speed-up plots and look if there has been any difference.



Average elapsed execution time per recursion size with 8 threads with using task dependencies



Average elapsed execution time per recursion size with 16 threads with using task dependencies



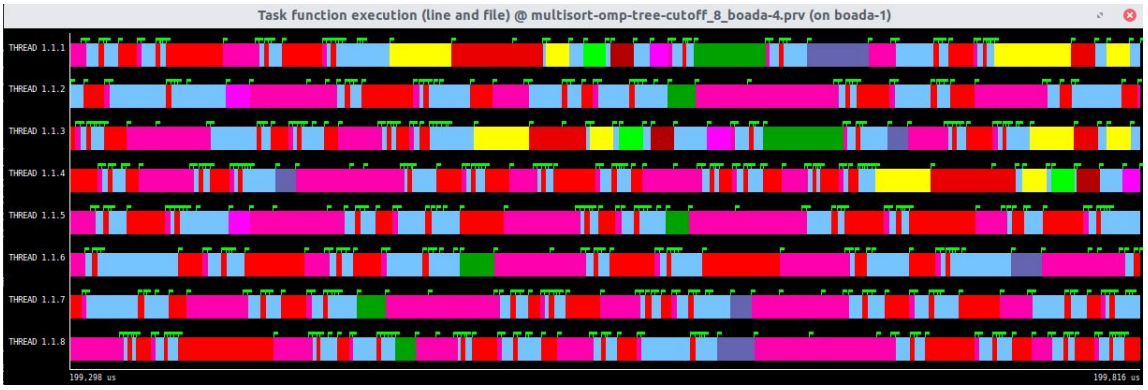
Speedup plots in tree strategy with cut off system set to 7 with task dependencies

As we can see, there is almost no difference between this speed-up plots and the original speed-up plots from the version with no task dependencies. Actually, the program performs worse in the new version as we have seen in the output of the submits.

- Execution time with no dependencies(multisort function) -> 0,024s
- Execution time with dependencies(multisort function) -> 0,025s

However, it is curious to see that the time between tasks is inferior in the dependencies version, as we can see in the following Paraver captures.

Task execution timeline with 8 threads using task dependencies



Task execution timeline with 8 threads **not** using task dependencies

Conclusions

In this laboratory assignment we have implemented two ways to parallelise the Multisort algorithm. This two ways are based in the two task decomposition strategies for recursive algorithms that we have learnt in theory class, the leaf and tree task decompositions. Apart from that, we have implemented two possible improvements for the tree strategy, a cut-off mechanism and a synchronisation mechanism based on point to point task dependencies.

Regarding on which task decomposition strategy is the best, we have no doubt that the tree strategy performs a lot better in this case, as the speed-up plots have shown us. The leaf strategy only creates tasks when the leaf is reached, on the other hand, the tree strategy creates tasks from the beginning of the multisort function, making it much better for parallelism as it uses more efficiently all the resources that is given.

Having stated that the tree task decomposition is the best, we move on talking about our conclusions about the cut-off system and the task dependencies mechanism.

After implementing the cut-off system, thanks to our submissions of the “submit-cutoff.sh” script, we can conclude that the best cut-off level is 7, improving the execution time but not the speed-up. However, we don’t have this positive results with the point to point dependencies synchronisation mechanism. Although we have an inferior synchronisation time, the total execution time is a little bit superior, making this mechanism not optimal for this program.