PAR Laboratory

Assignment Lab 4:

Divide and Conquer parallelism with OpenMP: Sorting.

Guillem Dubé Quintín par3103

Ricard Guixaró Trancho par3108

Fall 2021-2022

Date: 17/11/2022

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Index

1. Task Decomposition analysis for Mergesort

1.1. “Divide and conquer”

Execution times of the sequential version with problem size N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024:

make multisort-seq

- Initialization time in seconds: 0.863707
- Multisort execution time: 6.260682
- Check sorted data execution time: 0.015504

1.2. Task decomposition analysis with Tareador

Leaf Strategy:

To implement the Leaf Strategy, we added the necessary calls to create (*tareador_start_task()*) and end (*tareador_end_task()*) tareador tasks in each of the base cases of *basicsort* and *basicmerge* functions.

```
void basicsort(long n, T data[n]);
void basicmerge(long n, T left[n], T right[n], T result[n*2], long start,
long length);

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("Leaf-basicmerge-task");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("Leaf-basicmerge-task");
    } 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("Leaf-basicsort-task");
        basicsort(n, data);
        tareador_end_task("Leaf-basicsort-task");
    }
}
```

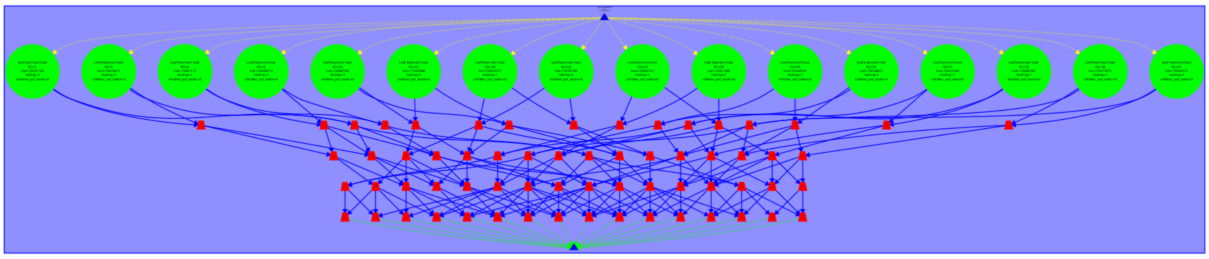


Figure 1: multisort.c decomposition using Left Strategy

sbatch ./submit-extrae.sh multisort-omp 8

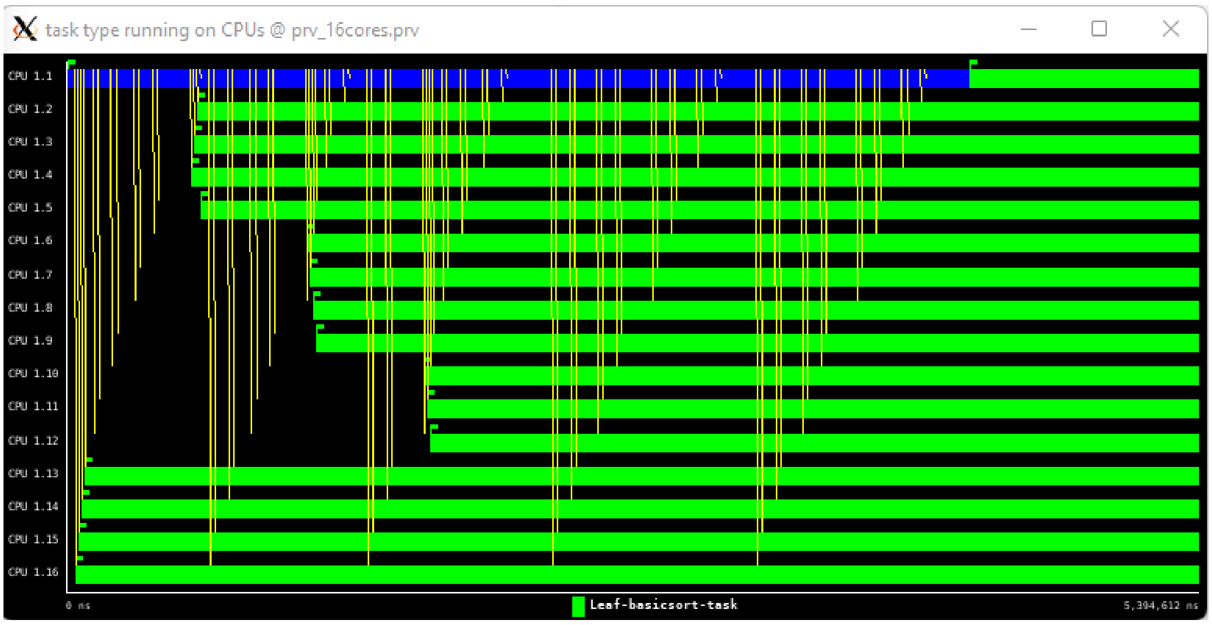
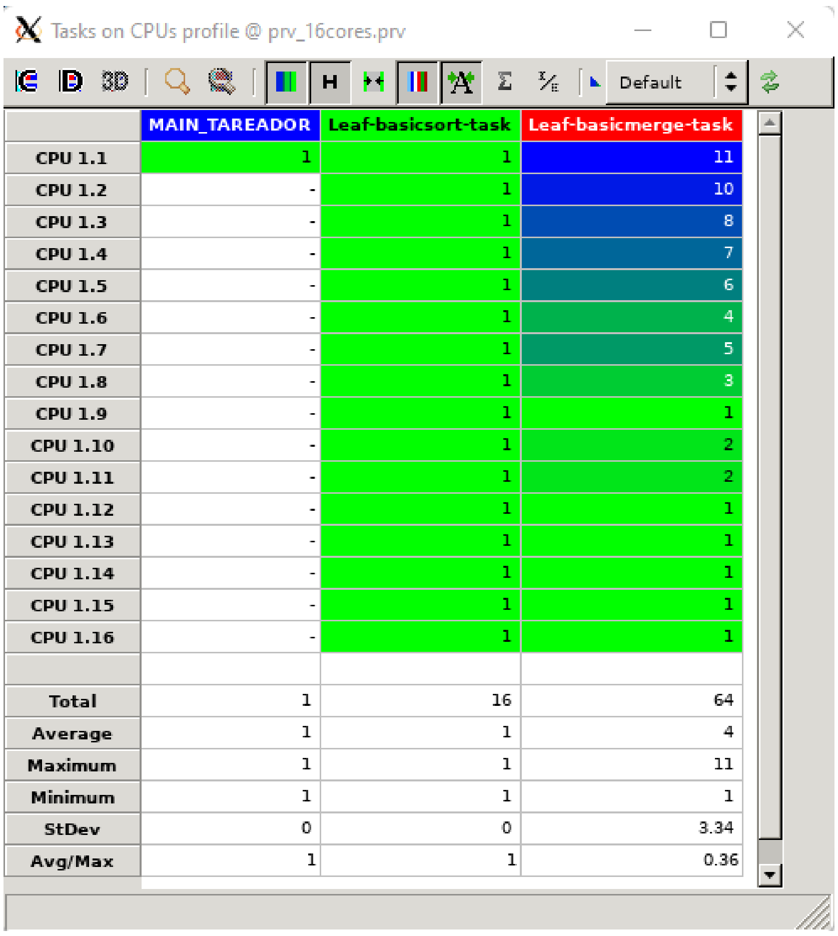


Figure 2: multisort.c execution implementing Leaf Strategy decomposition and using 16 threads


Tasks on CPUs profile @ prv_16cores.prv

	MAIN_TAREADOR	Leaf-basicsort-task	Leaf-basicmerge-task
CPU 1.1	1	1	11
CPU 1.2	-	1	10
CPU 1.3	-	1	8
CPU 1.4	-	1	7
CPU 1.5	-	1	6
CPU 1.6	-	1	4
CPU 1.7	-	1	5
CPU 1.8	-	1	3
CPU 1.9	-	1	1
CPU 1.10	-	1	2
CPU 1.11	-	1	2
CPU 1.12	-	1	1
CPU 1.13	-	1	1
CPU 1.14	-	1	1
CPU 1.15	-	1	1
CPU 1.16	-	1	1
Total	1	16	64
Average	1	1	4
Maximum	1	1	11
Minimum	1	1	1
StDev	0	0	3.34
Avg/Max	1	1	0.36

Figure 3: Tasks profile using Leaf Strategy

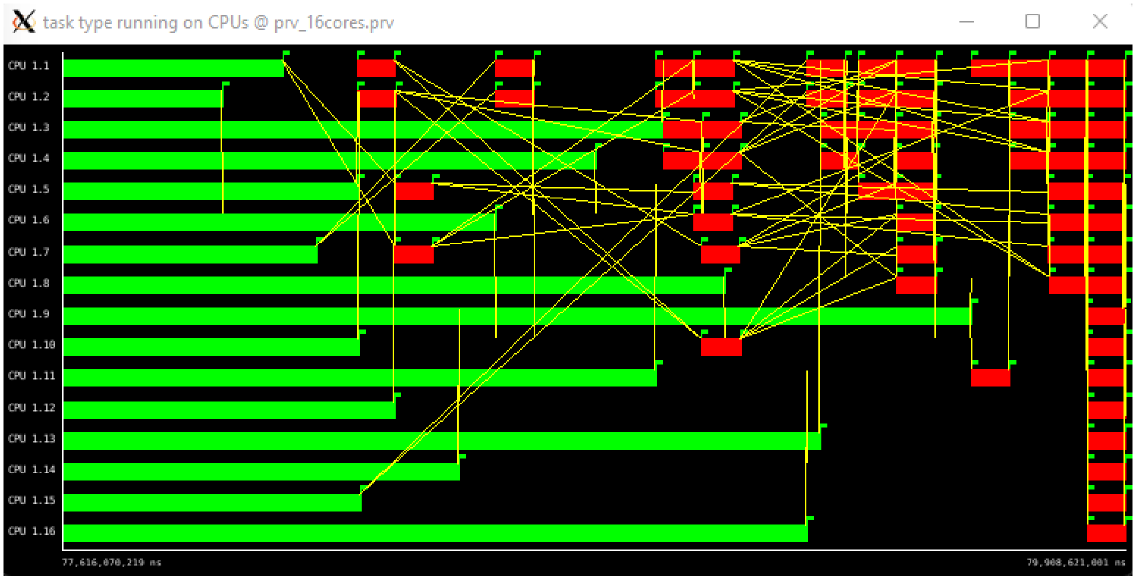


Figure 4: multisort.c execution implementing Leaf Strategy decomposition and using 16 threads

We corroborated the number of tasks that we obtained analyzing the code with the total number from the picture below.

- In the beginning of execution using leaf strategy, the main thread spends some time (insignificant compared to the basicsort time) creating all tasks dependencies.

Tree Strategy:

To implement the Tree Strategy, we added the necessary calls to create (*tareador_start_task()*) and end (*tareador_end_task()*) tareador tasks in each of the recursive calls of *multisort()* and *merge()* functions.

```
void basicsort(long n, T data[n]);
void basicmerge(long n, T left[n], T right[n], T result[n*2], long start,
long length);

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("mergemerge1");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("mergemerge1");
        tareador_start_task("mergemerge2");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("mergemerge2");
    }
}

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("merge1");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("merge1");
        tareador_start_task("merge2");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("merge2");
        tareador_start_task("merge3");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("merge3");
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

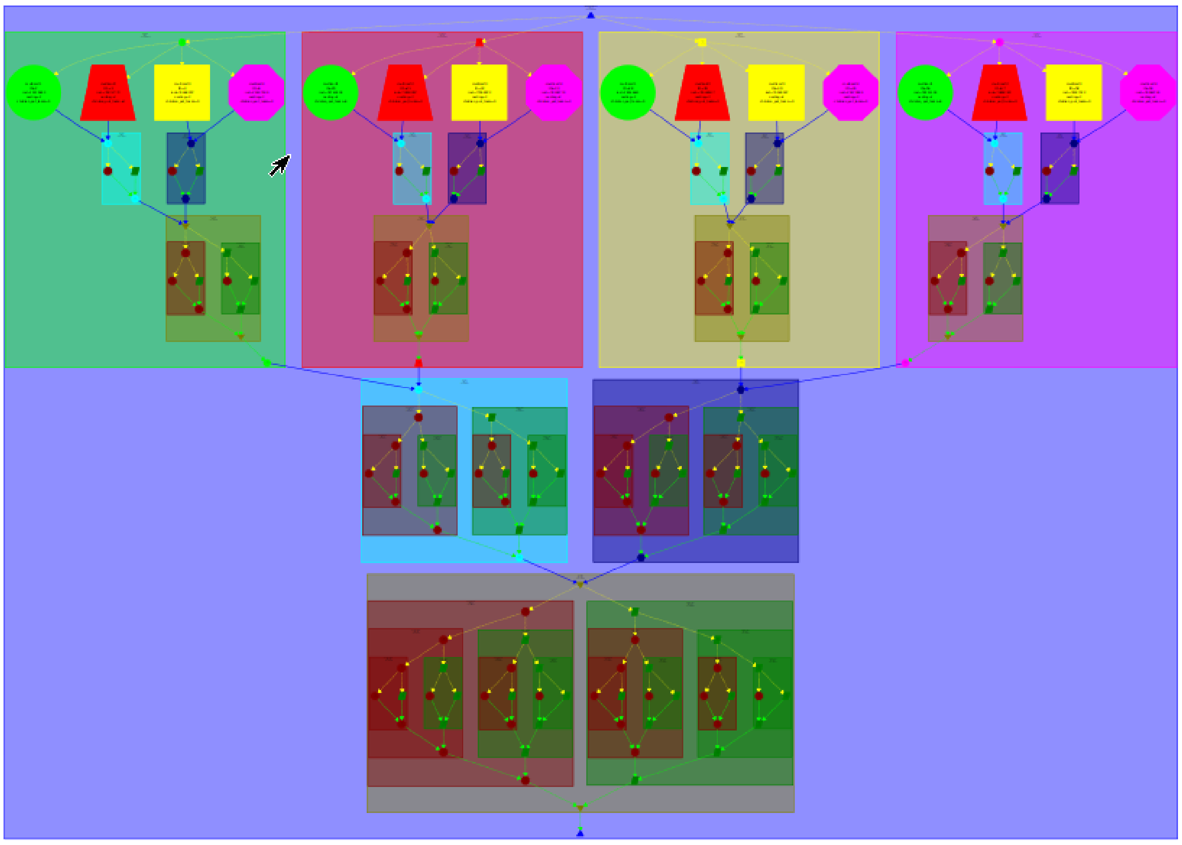


Figure 5: multisort.c decomposition using Tree Strategy

Tasks:

- *multisort1*: green circles
- *multisort2*: red trapezoids
- *multisort3*: yellow squares
- *multisort4*: pink octagons

- merge1: cian circles
- merge2: dark blue hexagon
- merge3: grey triangles
- merge4: red octagons
- merge5: dark green parallelograms

We know that the *run-tareador.sh* script uses the following options: *-n 32 -s 2048 -m 2048*, meaning that there are a total of 32 Kelements in the vector (32 * 1024 = 2^{15}), that the recursive decomposition during the sort phase will stop when 2048 * 4 (2^{13}) elements are reached and that the recursive decomposition during the merge phase will stop when 2048 * 2 (2^{12}) elements are reached.

By examining the *leaf* and *tree* task graphs, we create the following table, summing up the number of tasks depending on their function in the program:

- Tasks doing computation are the ones executing *basicsort()* and *basicmerge()* (the ones executing the leaves of the recursion tree, the ones located inside the base cases in case of *leaf*).

- Internal tasks are the ones that create new tasks (*multisort()* and *merge()* invocations).

What we can clearly see is that the code divides itself in 4 multisort() functions, each multisort() function will execute the 3 merge() functions, the order will be, the first two multisort() functions execute the first merge() function, the next two multisort() will execute the next merge() and then the 3rd merge will be executed at the end, using the results obtained in the merge() functions used before, as we can clearly see in the Figure 2.

We will comment also about the tasks graphs that are generated, including a table with the number of tasks that are generated at each recursion level for each task decomposition strategy.

Recursion level	Leaf Strategy		Tree Strategy	
	basicsort	*basicmerge*	*merge*	*multisort*
0	-	-	3	4
1	16	16	18	16
2	-	16	36	-
3	-	16	40	-
4	-	16	16	-
Total	16	64	113	20

Just by looking at the task graph, we can see that there are 16 *basicsort* and 64 *basicmerge* tasks in the leaf version. This is because in recursion level 0 *multisort()* is called 4 times; each of these calls, calls 4 *multisort()* (recursion level 1); finally, each of these calls *basicsort()* once: so we have 4 * 4 * 1 = 16 *basicsort* tasks. And there are 64 *basicmerge* tasks because each *multisort()* call in recursion level 0 will, eventually, make 8 *basicmerge()* calls, hence, 8 * 4 *basicmerge* tasks due to the first 4 multisort() calls (not counting the one made in *main()*). The first 2 *merge()* calls will also produce 8 *basicmerge* tasks each. The third *merge()*, on the other hand, will end up creating 16 *basicmerge* tasks. So, in total, we have 8*4 + 8*2 + 16 = 64 *basicmerge* tasks.

* In the *tree* version, 16 tasks execute *basicsort()*, but they are not *basicsort* tasks, just as there are 64 tasks executing *basicmerge()*, but they are not *basicmerge* tasks. Simply, in both *leaf* and *tree* versions the recursive decompositions during both sort and merge phases stop at the same level, meaning that the *leaf* tasks always execute *basicsort()/basicmerge()*, but the *tree* tasks only execute *basicsort()/basicmerge()* when they're at the last recursion level.

Now, let's look at the internal tasks. In the *leaf* case, there are no internal tasks, because, as we know, tasks are created after the decomposition. In the *tree* version, however, this is not true.

In the second table, we've summarized the number of internal tasks seen in the *tree* task graph. As we can see, at recursion level 0 7 tasks are created in total: 4 *multisort* tasks (*multisort1*, *multisort2*, *multisort3*, *multisort4*) and 3 *merge* tasks (*merge1, merge2, merge3*).

At the next recursion level, each of the *multisort* tasks from the upper level creates 4 *multisort* tasks each: 4*4 = 16 in total. At the same recursion level we can see that 18 *merge* tasks are created: 4 *merge1*, 4 *merge2* and 4 *merge3*, 3 *merge4* and 3 *merge5* (3*4 + 2 * 3 = 18).

As we can see in the task graph, no more *multisort* tasks are created (Levels 2, 3 and 4 have no *multisort* tasks).

At level 2 each of the 18 level 1 *merge* tasks create 2 merge *tasks* each (*merge4* and *merge5*).

At level 3 the level 1 *merge3* tasks that come from level 0 *multisort* tasks will end up creating 4 *merge* tasks each (2 *merge4* and 2 *merge5*): 4 * 4 = 16. Also, each of the level 2 *merge* tasks that come from level 0 *merge* tasks will create 2 *merge* tasks (1 *merge4* and 1 *merge5*) each: 12 * 2 = 24. In total, we'll have 16 + 24 = 40 *merge* tasks in level 3.

At the last recursion level, only the level 3 *merge* tasks with level 0 *merge* parent will create 2 tasks each: total = 16 tasks.

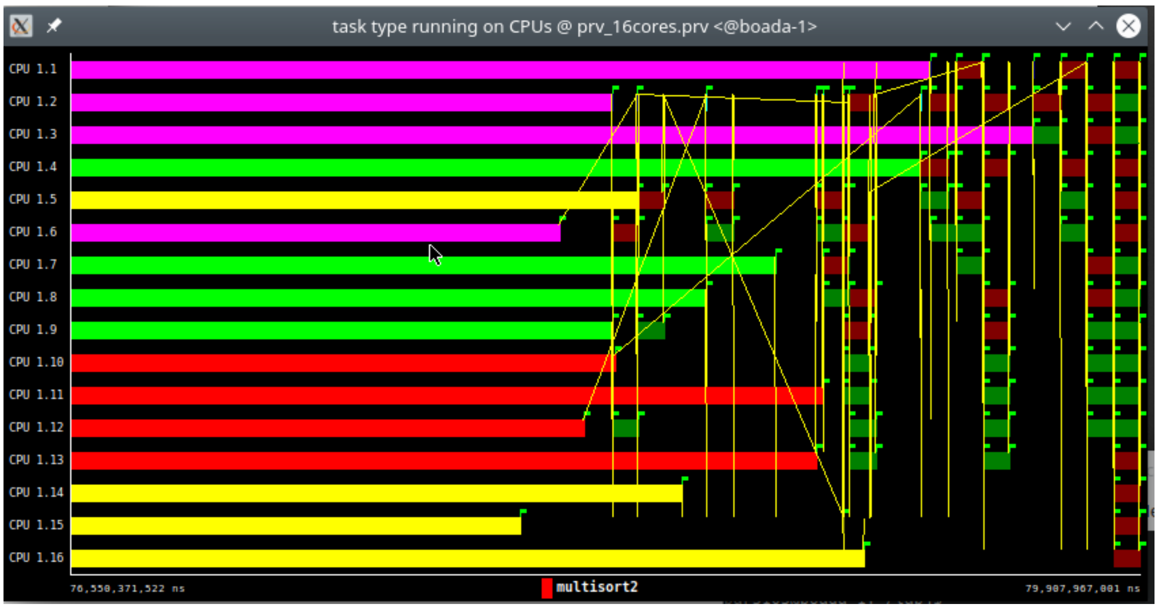
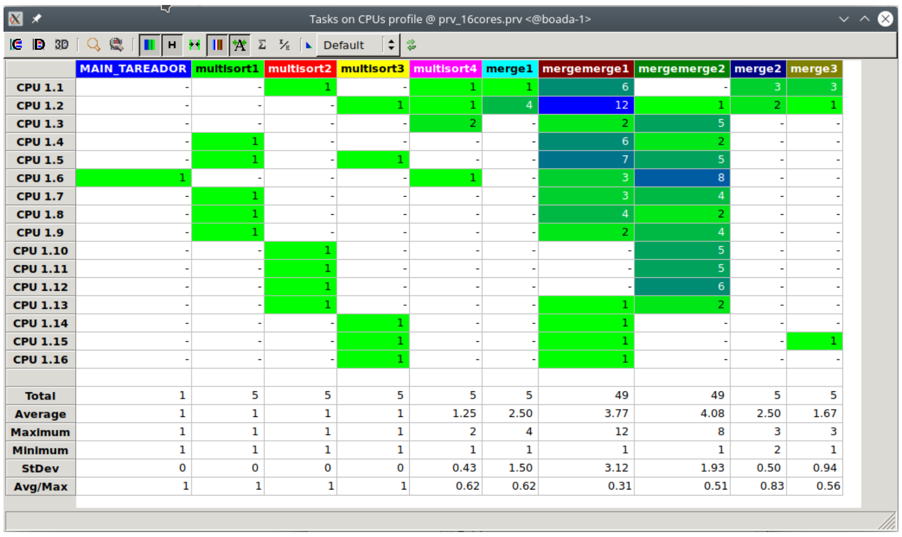


Figure 6: multisort.c execution implementing Tree Strategy decomposition and using 16 threads

We can see that the basic sort is creating very little tasks with merges, but the main execution time is from the basic sort, multisort. In terms of task creation we don't have anything special, all threads execute a basic sort that at the end creates the mergesort.



Tasks on CPUs profile @ prv_16cores.prv <@boada-1>

Default

	MAIN_TAREADOR	multisort1	multisort2	multisort3	multisort4	merge1	mergemerge1	mergemerge2	merge2	merge3
CPU 1.1	-	-	1	-	1	1	6	-	3	3
CPU 1.2	-	-	-	1	1	4	12	1	2	1
CPU 1.3	-	-	-	-	2	-	2	5	-	-
CPU 1.4	-	1	-	-	-	-	6	2	-	-
CPU 1.5	-	1	-	1	-	-	7	5	-	-
CPU 1.6	1	-	-	-	1	-	3	8	-	-
CPU 1.7	-	1	-	-	-	-	3	4	-	-
CPU 1.8	-	1	-	-	-	-	4	2	-	-
CPU 1.9	-	1	-	-	-	-	2	4	-	-
CPU 1.10	-	-	1	-	-	-	-	5	-	-
CPU 1.11	-	-	1	-	-	-	-	5	-	-
CPU 1.12	-	-	1	-	-	-	-	6	-	-
CPU 1.13	-	-	1	-	-	-	1	2	-	-
CPU 1.14	-	-	-	1	-	-	1	-	-	-
CPU 1.15	-	-	-	1	-	-	1	-	-	1
CPU 1.16	-	-	-	1	-	-	1	-	-	-
Total	1	5	5	5	5	5	49	49	5	5
Average	1	1	1	1	1.25	2.50	3.77	4.08	2.50	1.67
Maximum	1	1	1	1	2	4	12	8	3	3
Minimum	1	1	1	1	1	1	1	1	2	1
StDev	0	0	0	0	0.43	1.50	3.12	1.93	0.50	0.94
Avg/Max	1	1	1	1	0.62	0.62	0.31	0.51	0.83	0.56

Figure 7: Tasks profiles using Tree Strategy

The first 4 *multisort* can be executed in parallel but, as we can clearly see, the following tasks (apart from *merge1 merge2* and *merge3,* correspondent to the first 4 *multisorts*) will be on different recursion levels; they will be the children tasks of the recursion level 0 tasks.

Due to this *nesting*, to ensure correct synchronization, a ***taskgroup*** construct will be more adequate, as with *taskgroup* the current task waits not only for the child tasks generated in the *taskgroup*, but also for all the descendants of those child tasks.

We can see that there is only a slight difference between the two strategies, giving *tree* an edge, but it can be seen only when working with very small factors of 10; when working with µs the difference can't be seen.

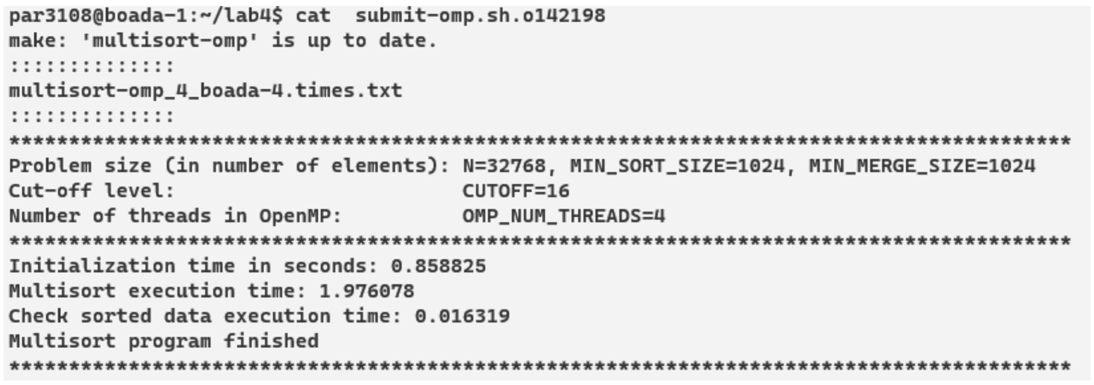
2. Shared-memory parallelisation with OpenMP tasks

2.1. Leaf strategy in *OpenMP*

As seen in the "Parallelisation strategies" section of this document, the first thing we'll do is add *#pragma omp parallel* and *#pragma omp single* before the *multisort()* invocation in *main()*.

The *pragma omp parallel* construct is needed to start the parallel execution: one implicit task is created for each thread, but we want only one to execute the *multisort()* invocation - this is why we use the *single* construct.
Next, we'll create a task for each leaf of the recursion tree (base cases: *basicsort* and *basicmerge*, granularity: one) with *#pragma omp task*, updating *void multisort()* (code below) and *void merge()*.

```
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);
    }
}
```


```
par3108@boada-1:~/lab4$ cat  submit-omp.sh.o142198
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_4_boada-4.times.txt
::::::::::::::
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=4
*****************************************************************************************
Initialization time in seconds: 0.858825
Multisort execution time: 1.976078
Check sorted data execution time: 0.016319
Multisort program finished
*****************************************************************************************
```
Figure 8: Execution times and results obtained from the Leaf Strategy version

We can see that apart from *#pragma omp task*, we've also used *#pragma omp taskwait* twice: once after the *multisort()* invocations and once after the first two *merge()* invocations; this is because we need to ensure that the dependencies between tasks observed during the analysis with *Tareador* are taken into account.

- We will also use the command, ***sbatch submit-strong-omp.sh*** *to generate the plots so we can analyse how good this strategy can be.*

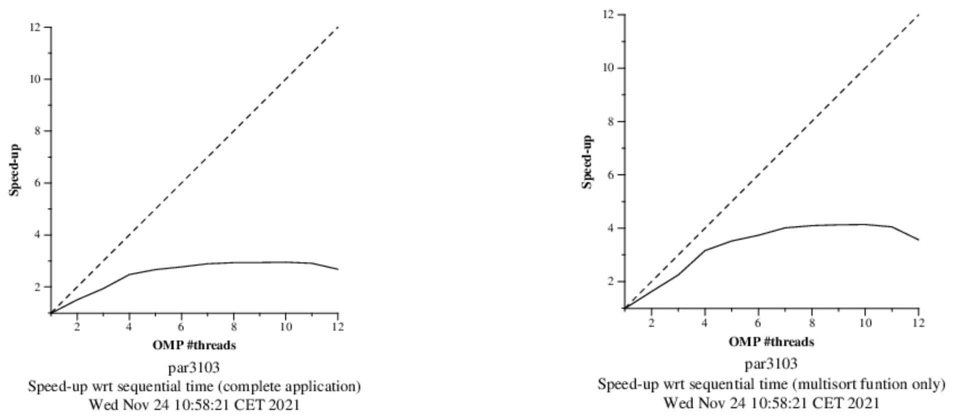


Figure 9: Multisort scalability plot for the Leaf Strategy

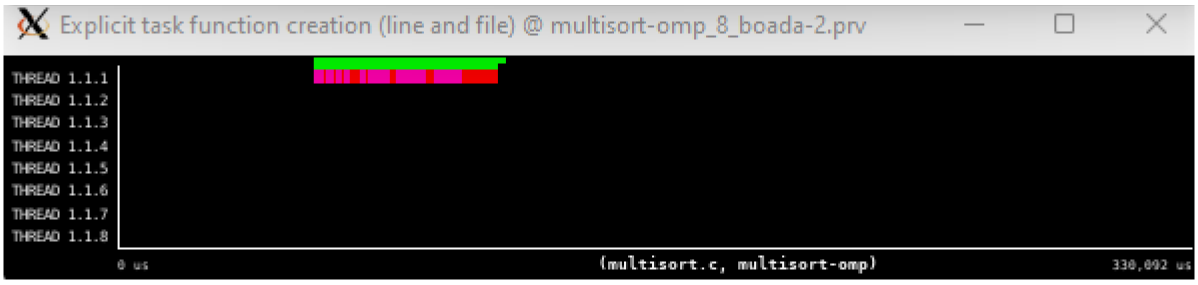


Figure 10: Explicit task execution for the Leaf Strategy

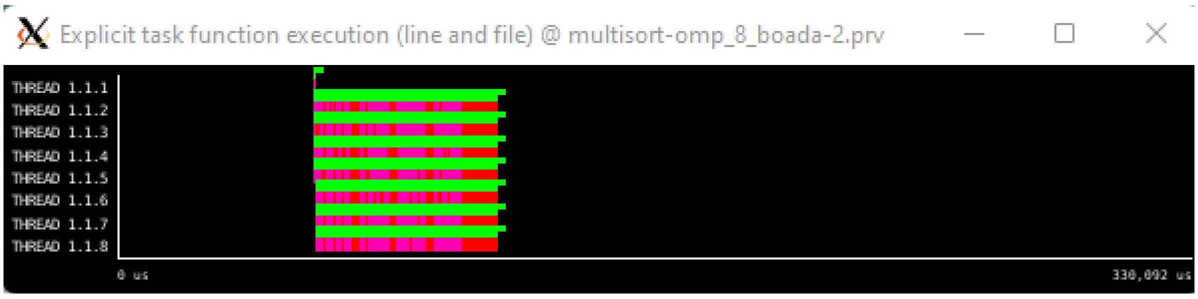


Figure 11: Explicit task creation for the Leaf Strategy

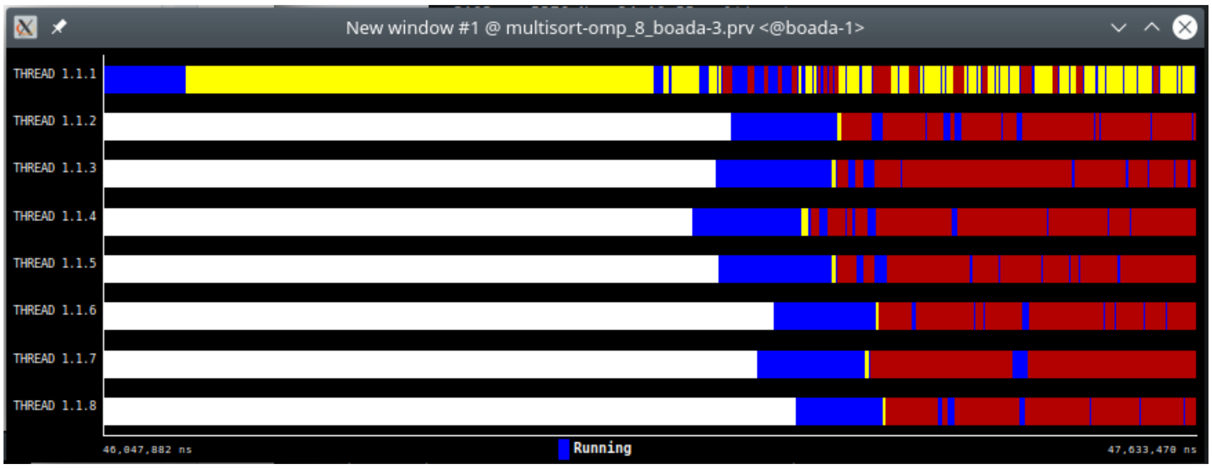


Figure 12: Execution of multisort.c using Leaf Strategy

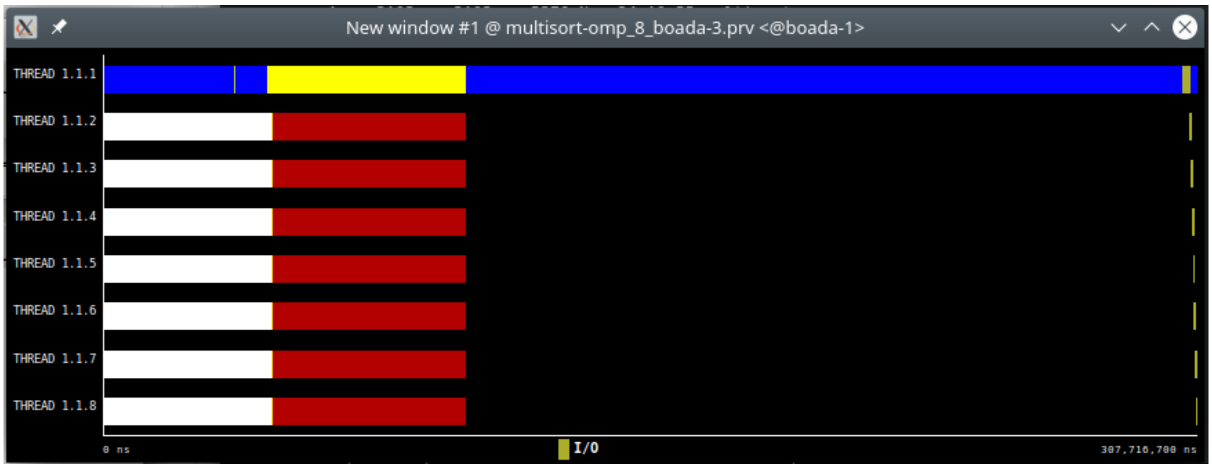
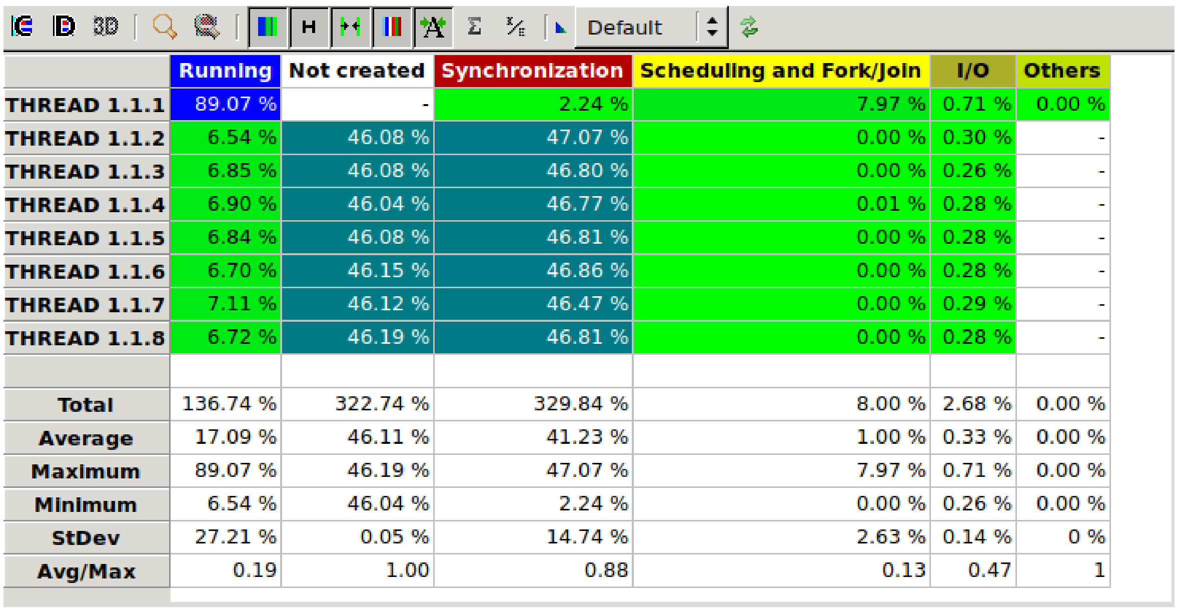


Figure 13: Zoomed execution of multisort.c using Leaf Strategy



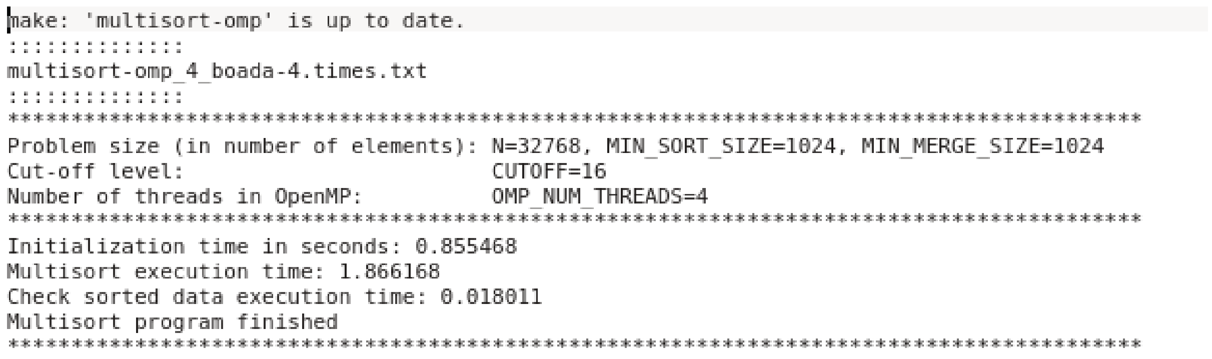
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	89.07 %	-	2.24 %	7.97 %	0.71 %	0.00 %
THREAD 1.1.2	6.54 %	46.08 %	47.07 %	0.00 %	0.30 %	-
THREAD 1.1.3	6.85 %	46.08 %	46.80 %	0.00 %	0.26 %	-
THREAD 1.1.4	6.90 %	46.04 %	46.77 %	0.01 %	0.28 %	-
THREAD 1.1.5	6.84 %	46.08 %	46.81 %	0.00 %	0.28 %	-
THREAD 1.1.6	6.70 %	46.15 %	46.86 %	0.00 %	0.28 %	-
THREAD 1.1.7	7.11 %	46.12 %	46.47 %	0.00 %	0.29 %	-
THREAD 1.1.8	6.72 %	46.19 %	46.81 %	0.00 %	0.28 %	-
Total	136.74 %	322.74 %	329.84 %	8.00 %	2.68 %	0.00 %
Average	17.09 %	46.11 %	41.23 %	1.00 %	0.33 %	0.00 %
Maximum	89.07 %	46.19 %	47.07 %	7.97 %	0.71 %	0.00 %
Minimum	6.54 %	46.04 %	2.24 %	0.00 %	0.26 %	0.00 %
StDev	27.21 %	0.05 %	14.74 %	2.63 %	0.14 %	0 %
Avg/Max	0.19	1.00	0.88	0.13	0.47	1

Figure 14: Overheads related with synchronization and task scheduling

2.2 Tree strategy in OpenMP

This time, we will parallelise the multisort.c algorithm implementing the tree strategy algorithm.

```
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 taskgroup
        {
        #pragma omp task
        merge(n, left, right, result, start, length/2);
        #pragma omp task
        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
        #pragma omp taskgroup
        {
        #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 taskgroup
        {
        #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 task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        basicsort(n, data);
    }
}
```



```
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_4_boada-4.times.txt
::::::::::::::
*****************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=4
*****************************************************************************
Initialization time in seconds: 0.855468
Multisort execution time: 1.866168
Check sorted data execution time: 0.018011
Multisort program finished
*****************************************************************************
```

Figure 15: Execution times and results obtained from the Tree Strategy version

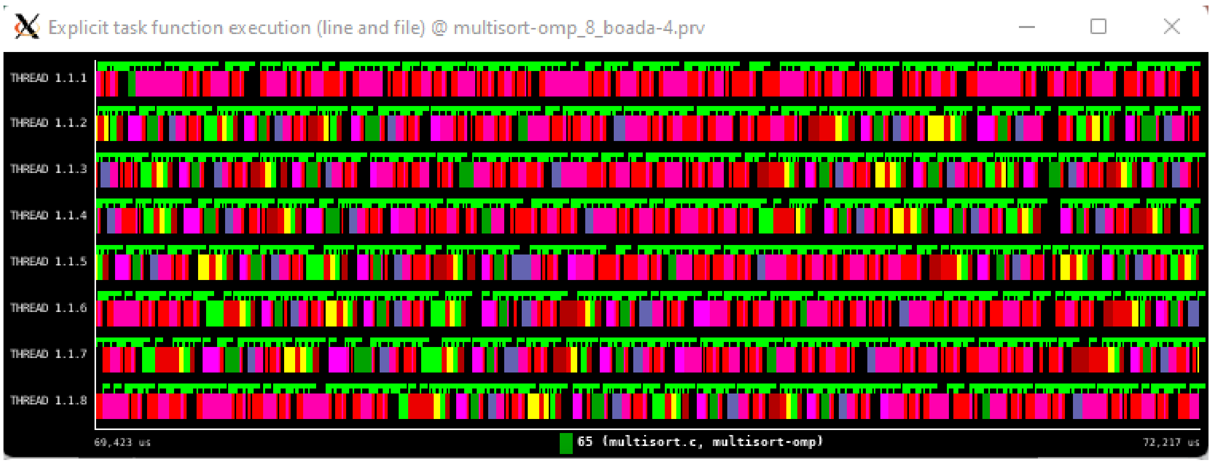


Figure 16: Explicit task execution for the Tree Strategy

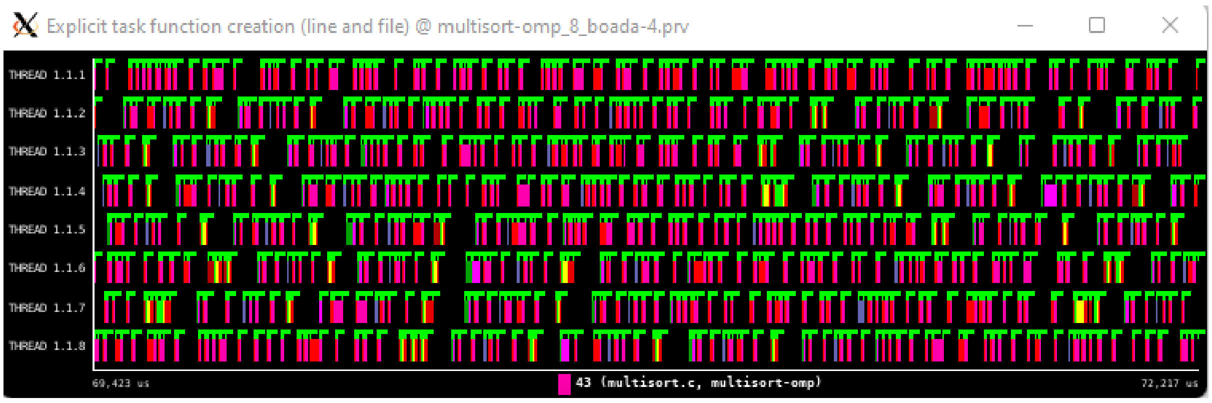


Figure 17: Explicit task creation for the Tree Strategy

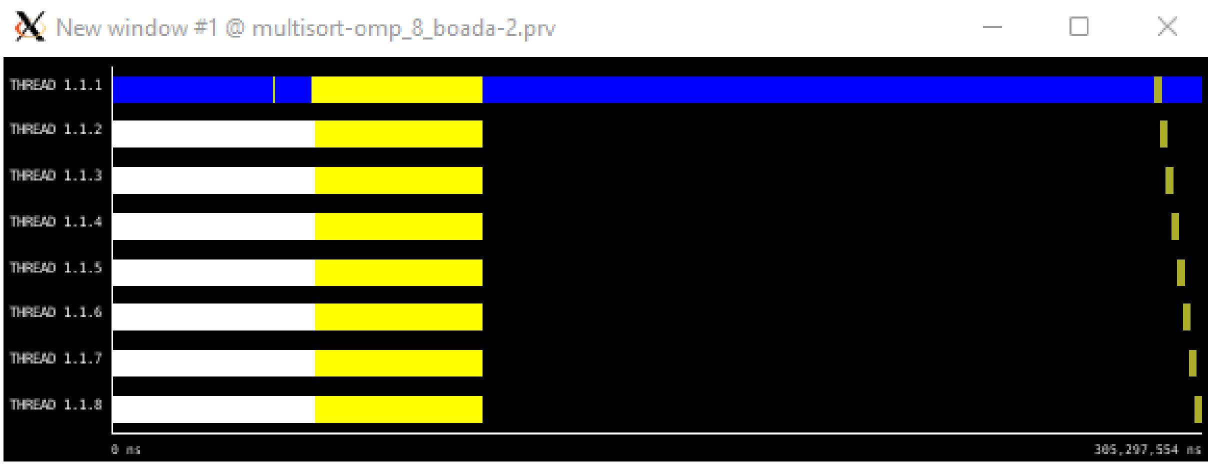


Figure 18: Execution of multisort.c using the Tree Strategy

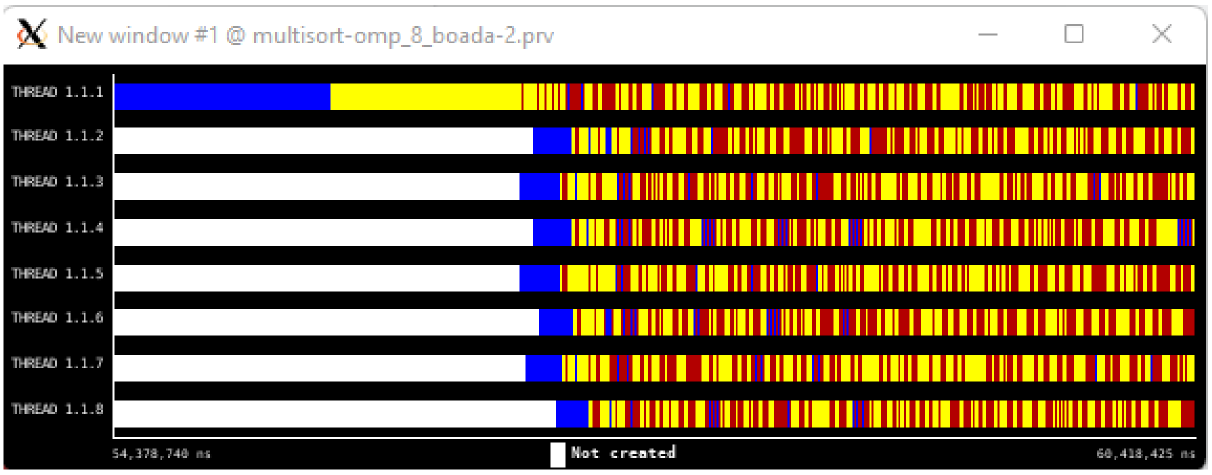
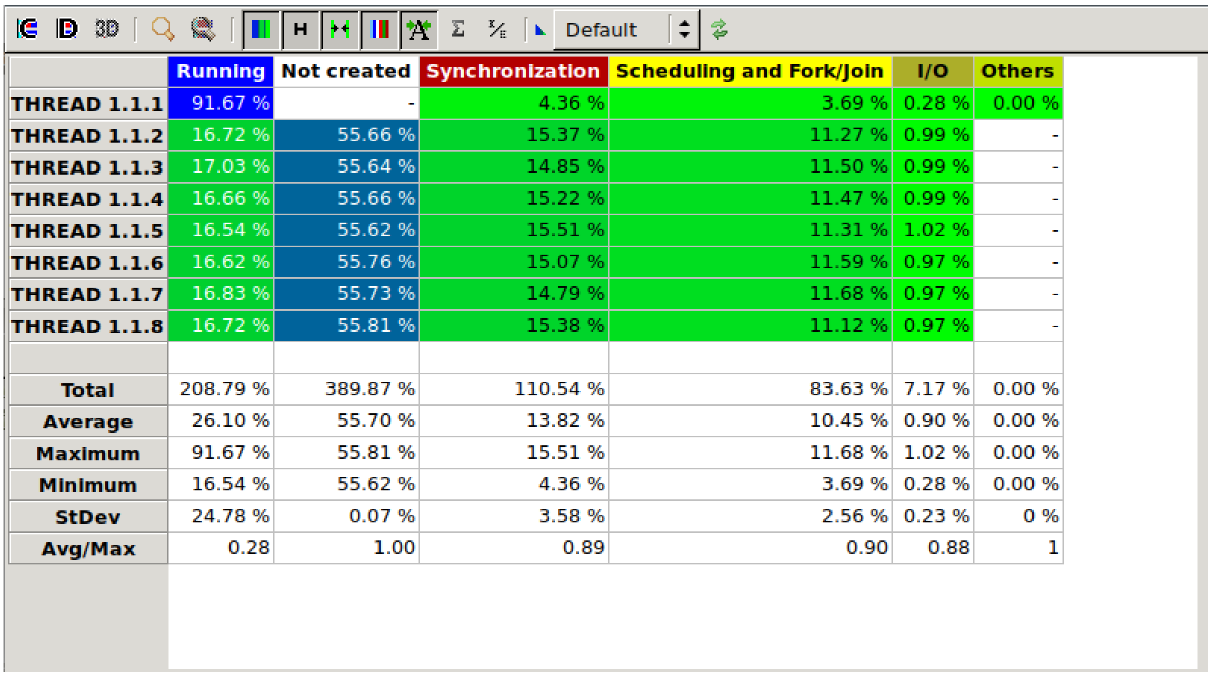


Figure 19: Zoomed execution of multisort.c using the Tree Strategy



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	91.67 %	-	4.36 %	3.69 %	0.28 %	0.00 %
THREAD 1.1.2	16.72 %	55.66 %	15.37 %	11.27 %	0.99 %	-
THREAD 1.1.3	17.03 %	55.64 %	14.85 %	11.50 %	0.99 %	-
THREAD 1.1.4	16.66 %	55.66 %	15.22 %	11.47 %	0.99 %	-
THREAD 1.1.5	16.54 %	55.62 %	15.51 %	11.31 %	1.02 %	-
THREAD 1.1.6	16.62 %	55.76 %	15.07 %	11.59 %	0.97 %	-
THREAD 1.1.7	16.83 %	55.73 %	14.79 %	11.68 %	0.97 %	-
THREAD 1.1.8	16.72 %	55.81 %	15.38 %	11.12 %	0.97 %	-
Total	208.79 %	389.87 %	110.54 %	83.63 %	7.17 %	0.00 %
Average	26.10 %	55.70 %	13.82 %	10.45 %	0.90 %	0.00 %
Maximum	91.67 %	55.81 %	15.51 %	11.68 %	1.02 %	0.00 %
Minimum	16.54 %	55.62 %	4.36 %	3.69 %	0.28 %	0.00 %
StDev	24.78 %	0.07 %	3.58 %	2.56 %	0.23 %	0 %
Avg/Max	0.28	1.00	0.89	0.90	0.88	1

Figure 20: Overheads related with synchronization and task scheduling

- **sbatch submit-strong-omp.sh**

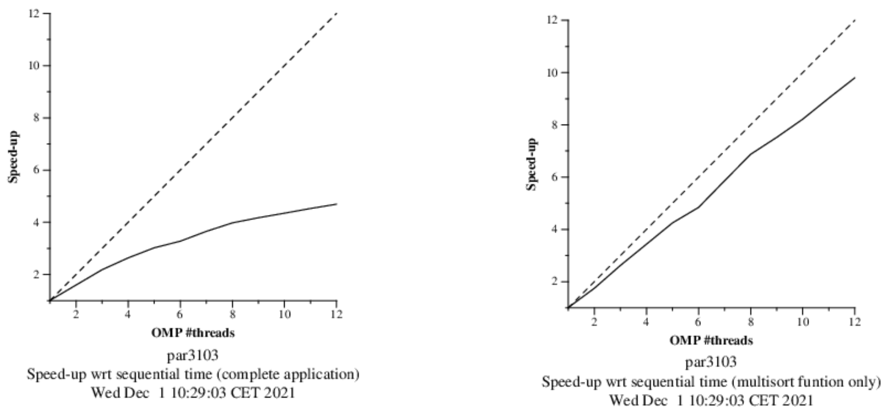


Figure 21: Multisort scalability plot for the Tree Strategy

We will finally look at the outputs of both executions to check that there are no errors for all numbers of threads - there aren't. And yet the speed-ups achieved aren't significant. Both *leaf* version speed-up plots are far from the ideal, and even though in the *tree* version the multisort only plot shows a good speed-up, the complete application one is still very low.

We submit the execution of the binaries using the *submit-extrae.sh* script for 8 processors, tracing the execution of the parallel execution with an input of *-n 128 -s 128 -m 128*.

In the case of the *tree* version, we've had to zoom-in a lot more than with *leaf* to be able to see the *Running* regions within the yellow (*Scheduling*) region seen in the fit-to-scale trace. In the *leaf* version, the yellow regions are only executed by the first thread. This is because the tasks are created during the decomposition in *tree* and after the decomposition in *leaf*.

We can also see that in both *tree* and *leaf*, at the beginning there is a big sequential portion - done by thread 1.1.1, while threads 1.1.2 - 1.1.8 are not even created. This is because of the code that isn't parallelized before the *multisort()* invocation in *main()* - the reason for the differences between the scalability for the whole program and for the multisort function only.

2.3. Task granularity control: the cut-off mechanism

In this session, we would like to control the number of leaves in the recursion tree that are executed by each computational task - in other words, we want to control the granularity of the tasks. We'll do it with a *cut-off* mechanism that allows us to control the maximum recursion level for task generation. We'll do it with *omp_in_final()* intrinsic and the *final* clause for *pragma omp task*. So we finally modified the code of the tree strategy in order to implement a *cut-off* mechanism based on recursion level.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int
level) {
  if (length < MIN_MERGE_SIZE*2L) {
    // Base case
    basicmerge(n, left, right, result, start, length);
  } else {
    // Recursive decomposition
    if(!omp_in_final()) {
#pragma omp taskgroup
{
#pragma omp task final (level>=CUTOFF)
  merge(n, left, right, result, start, length/2, level+1);
#pragma omp task final (level>=CUTOFF)
  merge(n, left, right, result, start + length/2, length/2, level+1);
}
    } else {
      merge(n, left, right, result, start, length/2, level+1);
      merge(n, left, right, result, start + length/2, length/2, level+1);
    }

  }
}

void multisort(long n, T data[n], T tmp[n], int level) {
  if (n >= MIN_SORT_SIZE*4L) {
    // Recursive decomposition
    if(!omp_in_final()) {
#pragma omp taskgroup
{
#pragma omp task final (level>=CUTOFF)
```

```
  multisort(n/4L, &data[0], &tmp[0], level+1);
#pragma omp task final (level>=CUTOFF)
  multisort(n/4L, &data[n/4L], &tmp[n/4L], level+1);
#pragma omp task final (level>=CUTOFF)
  multisort(n/4L, &data[n/2L], &tmp[n/2L], level+1);
#pragma omp task final (level>=CUTOFF)
  multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], level+1);
}
#pragma omp taskgroup
{
#pragma omp task final (level>=CUTOFF)
  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, level+1);
#pragma omp task final (level>=CUTOFF)
  merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, level+1);
}

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

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

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

      merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, level+1);
    }
  } else {
    // Base case
    basicsort(n, data);
  }
}
```

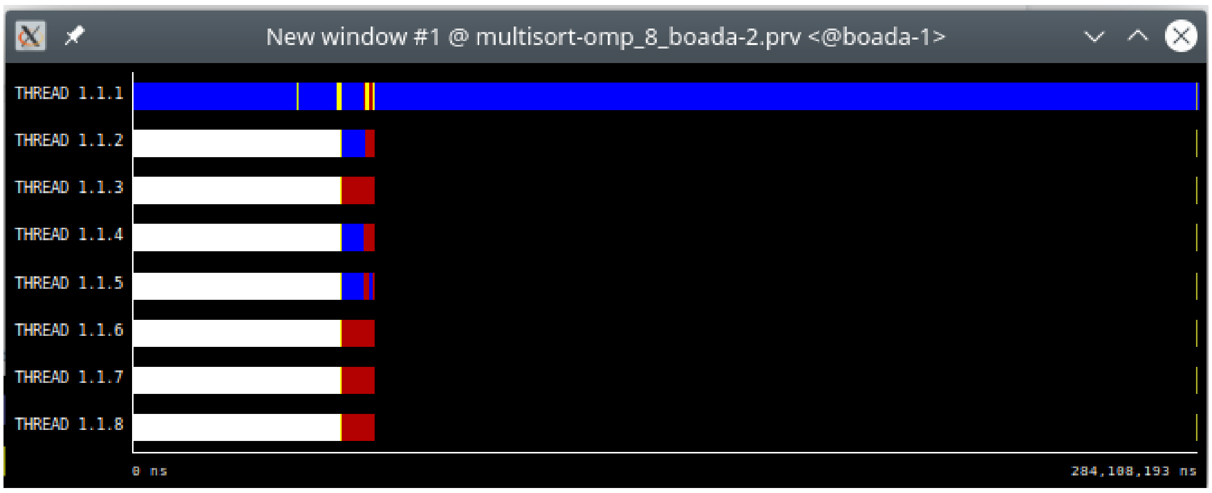


Figure 22: Execution of multisort.c using Tree Strategy with cut off set to 0

As the picture below shows, when the cut-off is set to 0, we just have one recursion level in both multisort() and merge() algorithms.

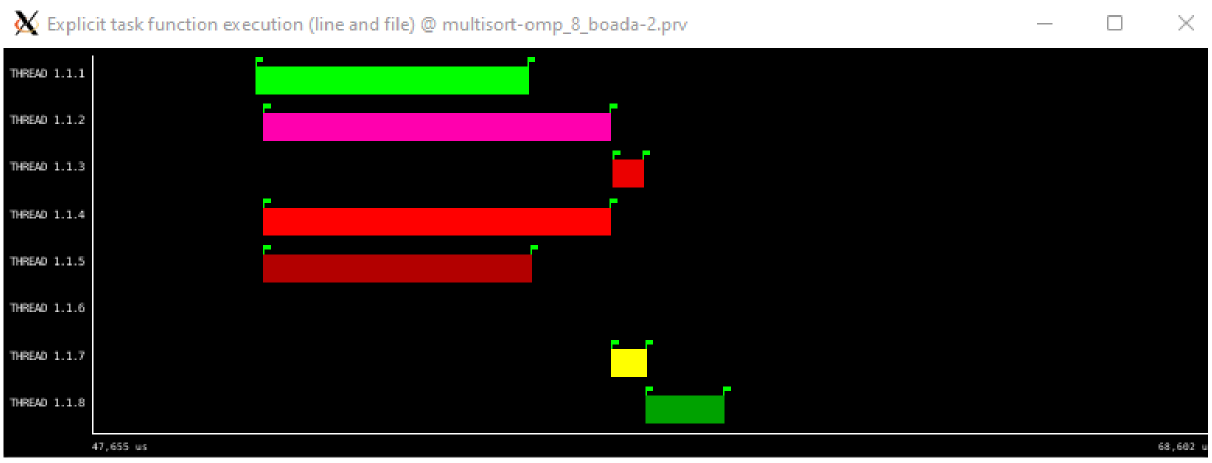


Figure 23: Explicit task execution for the Tree Strategy with cut-off set to 0

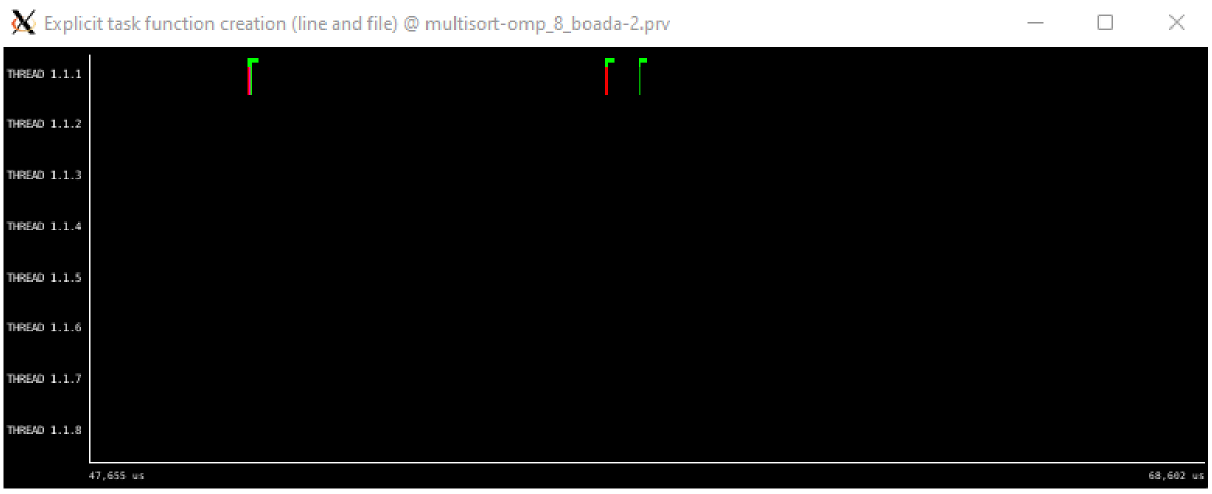
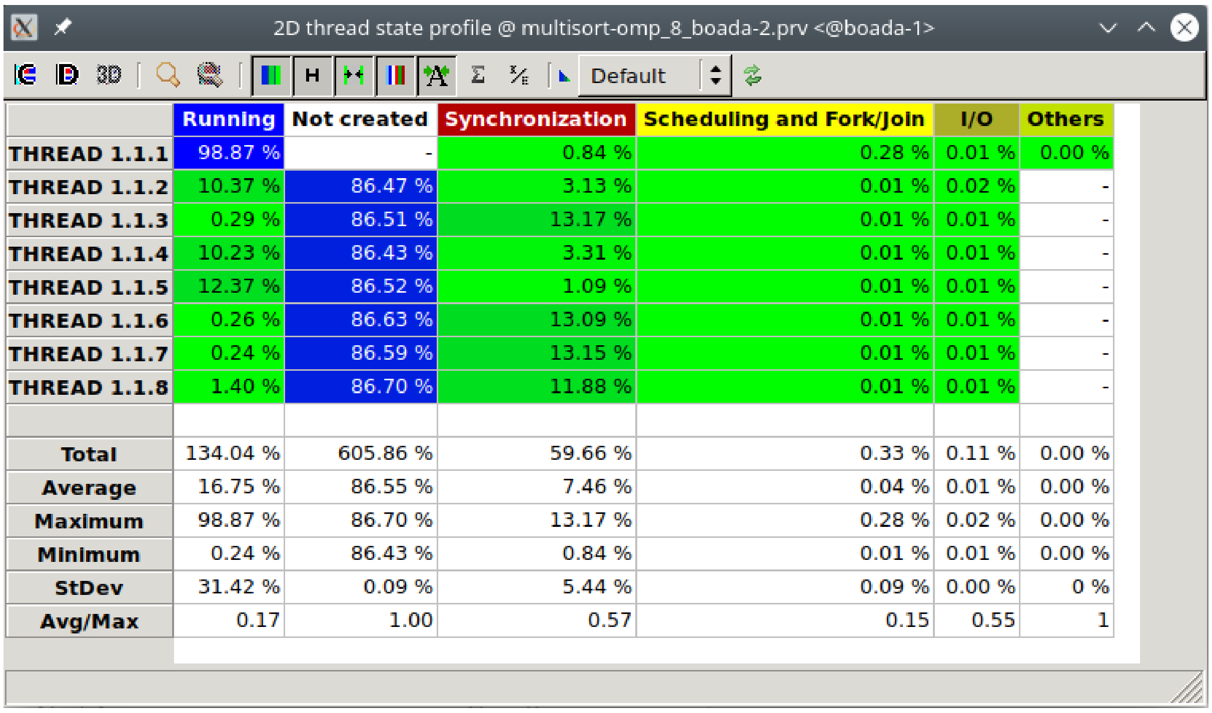


Figure 24: Explicit task creation for the Tree Strategy with cut-off set to 0

We can see in *Figure 21*, one yellow and one red task executing in parallel, in threads 1.1.5 and 1.1.7, correspondingly. These are the *merge* tasks of the code (the first 2 *merge* tasks in *void multisort()*). We can clearly see that they wait for the four *multisort* tasks to be done, due to the barrier produced by *taskgroup*. Again, there is only one set of them, due to *CUTOFF* = 0.



2D thread state profile @ multisort-omp_8_boada-2.prv <@boada-1>

Default

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	98.87 %	-	0.84 %	0.28 %	0.01 %	0.00 %
THREAD 1.1.2	10.37 %	86.47 %	3.13 %	0.01 %	0.02 %	-
THREAD 1.1.3	0.29 %	86.51 %	13.17 %	0.01 %	0.01 %	-
THREAD 1.1.4	10.23 %	86.43 %	3.31 %	0.01 %	0.01 %	-
THREAD 1.1.5	12.37 %	86.52 %	1.09 %	0.01 %	0.01 %	-
THREAD 1.1.6	0.26 %	86.63 %	13.09 %	0.01 %	0.01 %	-
THREAD 1.1.7	0.24 %	86.59 %	13.15 %	0.01 %	0.01 %	-
THREAD 1.1.8	1.40 %	86.70 %	11.88 %	0.01 %	0.01 %	-
Total	134.04 %	605.86 %	59.66 %	0.33 %	0.11 %	0.00 %
Average	16.75 %	86.55 %	7.46 %	0.04 %	0.01 %	0.00 %
Maximum	98.87 %	86.70 %	13.17 %	0.28 %	0.02 %	0.00 %
Minimum	0.24 %	86.43 %	0.84 %	0.01 %	0.01 %	0.00 %
StDev	31.42 %	0.09 %	5.44 %	0.09 %	0.00 %	0 %
Avg/Max	0.17	1.00	0.57	0.15	0.55	1

Figure 25: Overheads related with synchronization and task scheduling

Next, we set the cut-off value to 1, so that now we will have two recursion levels in both multisort() and merge() algorithms.

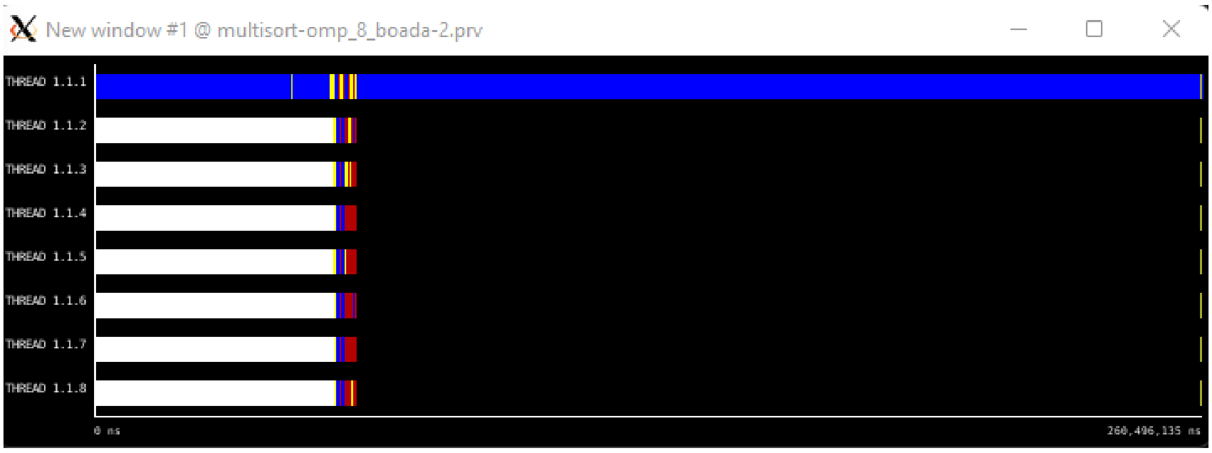


Figure 26: Execution of multisort.c using Tree Strategy with cut off set to 1

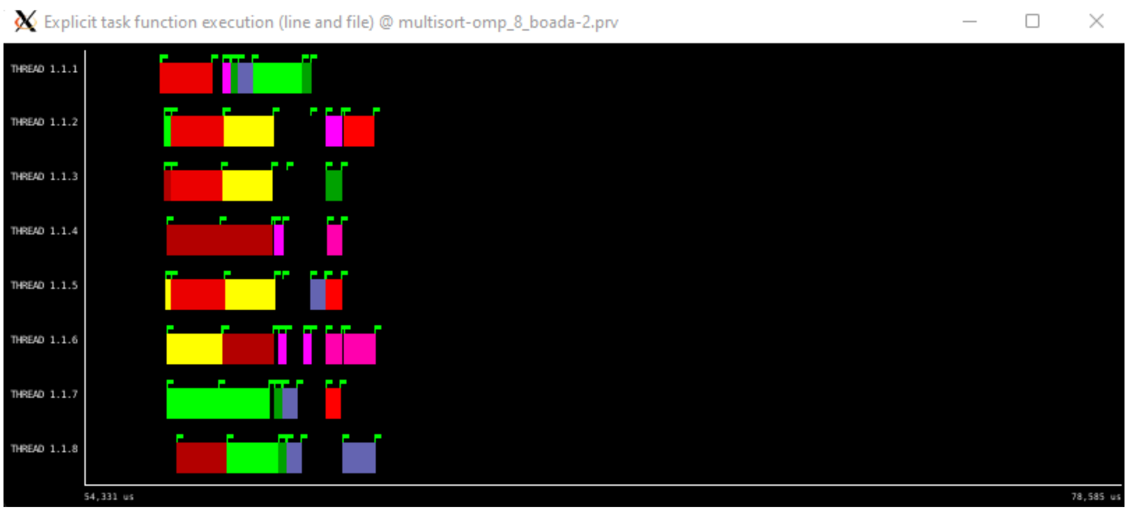


Figure 27: Explicit task execution for the Leaf Strategy with cut-offf set to 1

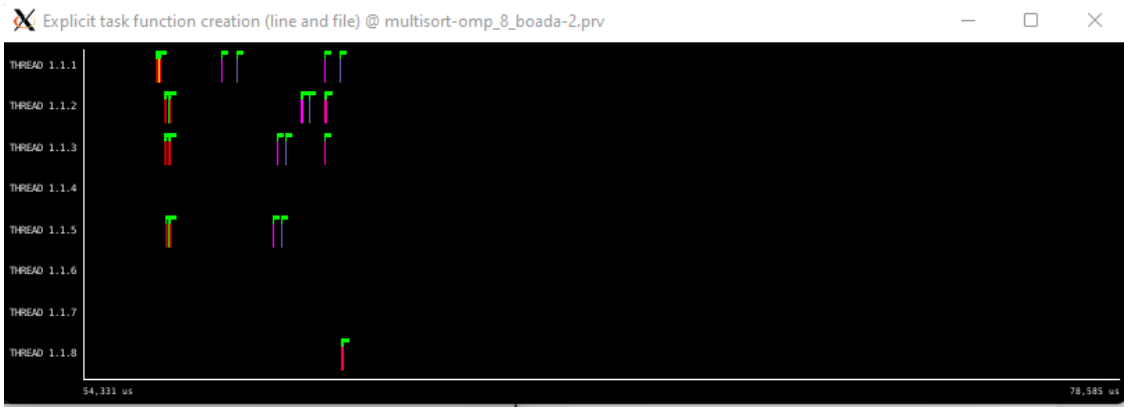
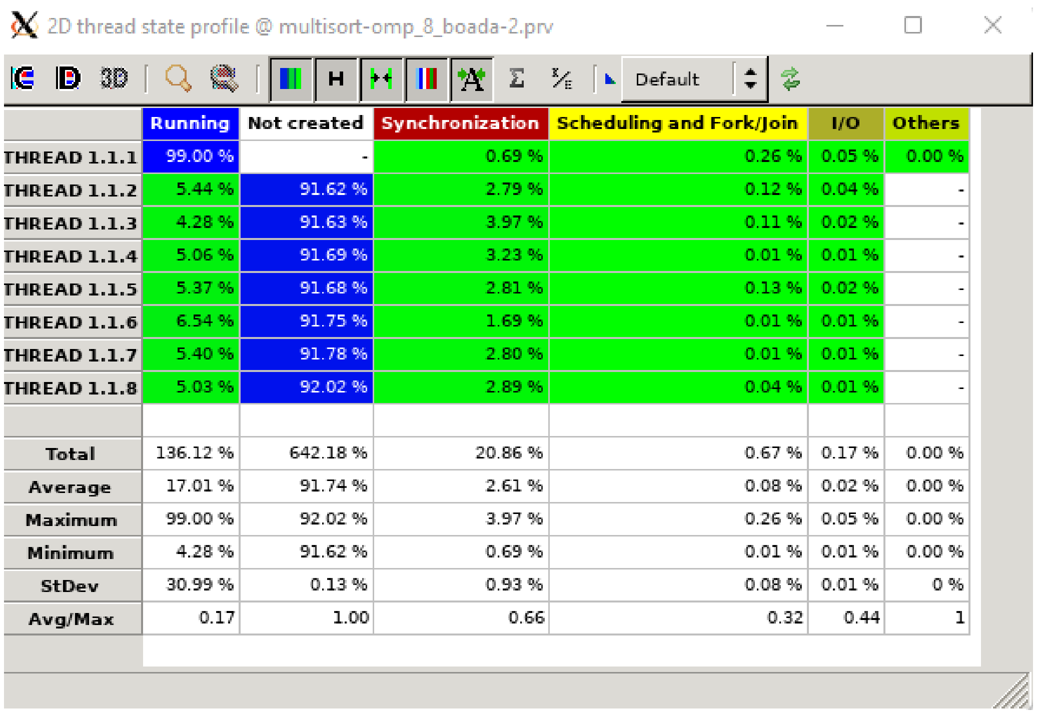


Figure 28: Explicit task creation for the Leaf Strategy with cut-off set to 1



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.00 %	-	0.69 %	0.26 %	0.05 %	0.00 %
THREAD 1.1.2	5.44 %	91.62 %	2.79 %	0.12 %	0.04 %	-
THREAD 1.1.3	4.28 %	91.63 %	3.97 %	0.11 %	0.02 %	-
THREAD 1.1.4	5.06 %	91.69 %	3.23 %	0.01 %	0.01 %	-
THREAD 1.1.5	5.37 %	91.68 %	2.81 %	0.13 %	0.02 %	-
THREAD 1.1.6	6.54 %	91.75 %	1.69 %	0.01 %	0.01 %	-
THREAD 1.1.7	5.40 %	91.78 %	2.80 %	0.01 %	0.01 %	-
THREAD 1.1.8	5.03 %	92.02 %	2.89 %	0.04 %	0.01 %	-
Total	136.12 %	642.18 %	20.86 %	0.67 %	0.17 %	0.00 %
Average	17.01 %	91.74 %	2.61 %	0.08 %	0.02 %	0.00 %
Maximum	99.00 %	92.02 %	3.97 %	0.26 %	0.05 %	0.00 %
Minimum	4.28 %	91.62 %	0.69 %	0.01 %	0.01 %	0.00 %
StDev	30.99 %	0.13 %	0.93 %	0.08 %	0.01 %	0 %
Avg/Max	0.17	1.00	0.66	0.32	0.44	1

Figure 29: Overheads related with synchronization and task scheduling

Finally, we submit this last version with ***submit-cutoff-omp.sh*** to discover the best values of the *cut-off* variable in terms of performance.

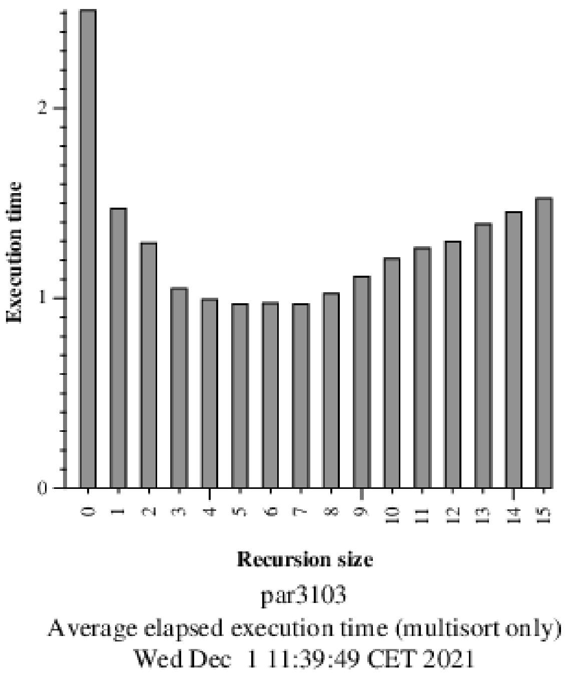


Figure 30: Average execution time plot for the tree strategy with recursive level control

With the data obtained with the plot above, we modified the values of *cut-off*, *sort_size* and *merge_size* in the *submit-strong-omp.sh* script, first, setting the *cut-off* to 5, and then, to 16 (default and maximum). In both cases, we changed the *sort_size* and *merge_size* values to 128.

We set the *cut-off* to 5 because, as the graph above shows (*Figure 13*), it is the most optimum value.

sbatch submit-strong-omp.sh

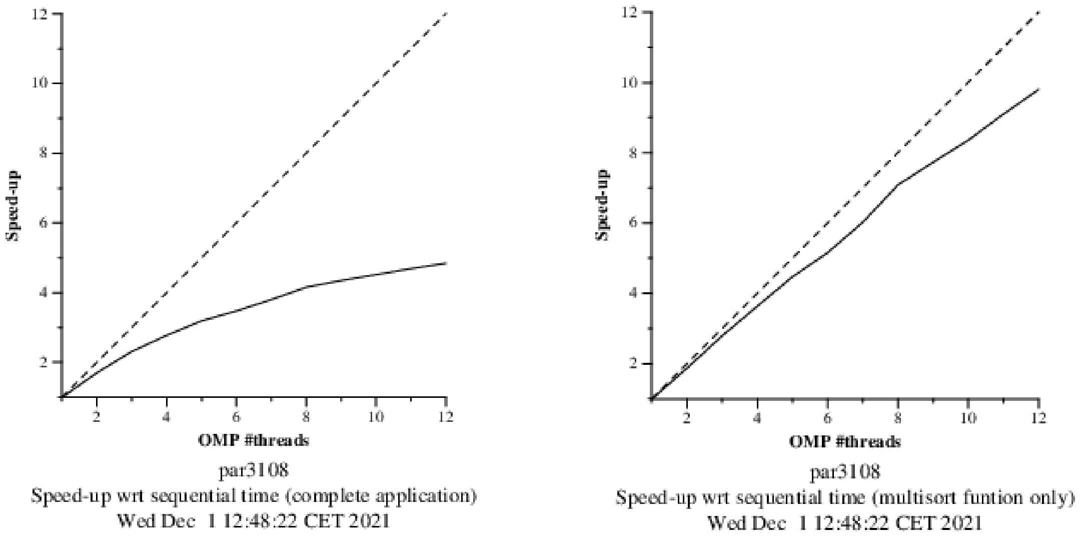


Figure 31: Multisort scalability plot for the tree strategy with recursive level control (5)

And finally, with *cut-off*'s default value (16).

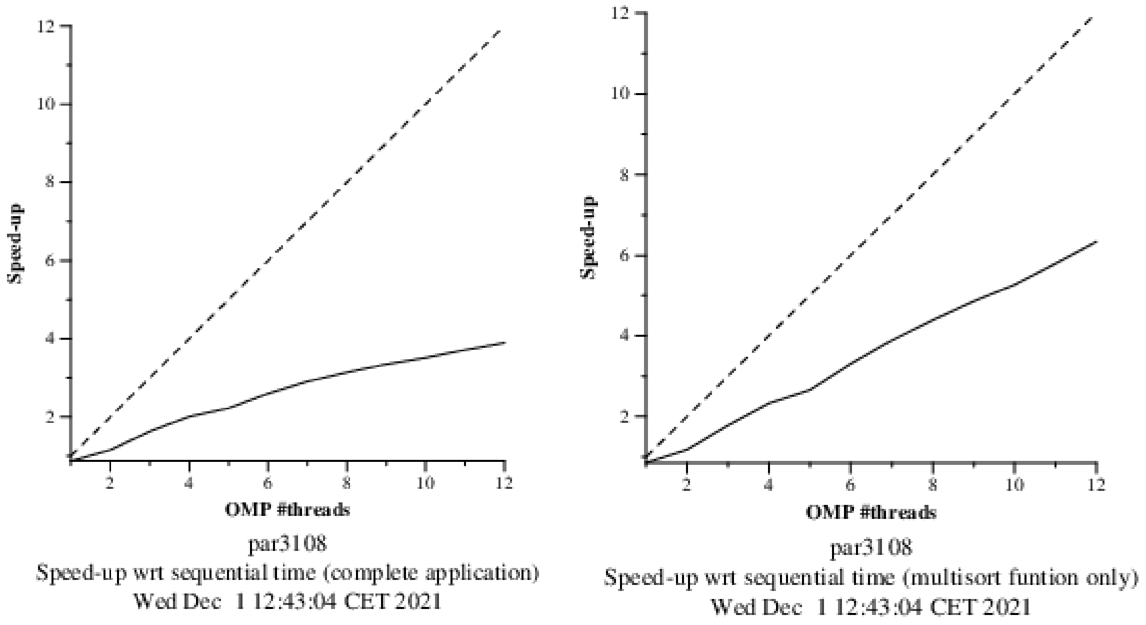


Figure 32: Multisort strong scalability plot for the tree strategy with recursive level control (16)

3. Using OpenMP task dependencies

Finally, we will update once more the tree version, this time trying to synchronize by *point-to-point* dependencies. In order to do so, we added several *depend* clauses, to be sure that the specified task won't be executed until those from the *depend* clause have finished.

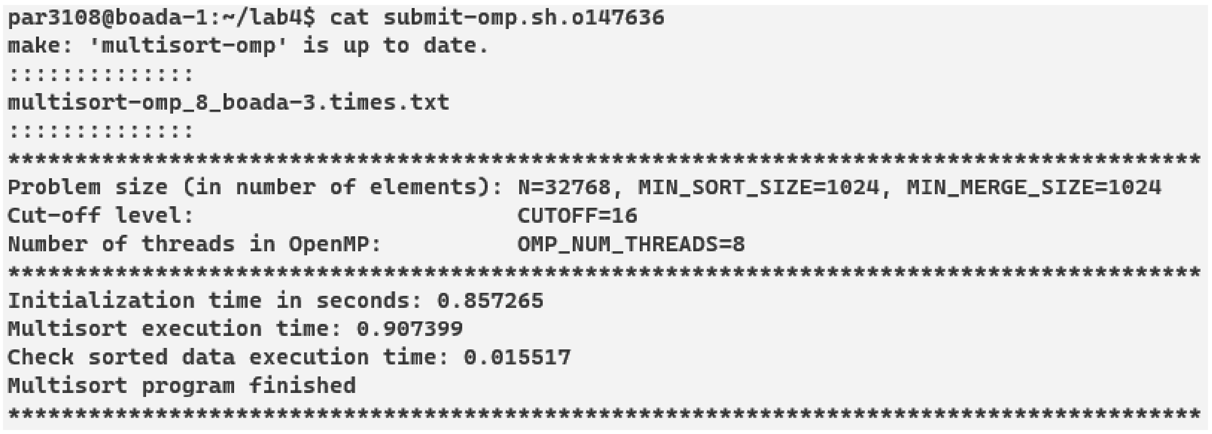
The following code is the one we obtained after adding those clauses.

```
void merge (long n, T left[n], T right[n], T result[n * 2], long start, long length,
int level) {
  if (length < MIN_MERGE_SIZE * 2L) {
      // Base case
#pragma omp taskwait
      basicmerge (n, left, right, result, start, length);
    } else {
      // Recursive decomposition
      if (!omp_in_final ()){
#pragma omp taskgroup
          {
#pragma omp task final (level>=CUTOFF)
            merge (n, left, right, result, start, length / 2, level + 1);
#pragma omp task final (level>=CUTOFF)
            merge (n, left, right, result, start + length / 2, length / 2, level + 1);
          }
        } else {
#pragma omp task final (level>=CUTOFF)
          merge (n, left, right, result, start, length / 2, level + 1);
#pragma omp task final (level>=CUTOFF)
```

```
          merge (n, left, right, result, start + length / 2, length / 2, level + 1);
        }
    }
}

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


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

          merge (n / 4L, &data[0], &data[n / 4L], &tmp[0], 0, n / 2L, level + 1);
           merge (n / 4L, &data[n / 2L], &data[3L * n / 4L], &tmp[n / 2L], 0, n / 2L,
level + 1);
          merge (n / 2L, &tmp[0], &tmp[n / 2L], &data[0], 0, n, level + 1);
        }
    } else {
      // Base case
      basicsort (n, data);
    }


}
```

Once we have the code, we compile using 8 processors and, with the image below, we check that the algorithm is working properly, and the data is ordered.

sbatch submit-omp.sh multisort-omp 8

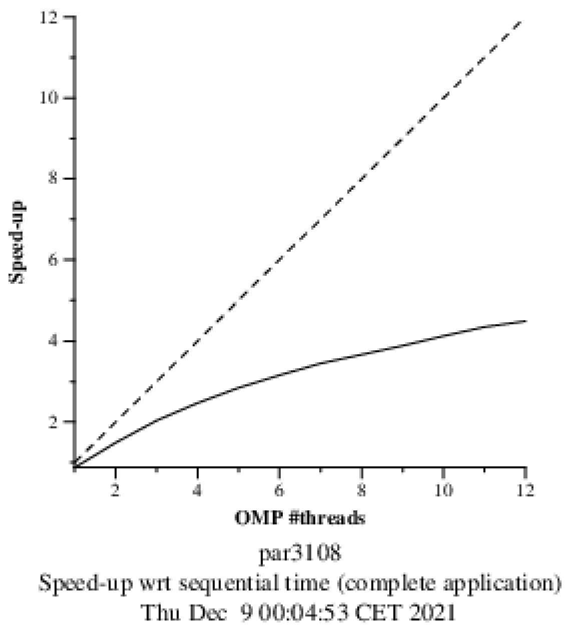


```
par3108@boada-1:~/lab4$ cat submit-omp.sh.o147636
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_8_boada-3.times.txt
::::::::::::::
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=8
*****************************************************************************************
Initialization time in seconds: 0.857265
Multisort execution time: 0.907399
Check sorted data execution time: 0.015517
Multisort program finished
*****************************************************************************************
```

Figure 33: Execution times and results obtained from the Tree Strategy version with cut off

To analyse the scalability of this new version, we submitted the code, and obtained the following plots.

sbatch submit-strong-omp.sh



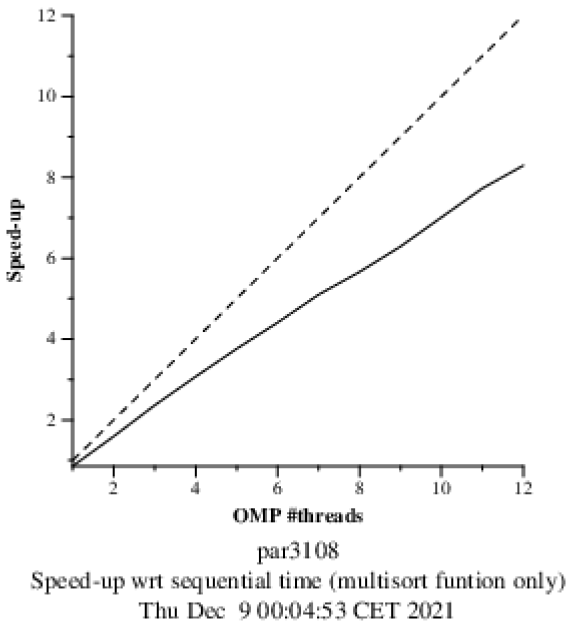


Figure 34: Multisort strong scalability plot for the tree strategy using cut off and task dependencies point-to-point.

In comparison to the scalability plots obtained in the previous versions of the tree strategy, this one is far from the best (the one with recursive level control 5), though is very similar to the basic. So in terms of performance, we cannot say that this new version with tasks synchronization *point-to-point* is better than the others.
In terms of performance, there aren't big changes. Comparing the speed-up plots with the ones of other other versions we can see that they are quite similar. In terms of programmability, however, the previous versions are simpler.

Regarding the difficulty of the code, this one is definitely far more complex.

Finally, we traced the execution with 8 processors and obtained the graphs below.

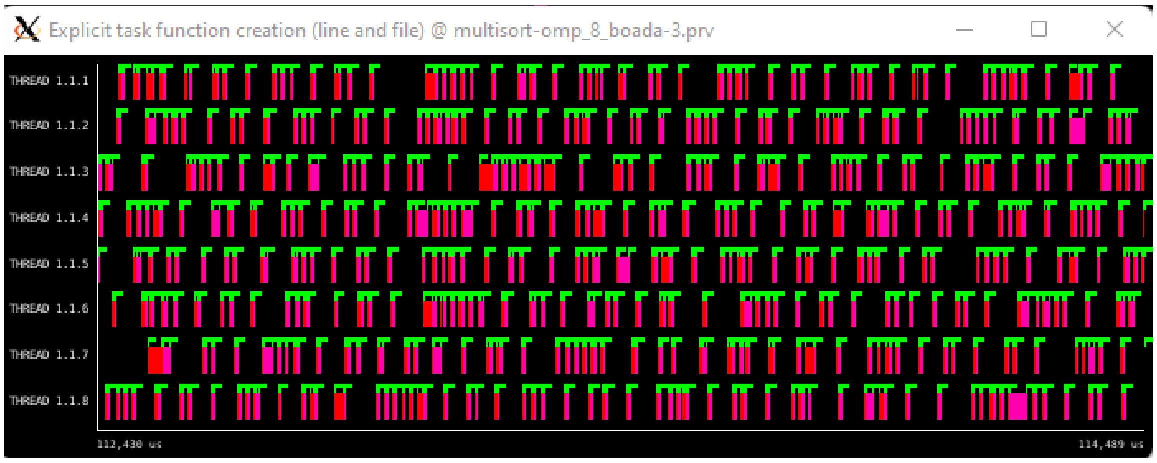


Figure 35: Explicit task creation for the tree strategy version with cut off and dependencies point-to-point

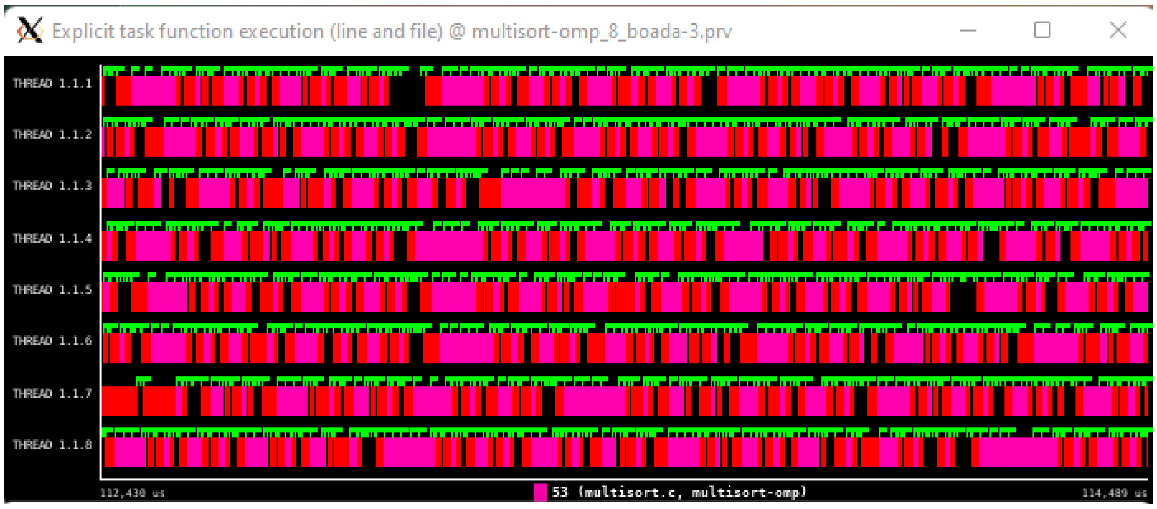


Figure 36: Explicit task execution for the tree strategy version with cut off and dependencies point-to-point

As we can see in the pictures above, the task creation and execution plot is nothing like the one from the cut-off version, because this time, we are using all eight threads to both create and execute tasks, instead of just some of them.

It is very similar, though, to the first and more basic version of the tree strategy (*figure 16* and *figure 17*), both strategies use all available resources (threads) to create and execute the necessary tasks and dependencies.

In conclusion, we could say that the *depend* clauses are creating a considerable amount of overhead in order to synchronize tasks, so we can see that these clauses are more powerful in terms of synchronization semantics.

4. Conclusions

After having studied and explored several versions for the parallelisation of the multisort() algorithm, it seems obvious that the best times and results in terms of performance are obtained when the tree strategy is being used.

After having implemented all possible variations of the tree strategy, first with *cut-off* and then adding the tasks dependencies *point-to-point*, we can conclude that the way dependencies among tasks are created has considerable consequences in terms of performance.