PAR Laboratory Assignment
Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

E. Ayguadé, J. R. Herrero, P. Martínez-Ferrer,
J. Morillo, J. Tubella and G. Utrera Spring
2021-22

Maria Montalvo Falcón (par 4214)
Victor Pla Sanchis (par4219)
Group: 42
Date: 12/05/2022
Course: 2021-2022 Q2

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

1 Introduction

In this assignment the main purpose is to explore the usage of parallelism with recursive algorithms. To do that, we were provided with an implementation of the merge sort algorithm. It is known that this algorithm first divides the structure (a vector in our case) in two parts of the same length, sort each part and then merge both parts with the merge function.
In order to properly work the code with the different tools, this assignment is splitted in three sessions. The first one is focused on *Tareador* tool and to inspect the dependencies. The second one is oriented on implementing different versions of code with OpenMP clauses. And finally, the third one session in which we worked out the clauses of dependencies with OpenMP clauses; a more natural and usable way to implement parallelism.

2
Analysis with Tareador

In this first laboratory session of the fourth assignment, we have worked with the help of *Tareador* tool to do the analysis related to leaf and tree strategy of the Merge Sort code proposed. In order to do that and support our ideas, we have generated the considered graphic figures.

2.0 Preliminary

First of all, we started understanding the code given. Then we executed the sequential version to see how it would behave. The command line to do that was: `./multisort-seq [-n 32768 -s 1024 -m 1024]` (values by default if nothing else is specified).

```
par4214@boada-1:~/lab4$ ./multisort-seq
********************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
********************************************************************************
Initialization time in seconds: 0.866203
Multisort execution time: 6.452966
Check sorted data execution time: 0.015436
Multisort program finished
********************************************************************************
```

Figure 2.0.1 Execution time of sequential proposed code

The results given here above shows that the program was correctly executed and the execution time of multisort (without initialization) was approximately 6.45 secods. These results have been applied to compare the scalability of the program in the following snections.

2.1 Leaf strategy analysis with Tareador

After that first contact with the code, we have analysed the implementation of mergesort with leaf strategy with the help of Tareador tool. Firstly, we modified the code to create the respective version (see Figure 2.1.1).

```
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("START LEAF TASK MERGE");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("END LEAF TASK MERGE");
    } 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("START LEAF TASK SORT");
        basicsort(n, data);
        tareador_end_task("END LEAF TASK SORT");
    }
}
```

Figure 2.1.1 Code modified for Leaf Strategy

As seen, there is a task for every call to *basicmerge* function and for every *basicsort* one (base cases of recursion of each function). We have created each task adding the clause `tareador_start_task("NAME")` and defining it ends with `tareador_end_task("NAME")`.

After that, we compiled the program with the command line: `make multisort-tareador` and executed with: `run-tareador.sh` script generating a small sample (-n 32 -s 2048 -m 2048). Then, we generated the task graph dependency shown below with *Tareador*.

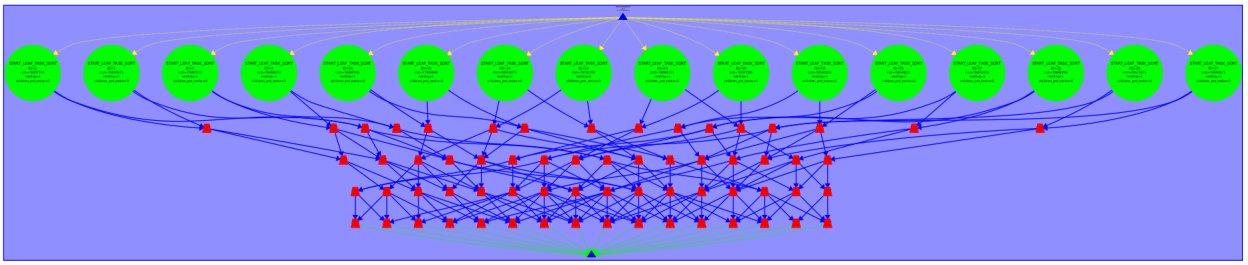

Figure 2.2.1 Leaf strategy TDG

As shown on figure 2.2.1, the green tasks are the ones related to the calls to function sort. These calls were perfectly parallelizable as the sort of a piece of the vector has no dependencies with other pieces of the vector. In addition to that, each merge task has dependendencies arriving to them; the reason why this happened is because there were more than one level of recursivity, and each edge created was related to each level.
Talking in terms of granularity, the TDG shows a big imbalance between the tasks related to *basicsort* calls and *basicmerge* ones. The calls of *basicsort* in green colour had more work of computation than merge ones.

To explore these dependencies deeply, we decided to inspect some edges with *Tareador.* Here below one could see these dependencies come from variable data and from variable tmp.
We have taken this into account when implementing the code with OpenMP clauses to protect these variables (in blue or orange the dependencies actions that produce the variable).

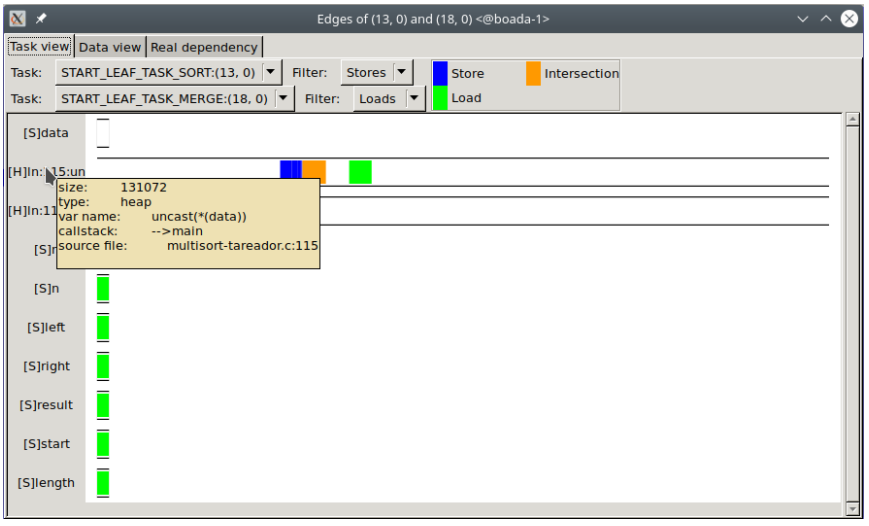

Figure 2.2.2 Dependencies leaf strategy

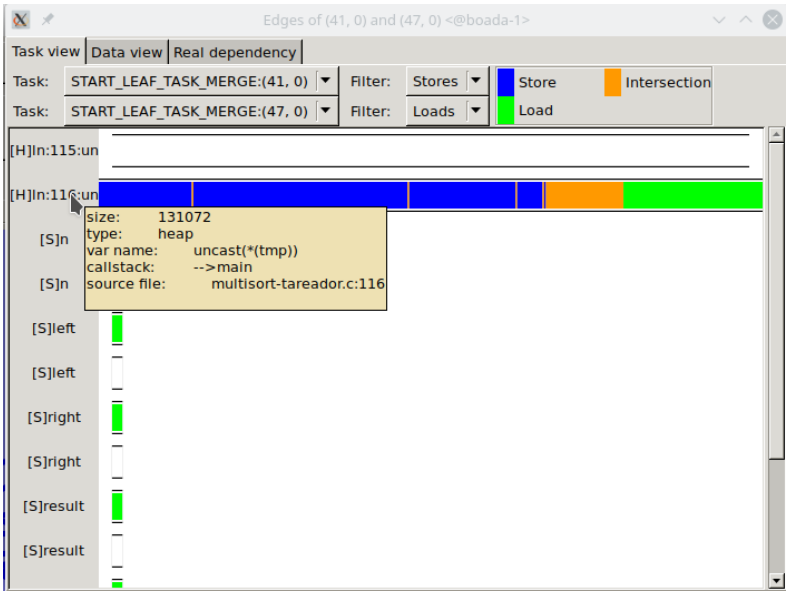

Figure 2.2.3 Dependencies leaf strategy

2.2 Tree strategy analysis with tareador

After analysing leaf strategy, we have seen how tree strategy behaved with Tareador. This strategy consisted on creating a task for every recursive call of merge and sort methods that multisort provides. So we moved on to create the sort tasks with the name “MERGE TASK” and the sort tasks named “SORT TASK” as shown in the image below.

```
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("MERGE TASK");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("MERGE TASK");

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

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

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

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

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

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

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

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

Figure 2.2.1 Code modified for Tree Strategy

As seen, in tree strategy the tasks are created on the recursive case instead of the base one. Next, we have visualised the TDG by executing `make multisort-tareador` and `run-tareador.sh` command lines. In figure 2.2.2 the green circles are representing the sort tasks and the red rectangles are representing the merge ones.

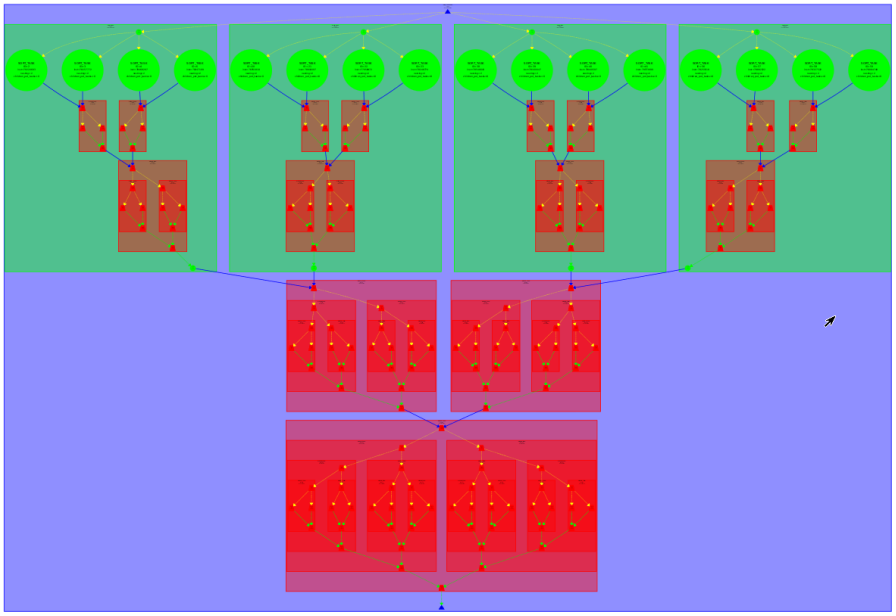

Figure 2.2.2 Tree strategy TDG

We clearly saw that all the tasks were inside of another one. That's because we were creating tasks inside of other tasks already created with a recursive structure. Comparing this graph with the previous about leaf strategy, one can note that tree strategy is creating a huge number of tasks and that in some proportion the creation of these tasks has been parallelised. In addition, it is shown that again, sort functions have no dependencies, meanwhile merge tasks always depend on 2 tasks, exactly, merge tasks need the previous sorted or merged data (left and right, in that case with the form *&data[ALn / BL]*). In terms of granularity, one can still appreciate a difference between tasks.

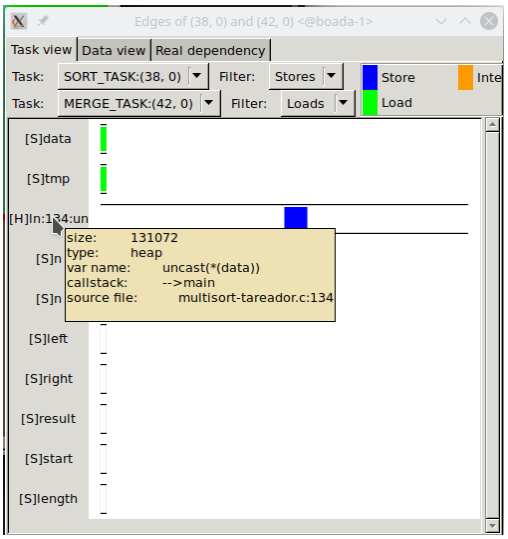

Figure 2.2.3 Tree strategy data dependency

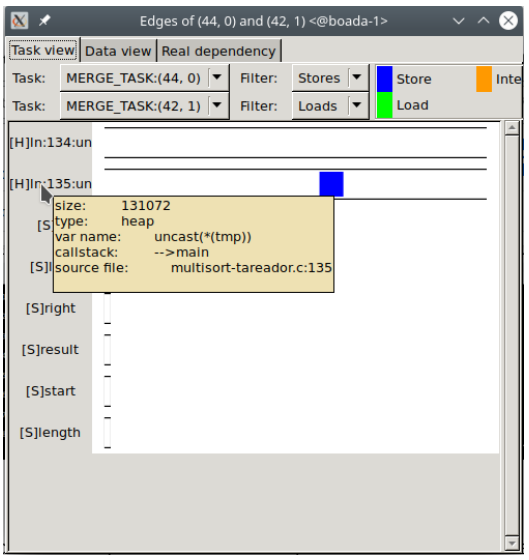

Figure 2.2.3 Tree strategy tmp dependency

As we did on leaf strategy, we analysed what variables were causing those dependencies. In fact, we had the same variables creating dependencies but now every task had just 1 or 2 dependencies.
Compared with the leaf strategy, tree strategy has a better dependency structure, but the granularity of the tasks stills the same. Also as we saw the structure of dependencies have been changed a little and that will affect the performance of the strategies as we will see in the next section.

2.3 Simulation with 16 processors

After the first analysis of the program with TDG and dependencies taken into account, we moved on to simulate an execution with 16 processors and with each of the strategies.

2.3.1 Leaf strategy simulation

First, we simulated the execution with 16 processors for the leaf strategy program. As seen here below, the sort task behaved as shown. We noted that to inspect deeply the first image we needed to explore the beginning and the ending of the same.

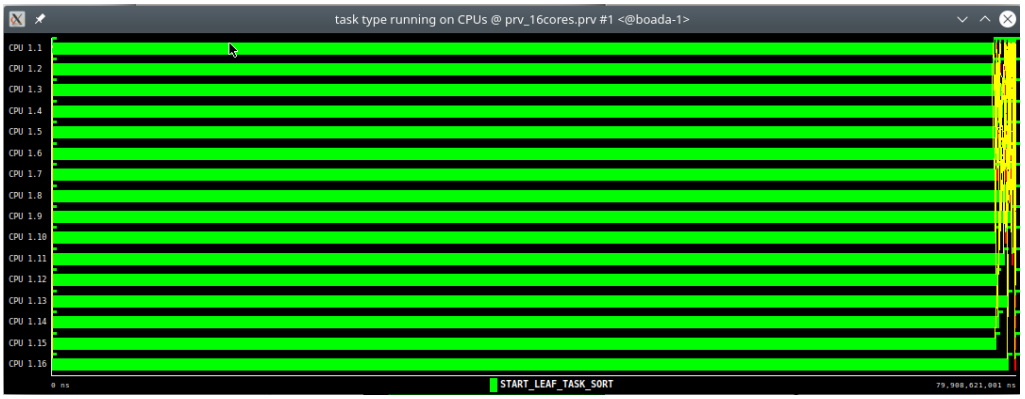

Figure 2.3.1.1 Sort task type running with 16 threads

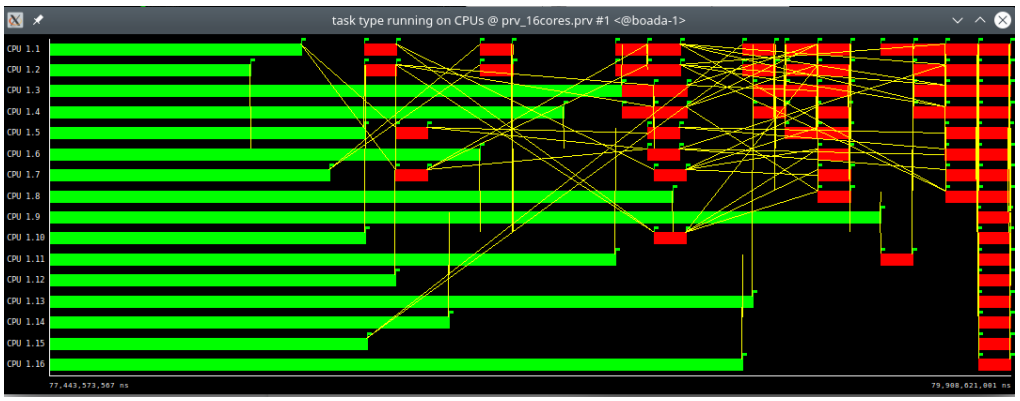

Figure 2.3.1.2 Sort task type running with zoom at the end and 16 threads

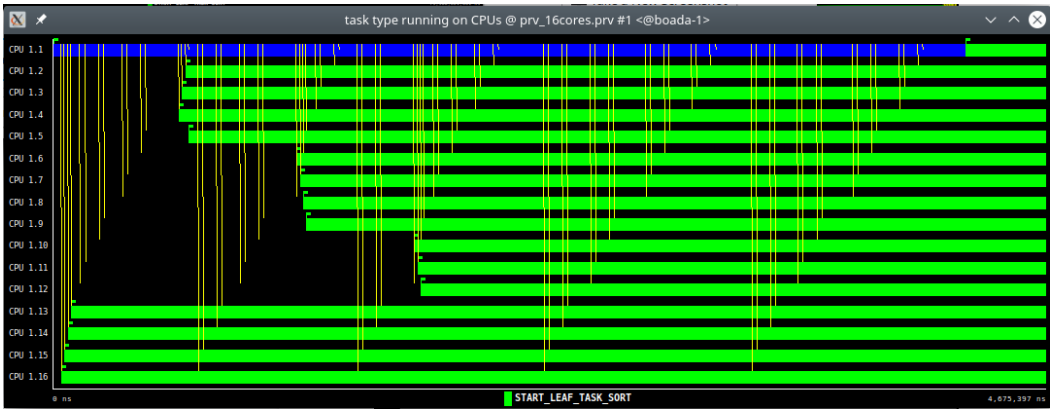

Figure 2.3.1.3 Sort task type running with zoom at the beginning and with 16 threads (thread 1 main task (creator))

If we look into figure 2.3.1.3 it is seen the creation of tasks with thread 1 as the responsible. With leaf strategy, the creation of tasks could not be so parallelised as each task is created at each call of *basicsort* or *basicmerge*. Moreover, in figure 2.3.1.2 it is shown that merge tasks had dependencies on previous sort or previous merge tasks.

2.3.2 Tree strategy simulation

After the simulation with leaf strategy code, we executed the tree strategy one in order to compare the behaviours.

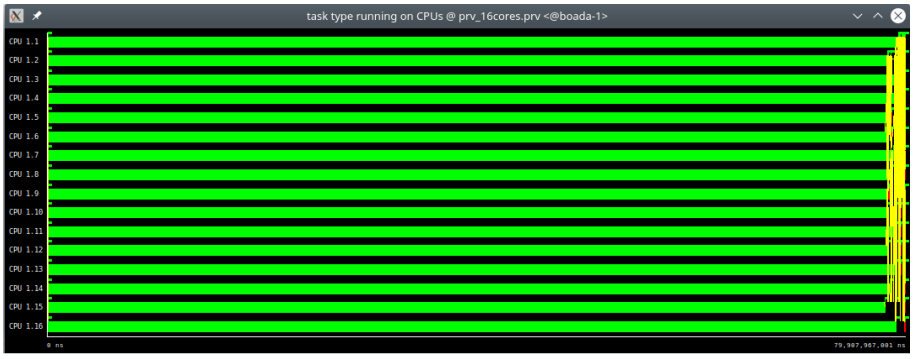

Figure 2.3.2.1 Sort task type running with 16 threads

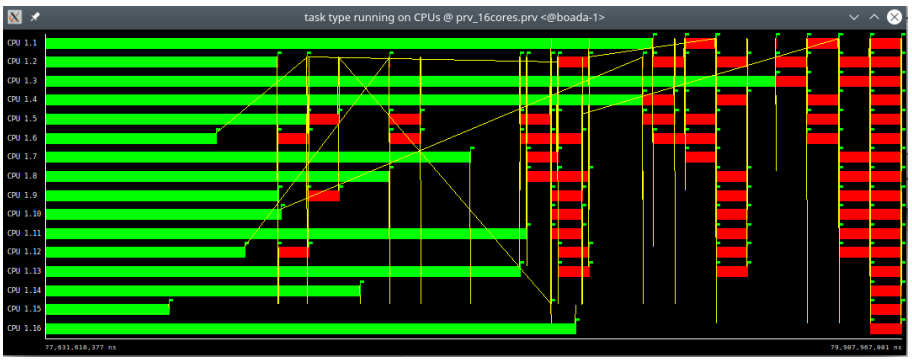

Figure 2.3.2.2 Sort task type running with zoom at the end and 16 threads

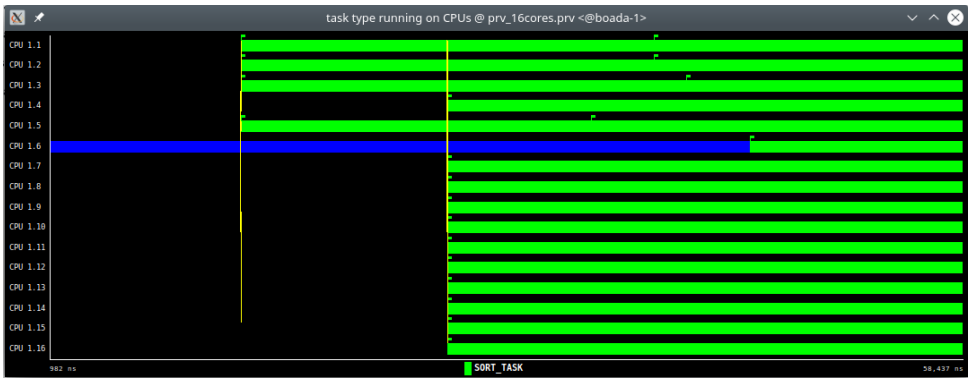

Figure 2.3.2.3 Sort task type running with zoom at the beginning and with 16 threads (thread 6 main task (creator))

In these figures it is shown a higher grade of parallelisation in the creation than in the previous one (see figure 2.3.2.3). Also, as we have observed on leaf strategy, the sort tasks are not creating dependencies while the merge ones are dependent on sort or merge ones.

3
Parallelisation and performance analysis with tasks

In the second session of this laboratory assignment we studied the parallelisation and the performance of the code with OpenMP clauses.

3.1 Leaf strategy with OpenMP

In order to do that study we firstly modified the code with OpenMP clauses. Here below could be seen the code with the respective clauses.

```
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);

        #pragma omp taskwait

    } else {
        // Base case
        #pragma omp task
        {
            basicsort(n, data);
        }
    }
}
```

Figure 3.1.1 Leaf strategy code with OpenMP

In leaf strategy, as we have said in the previous section, the tasks should be created in the base case. To do that, the `#pragma omp task` has been added above the *basicmerge* function and *basicsort* one. To avoid unsynchronization between tasks created, a `#pragma omp taskwait` has been added before the first merge function starts (it needs slices of vector sorted to do merge). Then after the first and second merge functions, in order to do the third merge depending on the two previous ones, we need another `#pragma omp taskwait` to synchronise them. Finally to make the program wait until all merges has been completed, a final clause `#pragma omp taskwait` was needed at the end.

After all these modifications, we executed the program to check if it was working correctly (see figure 3.1.2).

```
par4214@boada-1:~/lab4$ cat submit-omp.sh.o181747
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_16_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=16
********************************************************************************
Initialization time in seconds: 0.854715
Multisort execution time: 2.185420
Check sorted data execution time: 0.019669
Multisort program finished
********************************************************************************
```

Figure 3.1.2 Execution of leaf strategy with OpenMP clauses

In figure 3.1.2 is shown the execution with 16 threads, N equal to 32768, and MIN_SORT_SIZE and MIN_MERGE_SIZE both equal to 1024. The program was correctly executed as there weren't any error messages on the console. It is also seen that the time of initialisation was 0.85 seconds and the multisort execution time (without counting initialization) of multisort was around 2.19 seconds.

Comparing this execution to the sequential one that lasted 6.45 seconds, the program undoubtedly had a better performance in time, but we needed to support this first impression with the study of scalability.

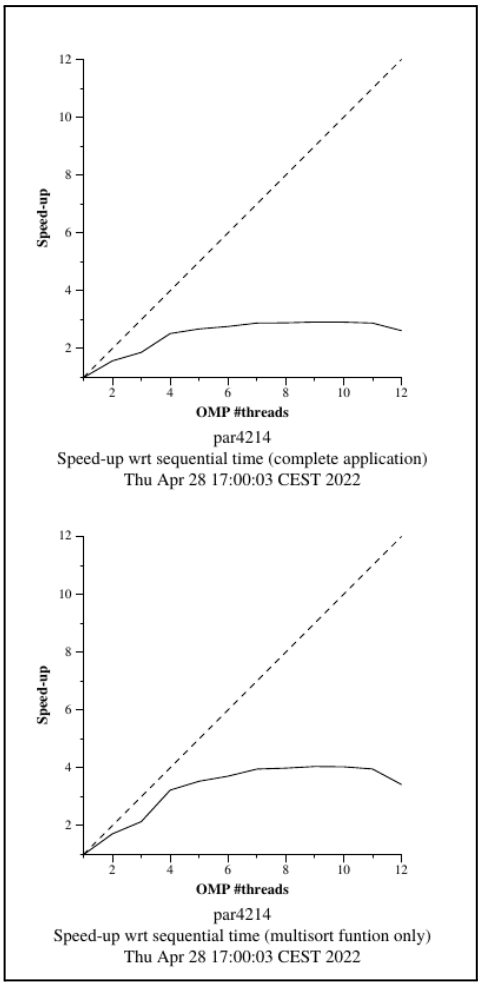

Figure 3.1.3 Strong scalability of leaf strategy

As we see, the performance has been improved in time execution but the scalability of our parallelised program is not good at all. In fact, with a number of threads equal to 10 and bigger, the program reduced the speed-up and the application was slower causing a worse performance than some executions with less threads. We can conclude that the synchronisation of our program causes a big overhead. To see how the program behaved with more accuracy, we have used the *Paraver* tool.

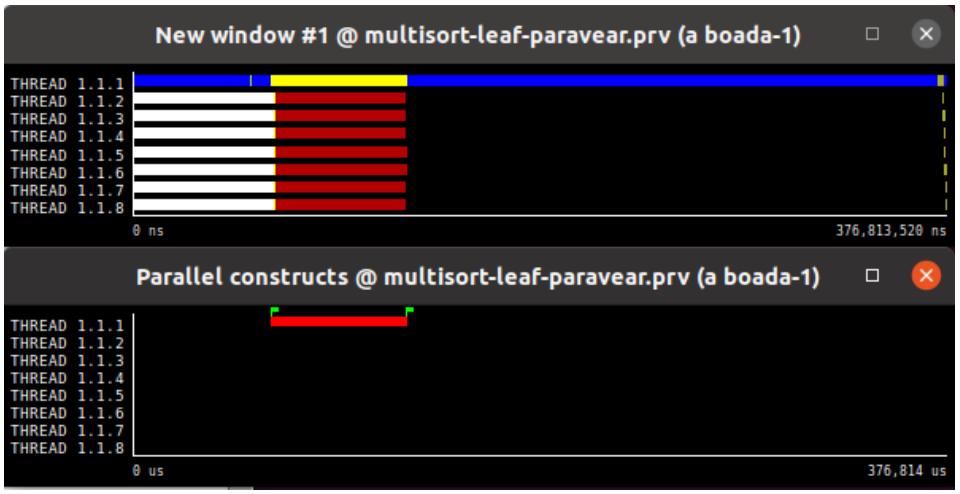

Figure 3.1.4 Parallel construct of leaf strategy with 8 processors

First, we executed the script `submit-extrae.sh` with 8 processors and we saw that the scalability and the parallelisation of the program was not good at all due to initialization time and the way tasks were created. Also, on figure 3.1.4, we can see that the main thread, or the thread that was creating the tasks is the number 1.

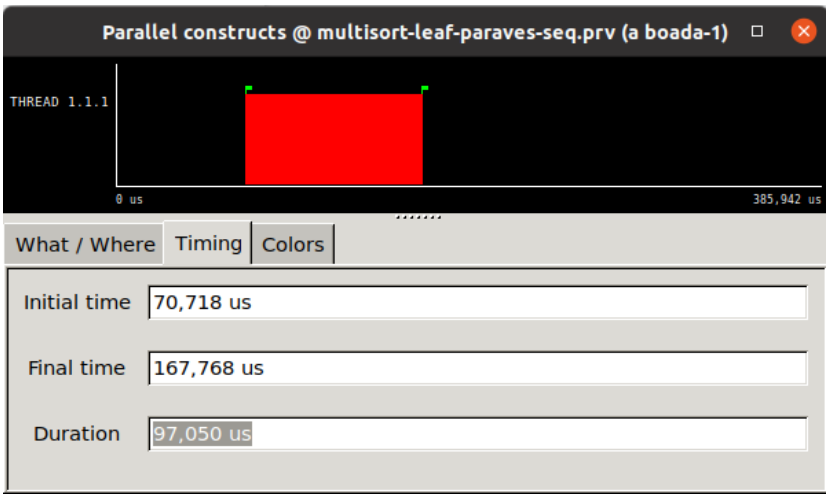

Figure 3.1.5.2 T1 time execution and Tpar time parallelizable on Paravear with 1 processor

With all this information we can see why our program is not scaling in a good way. In addition, it is seen that the worksharing of our tasks is well balanced (see figure 3.1.6). Another thing that can be deduced from the images is that when our program goes further we have some threads that are not useful because of the dependencies of merge calls. However, the sort calls get well

parallelised and there isn't a thread that gets a bigger proportion of explicit tasks to execute than others (unless the main thread, the number one, that is: initialising all the data, creating the tasks and executing the first level recursion tasks 7 + the main task).

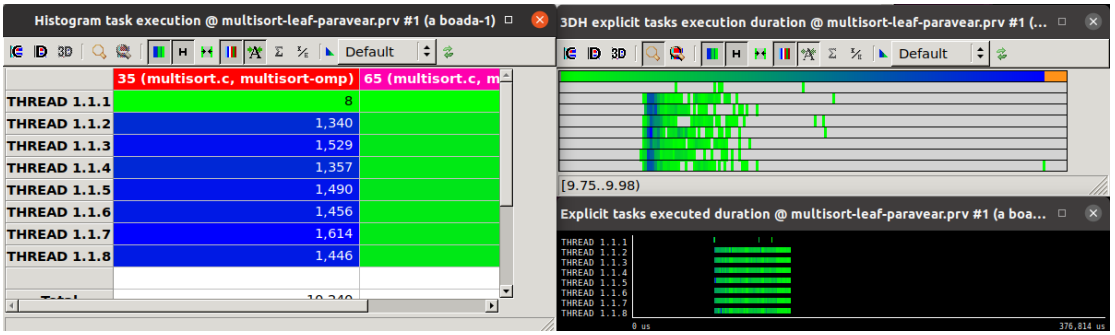

Figure 3.1.6 Histograms of explicit task executions

3.2 Tree strategy with OpenMP

In this section we have seen how tree strategy performance and scalability behaved, and we have compared the main results with leaf strategy. To do that, we initially modified the code to implement the tree strategy in OpenMP, the result code is shown in the figure below.

```
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);
    }
}
```

Figure 3.2.1 Code of tree strategy in OpenMP

As seen, we have implemented the task structure of the tree strategy and the synchronisation in the way to preserve the correct data as we previously saw with Tareador. In this case, tasks were created for every sort and merge recursive calls with `#pragma omp task` clause until the MIN_SORT_SIZE is reached. For the synchronisation we added a `#pragma omp taskwait`

clause as the same way we did on leaf strategy, after multisort calls, after the first and second merge functions and at the end (after the third merge invocation) to prevent the last thread finishing after the program does (it could happen in an extreme situation). After all those modifications, we executed the program with 16 threads to see if it was still correct and sorted the vector well (see figure 3.2.2).

```
par4214@boada-1:~/lab4$ cat submit-omp.sh.o183926
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_16_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=16
************************************************************************************************
Initialization time in seconds: 0.856397
Multisort execution time: 0.591211
Check sorted data execution time: 0.015660
Multisort program finished
************************************************************************************************
```

Figure 3.2.2 Execution of tree strategy with 16 threads

In the previous figure, it is shown that the time execution of multisort is improved in relation to the sequential program (it changed from 6.45 seconds to 0.59, without taking into account initialization). To see if this behaviour had the expected scalability, we executed the script to generate the following figure.

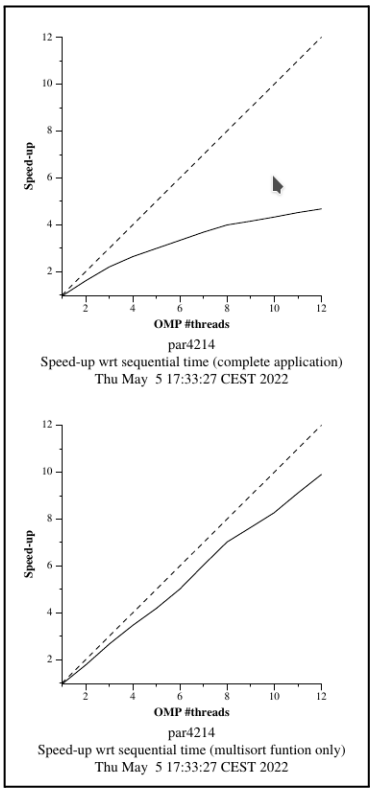

Figure 3.2.3 Strong scalability of tree strategy

On the previous figure it is shown the strong scalability of the program with tree strategy. As seen, even though the performance is better than for leaf strategy, the speedup of the program is not as good as a programmer would like. On the second graph of figure 3.2.3 it is shown the

strong scalability only for the multisort function; as we can see this part of the program is almost perfectly parallelised (embarrassingly parallel) and the only thing that is slowing down the global performance is the initialization and synchronisation of the merge to wait the multisort functions.

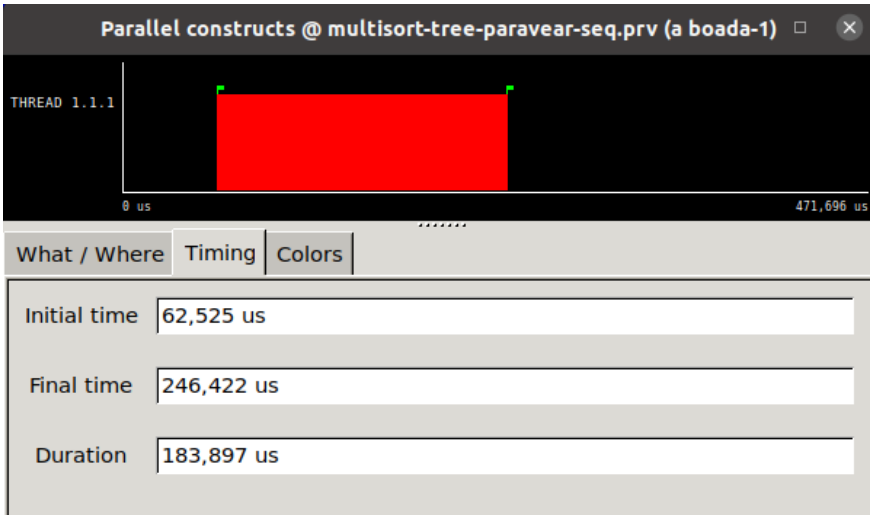

Figure 3.2.4 Parallelizable part of tree program and execution scheduling with 1 processor (to calculate T1 and Tpar)

Then, we moved on to study the parallelisation of tree strategy with the *Paraver* tool. With the first figures we can see again how the parallelizable part of the program is still very low, it is better than leaf strategy but remains very low.

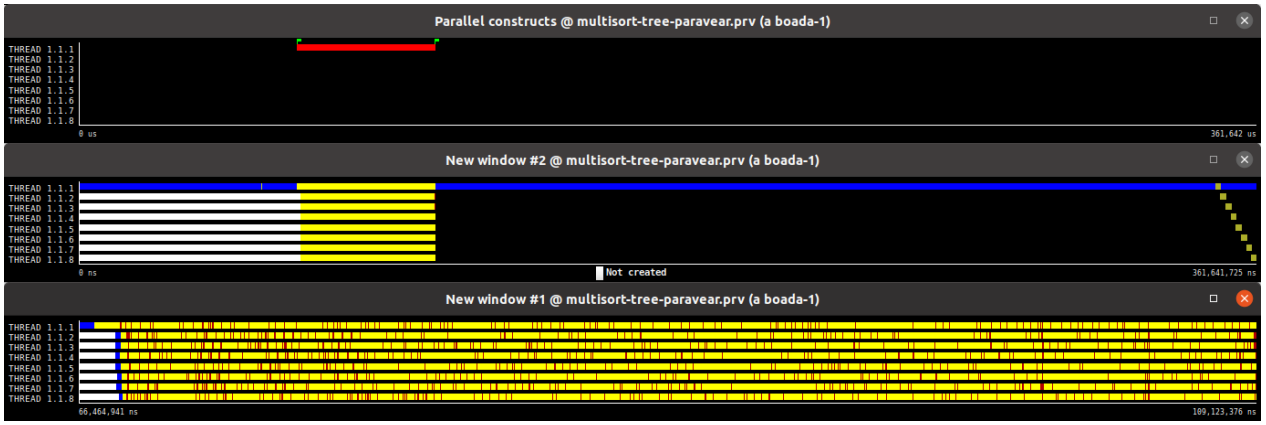

Figure 3.2.5 Parallel execution of tree strategy with Paraver and 8 processors

As seen on figure 3.2.5 the parallelizable part is wasting a lot of time scheduling (yellow part of the code) because of the structure of the tree strategy.

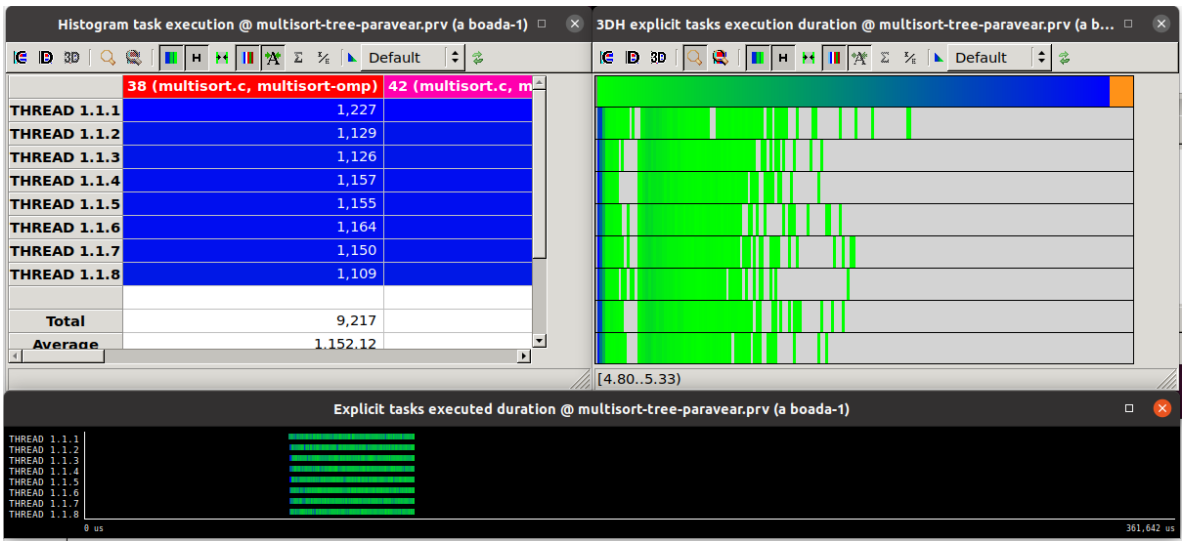

Figure 3.2.6 Explicit tasks executed per thread and time execution of explicit tasks

The explicit tasks executed per thread in tree strategy remain also well balanced (as we also previously saw on leaf one). We can see with the *explicit tasks execution duration* histogram, how the cost of every task is also well balanced in terms of time execution. The same thing happens as we saw in leaf strategy, when the merge calls are running some threads don't get explicits tasks to execute, that's because of the dependencies of merge calls again (in that case, the last merge call was executed by the thread number 5). The main difference that we got with tree strategy is that the main thread, the one that creates all tasks and executes the first level recursion, is now working and executing the same number of tasks as all the other threads.

3.3 Controlling task granularities: cut-off mechanism

Up to this point, we have implemented tree strategy without a mechanism to control the number of leaves. So following this part, here below is shown the code related to control the task granularity with a mechanism named cut-off.
To do that, we declare a new variable depth that was incremented in each recursive call of multisort and merge functions. After that, when the code it's not in the base case, we checked if it corresponds to a final task, if not then the code was executed creating new tasks otherwise, the calls were invoked without new tasks creations. See figure 3.3.1 to see the full code.

```
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);

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

Figure 3.3.1 Code of tree strategy with cut-off mechanism in OpenMP

After implementing the related modifications to the cut-off mechanism, we checked that the execution of the program was still correct. Here below it's shown the execution and the message of the console. The times are similar (at least with a cut-off equal to 4) to the previous implementation with tree strategy but without a cut-off mechanism.

```
par4214@boada-1:~/lab4$ cat submit-omp.sh.o184952
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_16_4_boada-2.times.txt
::::::::::::::
********************************************************************************
*
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=4
Number of threads in OpenMP:          OMP_NUM_THREADS=16
********************************************************************************
*
Initialization time in seconds: 0.855546
Multisort execution time: 0.557905
Check sorted data execution time: 0.015464
Multisort program finished
********************************************************************************
```

Figure 3.3.2 Execution of tree strategy with cut-off mechanism and with 16 threads

As we can see on the figure above, the execution runned correctly. Next, we moved on to analyse the program modified with the *Paraver* tool.
To see how the program behaved with a cut-off level equal to 0 (parameter -c), we have looked at the following two figures.

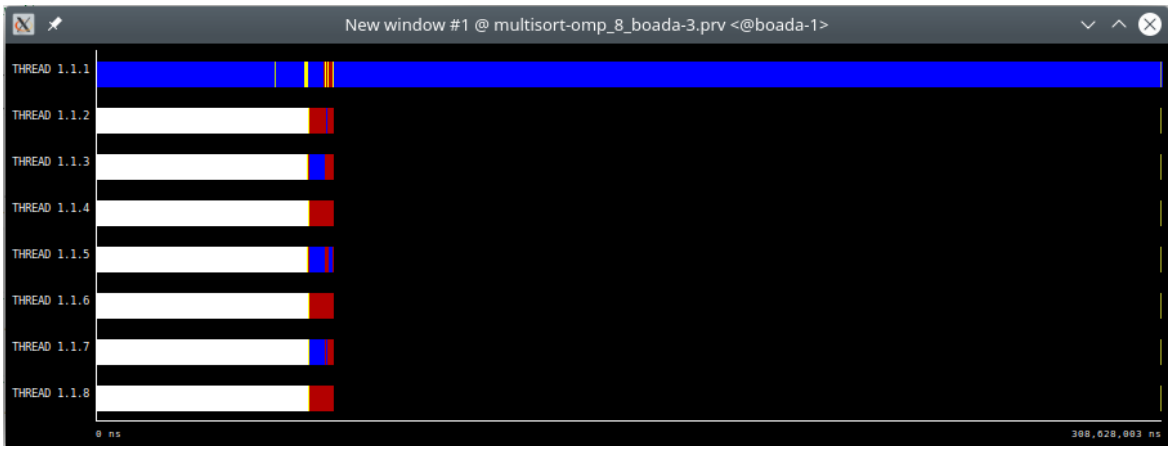

Figure 3.3.3 Execution with -C 0 parameter

Figure 3.3.4 Explicit task function execution with -C 0 parameter

On the previous pictures it is shown the execution behaviour. In the first figure, we can see how the first thread is the main creating and running the program. Threads 1, 3, 5 and 7 are the liables of the execution of sort functions. This is the reason why they have blue colour at the beginning. Then thread 2 and 7 are the ones doing the first two merge calls, and finally 5 doing the last merge (they spend a lot of time doing synchronisation).

In the second image, it is shown the explicit task function execution. Again, it is shown how threads 1,3,5 and 7 are the liables on sort functions and how 2 and 7 wait until they are finished. At the end, we have the fifth thread waiting for 2 and 7 to finish their merge functions. As the level of cut-off indicated in this case was 0, there are 7 tasks created in total (4 for sort functions and 3 for merge ones).

Next we execute again with the *Paraver* tool but with the cut-off level set to 1 in order to compare with the previous results with 0 as the cut-off level.

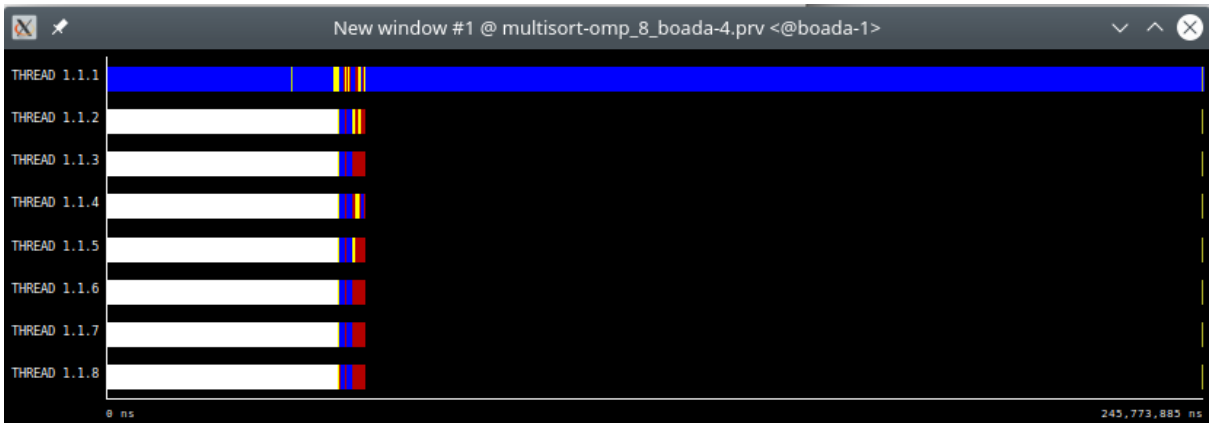

Figure 3.3.5 Execution with -C 1 parameter

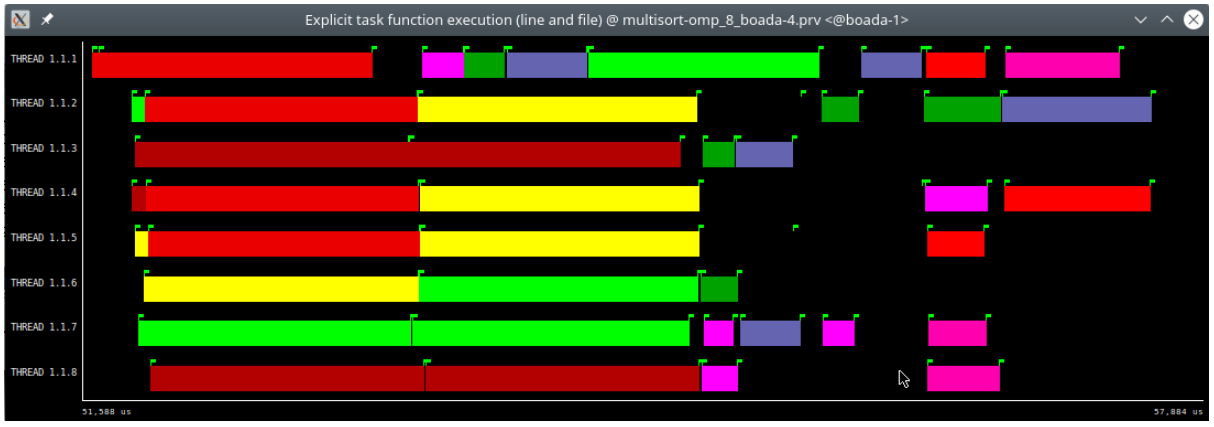

Figure 3.3.6 Explicit task function execution with -C 1 parameter

As seen on the previous figures, incrementing the cut-off from 0 to 1 has added a level of recursivity. Now, the number of sort functions is 16 and the merge ones are 21 (see figure 3.3.6). Looking at this image we can identificate the sort ones as the longer ones and without any dependence meanwhile the merge ones had to wait their previous sort or merge to finish in order to start their execution.
Following these results, one can conclude that incrementing the cut-off implies the addition of another level of recursivity. It would always happen unless the level of recursivity reached a point in which the size of the vector was smaller than the MIN_SIZE selected.

Moving to the next part, we had to explore values for the cut–off level depending on the number of processors used. To do that, we have execute the command line `#sbatch ./submit-cutoff-omp.sh 16`. As seen on the figure below the optimal point is reached

with a cutoff equal to 6, which means, executing with the parameter -c equal to 6. The internal parameter of the script were:

- Length of the vector equal to 32768 elements: -n 32768.
- Sort size (maximum length of the vector to order) equal to 128: -s 128.
- Merge size (maximum length of the vector to merge): -m 128.

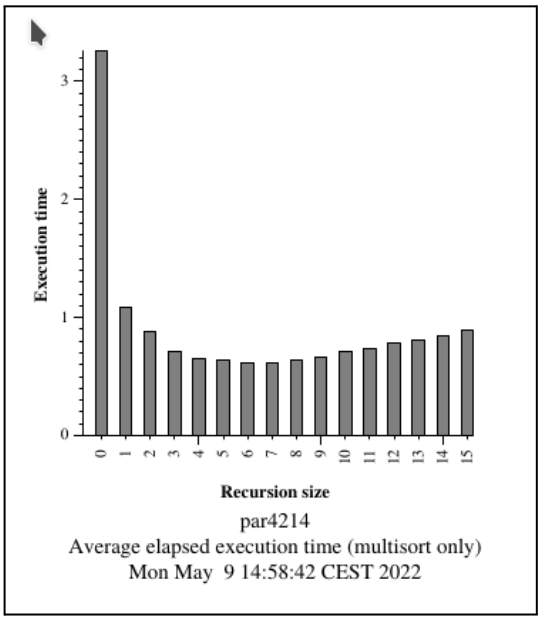

Figure 3.3.7 Relation of execution time and cut-off levels

On plot 3.3.7 it's seen that moving from cut-levels between 4 and 9 there is no such a big difference on the execution time; in fact, the execution time for levels equals to 6 and 7 the execution time is the same (and they are the optimal points). The reason why this could happen is because with the same number of threads, adding more levels of recursivity and therefore, more tasks hadn't a positive impact on the performance. Actually, at the end of the graph (cut-levels from 11 and bigger) one could see the time execution worsen. With small cut-off levels it is obvious that the performance is worse because they are not exploiting the parallelism at its maximum.

On the following point we were asked to compare the strong scalability when the optimal and maximum point of cut-off are set. Firstly, we executed the script with the maximum value (cut-off equal to 16), and sort_size and merge_size equal to 128. Here below are shown the plots generated.

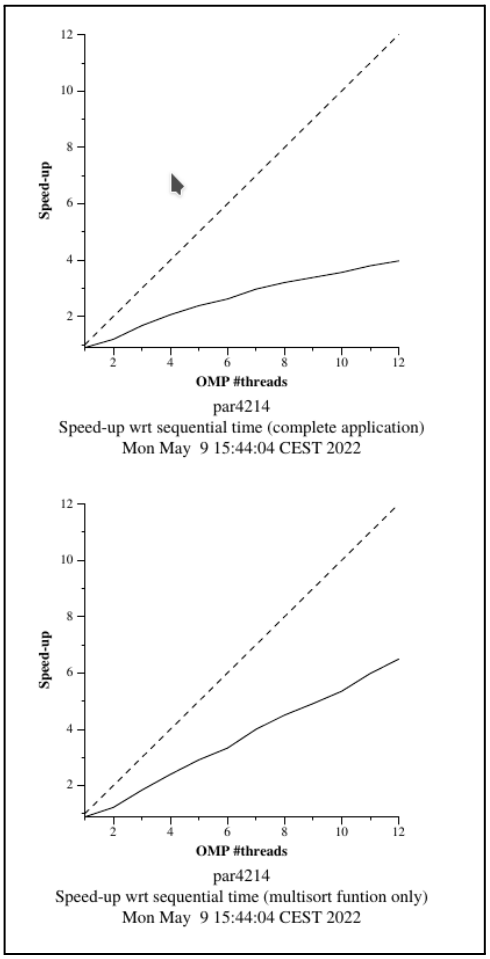

Figure 3.3.8 Strong scalability with cut-off equal to 16

Then, after this first submission, we modified the script in order to generate the plots related to the optimal values. Here below the adapted script.

```
size=32768
sort_size=128   # CANVI -> valor demanat
merge_size=128  # CANVI -> valor demanat
cutoff=6        # CANVI -> valor optim

np_NMIN=1
np_NMAX=12
N=3

# Make sure that all binaries exist
make $SEQ
make $PROG

HOST=$(echo $HOSTNAME | cut -f 1 -d'.')

out=$PROG-strong_${HOST}.txt»   # File where you save the execution results
aux=./tmp.txt     » # archivo auxiliar

outputpath1=./speedup1.txt
outputpath2=./speedup2.txt
outputpath3=./elapsed.txt
rm -rf $outputpath1 2> /dev/null
rm -rf $outputpath2 2> /dev/null
rm -rf $outputpath3 2> /dev/null

echo -n Executing $SEQ sequentially > $out
elapsed=0  # Acumulacion del elapsed time de las N ejecuciones del programa
partial=0  # Acumulacion del multisort time de las N ejecuciones del programa

i=0        # Variable contador de repeticiones
while (test $i -lt $N)
do
    echo $'\n' >> $out
    /usr/bin/time --format=%e ./$SEQ -n $size -s $sort_size -m $merge_size >> $out 2>$aux
```

Figure 3.3.9 Modifications of the script with cut-off equal to 6

Finally we executed to generate the plots related to the optimal cut-off value. See figure 3.3.10.

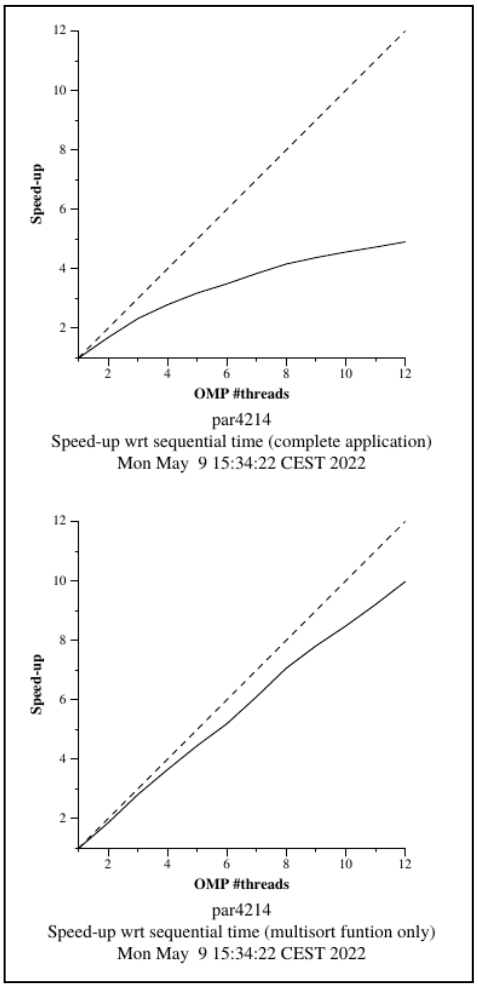

Figure 3.3.10 Strong scalability with cut-off equal to 6

Comparing both images, it is clearly seen that the cut-off value plays an important role on the performance on the performance of multisort function. As the multisort function is embarrassingly parallel, setting the cut-off to the optimal made this function to exploit all the parallelism with a scalability near to the ideal one. When adding more levels of recursivity and therefore, more tasks, the overhead of creation and synchronisation worsen the performance (as shown on figure 3.3.8 with cut-off equal to 16). In addition, looking at the plots of the complete application, the speedup it's slightly better on the performance with cut-off equal to 6 than in the one with cut-off equal to 16, but this difference it's not as significant as the difference shown on multisort function.

4
Parallelisation and performance analysis with task dependencies

4.1 Tree strategy with depend OpenMP clauses

In this section we have seen another way to implement task dependencies with OpenMP. As we can see in the code below (see figure 4.1.1), we had to determine which variables have possible dependencies to the other tasks, typing which variables are the output of the task and which ones are the input. With that method OpenMP can automatically take care of the dependencies of every task and it is a simpler and natural way for the programmer to manage all the dependencies of the code that can be parallelized. Also, we could remove some *taskwait* clauses of the code but there are other ones that can't be removed, as for example, the last *taskwait* clause of every recursion level that have to wait for all the tasks of the same level.

```
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);

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

Figure 4.1.1 Code of tree strategy with cut-off mechanism and depend clauses

To verify the correctness of the code, we executed the command line `#sbatch ./submit-omp.sh multisort-omp 16 16`. As we can see here below the vector was well sorted. The time execution is also similar to previous executions with tree strategy.

```
par4214@boada-1:~/lab4$ cat submit-omp.sh.o186355
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_16_16_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=16
************************************************************************************
Initialization time in seconds: 0.856289
Multisort execution time: 0.551234
Check sorted data execution time: 0.015415
Multisort program finished
************************************************************************************
```

Figure 4.1.2 Execution program of section 4 to verify the correctness

Following with the same code, we have now executed the script `submit-strong.sh` to see how scalable the code was in this case. As we expected, the scalability of the program didn't change at all compared with the previous one (tree strategy with cut-off mechanism) because the program is doing the same dependency structure. Again the scalability of the application in general is not as near to the ideal as one could desire but it is rounding the 4 of speedup from a number of threads equal to 7 and bigger. Also, multisort function as expected, is almost perfectly parallelised.

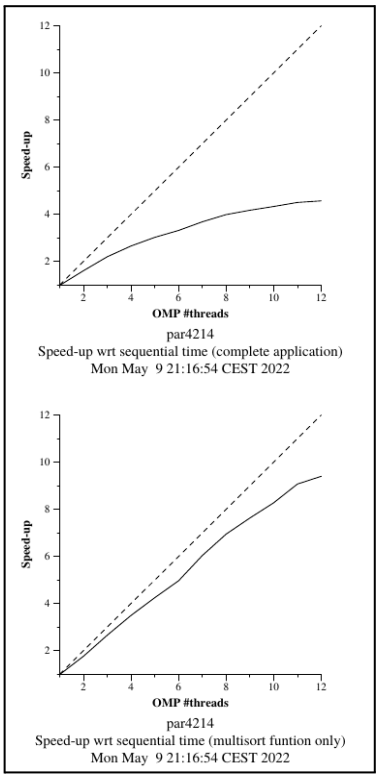

Figure 4.1.3 Strong scalability behaviour of tree strategy with cut-off mechanism and depend clauses

To verify our first impressions, we analysed this code with the help of the *Paraver* tool. So, we executed the script with the command line `submit-extrae.sh`

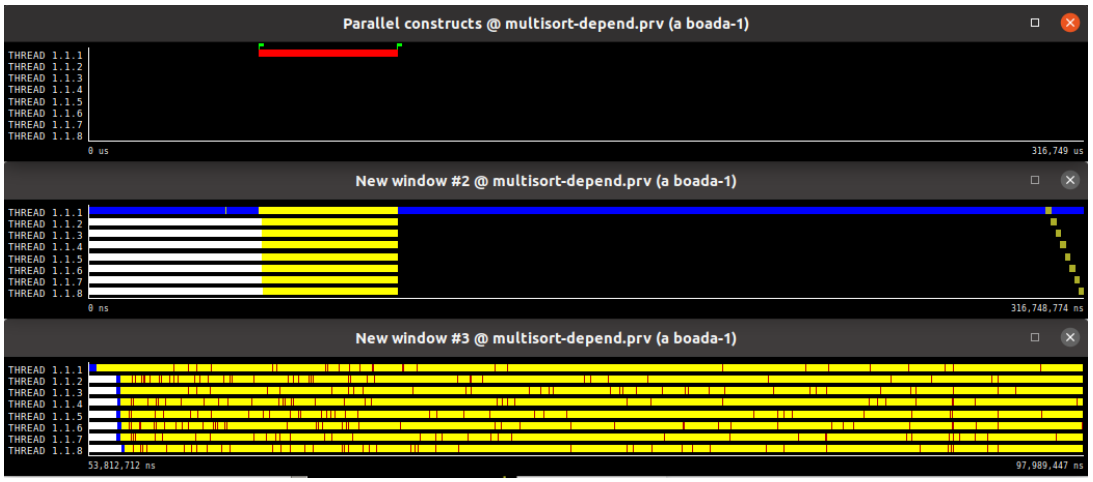

Figure 4.1.4 Strong scalability behaviour of tree strategy with cut-off mechanism and depend clauses

As shown on the image above, the general behaviour of this implementation is similar to the one of tree strategy. The time scheduling is still high (as the implementation follows a tree structure and there are a lot of recursive calls creating tasks). Again, the parallelizable part remains low. All these characteristics make sense as we really have not changed the structure of the program, only the clauses that create them. So in fact, the internal shape is almost the same.

Figure 4.1.5 Explicit task function execution:
above with depend clauses below without depend clauses and with taskwait

To verify if our previous conclusions were correct, we tried to compare the explicit task function execution diagrams of the program with depend clauses and the previous one with taskwait. As both of the executions have a duration of 1.000us, the scale is the same, and it is seen that the dependencies create lower time lapses to wait in the case of depend clauses; so in fact there are some little improvements with this implementation. After all, we can not assure that the improvement is real, because the executions are not in the same exact time (almost the same but not exactly), but we thought that it could be a general behaviour.

Finally we wanted to check if the granularity of the tasks was similar to previous executions and were well balanced. So we generated the histogram related. See here below the image 4.1.6.

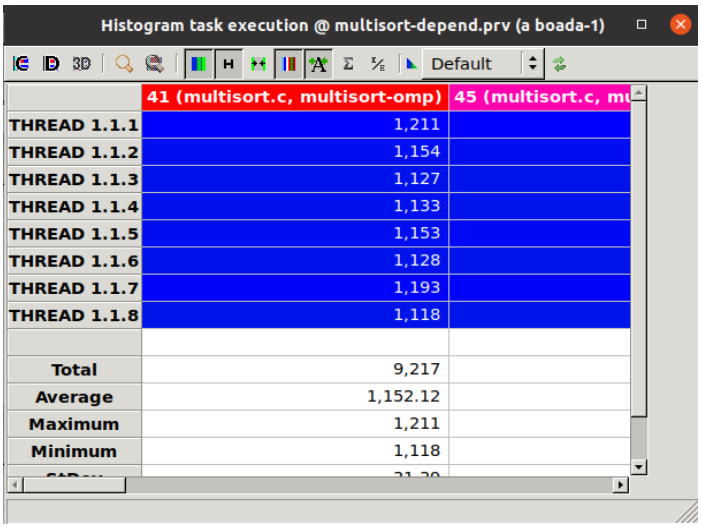

Figure 4.1.6 Histogram of explicit task execution

As shown, the threads were doing a similar proportion of tasks, so the work was well balanced. In addition, if one compares this histogram with the one of the figure 3.2.6 (implementation of tree strategy with OpenMP), the proportions are similar and the average is exactly the same. The reason why this happens is because both implementations create the same tasks and in the same order, the only thing that has changed is the clauses applied on the code and its way to synchronise tasks.

5 Optional

5.1 Optional 1

In order to explore the performance of scalability when the maximum cores were set to 24, we firstly changed the script `submit-strong.sh` as shown on figure 5.1.1.

```
np_NMIN=1
np_NMAX=24      # CANVI -> valor demanat
N=3
```

Figure 5.1.1 Script modified to set np_NMAX equal to 24

After we adapted the script, we submitted the work to generate the strong scalability plots of tree strategy related with a cut-off equal to 16 and 24 threads.

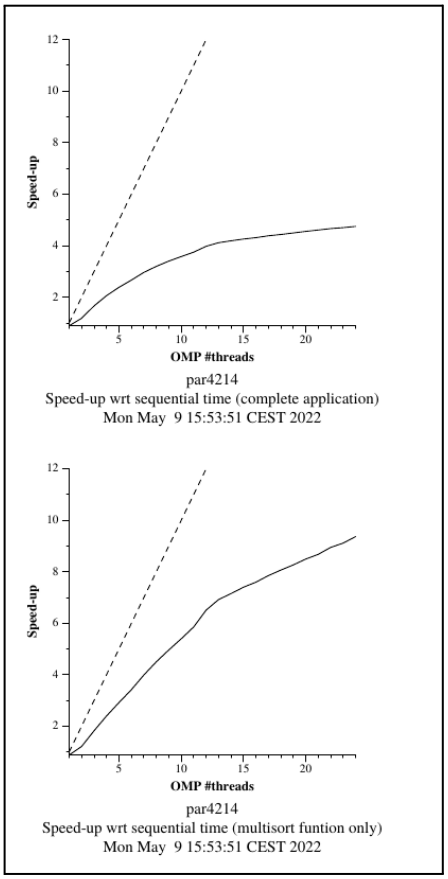

Figure 5.1.2 Strong scalability plots with np_NMAX equal to 24 and cut-off level equal to 16

We also thought that it was a good idea to have the better performance plots, so we submitted the work with 24 threads and with cut-off level equal to 6 (in order to see how it would behave optimally, see figure 3.3.7 to see the optimum value of cut-off).

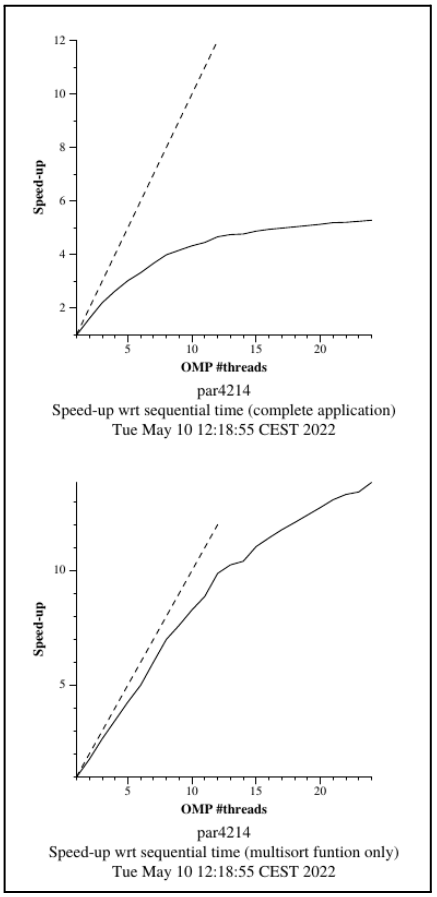

Figure 5.1.3 Strong scalability plots with np_NMAX equal to 24 and cut-off level equal to 6

Seeing these plots, they are clearly shown that the performance keeps improving when the number of threads is set to 24. The reason why this happens is because despite that from boada-1 to boada-4 there are only 12 physical cores, they all have 2 threads per core. See table below obtained on laboratory assignment 1:

	boada-1 **to** boada-4
Number of sockets per node	2
Number of cores per socket	6
Number of threads per core	2
Maximum core frequency	2.40 Ghz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	256 KB
Last-level cache size (per-socket)	128 MB
Main memory size (per socket)	12 GB
Main memory size (per node)	23 GB

Table 5.1.1 Summarised information from boada-1 to boada-4

So as boada can be executing 24 tasks at the same time, the performance of the program in time execution terms can be improved if we modify the number of threads to 24.
We have also generated the figures below with the Parever tool in order to see how the work was shared between the 24 threads.

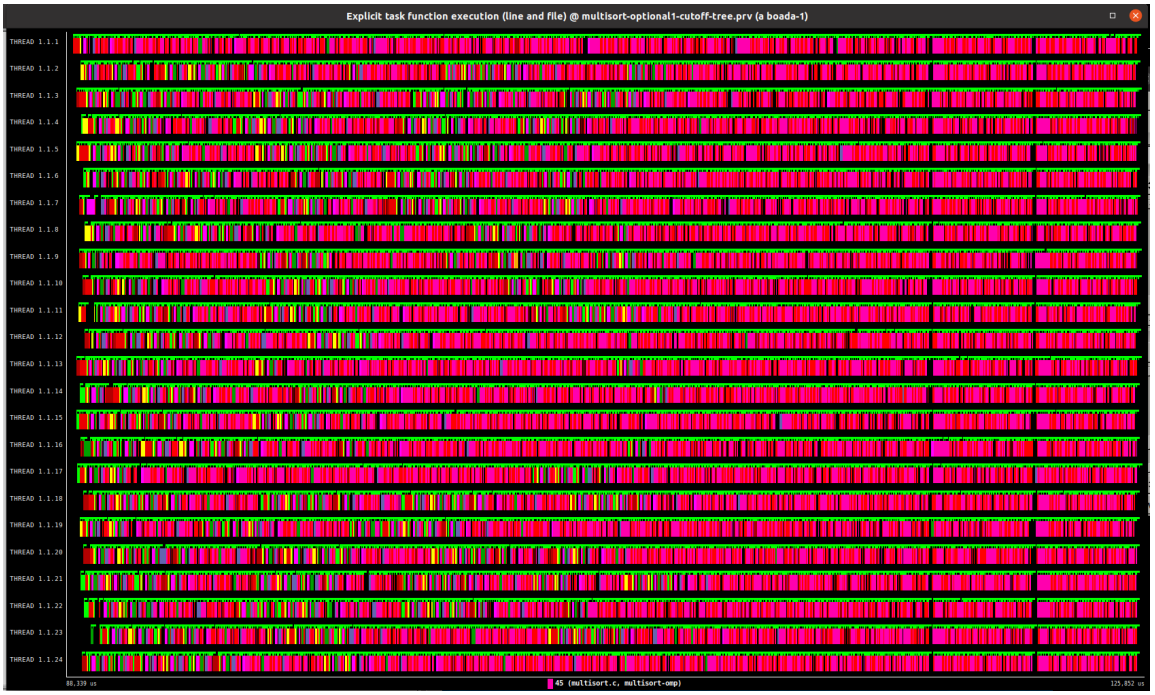

Figure 5.1.4 Explicit task function execution

Figure 5.1.5 Explicit task function execution with zoom in

Taking into account figures 5.1.4, 5.1.5 it is shown that the work was well balanced among the 24 threads and that is another reason why the improvement on time execution could happen. The figures show how the explicit tasks are executed during the execution (the largest are the sort calls and the shortests the merge ones) and how the dependencies are working.

5.2 Optional 2

In this second optional part we were asked to parallelise both functions related to the initialization of the data and tmp vectors.
In order to accomplish this, we firstly modified the code. Here below the image of the code modified. We added the clause for before the loop in order to parallelise it.

```
static void initialize(long length, T data[length]) {
    long i;

    // initialize data[0] to avoid if clause inside of the loop (optimitzation)
    data[0] = rand();

    // we parallelise the other part of the vector
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 1 /* initialize i to 1 */; i < length; i++) {
            data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
    }

}

static void clear(long length, T data[length]) {
    long i;
    // we parallelise the vector initialization
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < length; i++) {
            data[i] = 0;
        }
    }
}
```

Figure 5.2.1 Code of optional 2: parallelising initialization

After modifying the code, we checked the correctness of the same by executing it. As shown on the figure below we have decremented the initialization time from 0.85 to 0.07 seconds.

```
par4214@boada-1:~/lab4$ cat submit-omp.sh.o186404
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_16_boada-2.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=16
********************************************************************************
Initialization time in seconds: 0.074452
Multisort execution time: 0.542282
Check sorted data execution time: 0.018186
Multisort program finished
********************************************************************************
```

Figure 5.2.2 Execution program of optional 2

Then, we explored the scalability of the new implementation executing the script with the command line `submit-strong.sh`. As shown in the image below, the performance of the application has been improved considerably as we have modified the initialization and by this, increment the parallelizable time. The program previously was spending around 0.55 seconds on the multisort function and 0.85 on the initialization one, now decremented to 0.07 seconds.

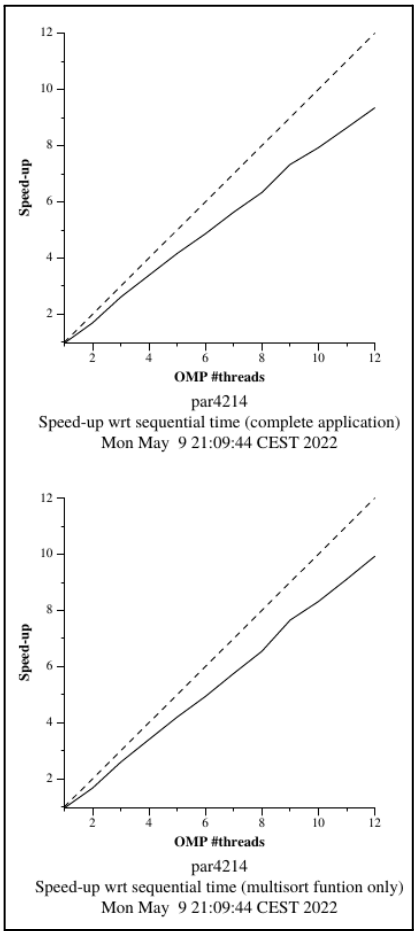

Figure 5.2.4 Strong scalability plots with optional 2 code

6

Conclusions

In this final section we are going to explain the main conclusions we extracted. First of all, we will talk about the analysis with Tareador of both codes, then we will continue with the OpenMP implementations results of the performances and finally we will talk about the optional parts.

In the first class we made the Tareador analysis of leaf and tree strategies, two different ways to parallelise a recursive code. We saw that the dependencies changed depending on how the program was creating the tasks. We also see how to maintain the correctness of the data during the execution of the program studying the dependencies.

Continuing we have implemented the codes with OpenMP and we executed them to see the performance. With the results on hand, we have seen that the different ways to create the tasks have been reflected in the scalability of our programs. On leaf strategy we got a good improvement on time execution but in terms of scalability was much worse than the tree strategy, around 10 processors leaf strategy started to decrease the speed-up. We also added a cut-off mechanism on the tree strategy and observed how it was working. The results of the cut-off changed by a little the scalability performance and allowed us to control the recursivity level and the number of tasks created per execution. Next in this section, we implemented a new way to manage the dependencies of the tasks, an easier way for the programmer to write the code. That change didn't modify at all the performance of our code but we saw a little difference about how the synchronisation was managed (a little bit more compacted).

Finally, we made optional parts to see some interesting changes that we could make to the tree strategy program. The first one was related to stressing the scalability of the program changing the number of threads to 24 and seeing if the speed-up were continuing increasing knowing that boada had just 12 physical cores. As we saw the scalability continued increasing and the reason was that each physical core had 2 threads that could in some way manage to execute up to 24 tasks at the same time. To end up with the second optional part we increased the parallelizable part of the program parallelising the initialization of the data and tmp vectors (that was implemented in a sequential way). Those final results concluded with an improvement of 3% respect on the old version of the φ factor.