Lab 4:
Divide and Conquer parallelism with OpenMP: Sorting

Group 2208

Course 2020/2021 Q1

Arnau Lamiel
Oscar Cañabate

arnau.lamiel@estudiantat.upc.edu
oscar.canabate@estudiantat.upc.edu

Index

1. Introduction

Sorting algorithm is one of the most useful algorithms in programming. In this work, we will focus on the mergesort, that uses a multisort, and the different ways to parallelize it.

Now, we will explain how this algorithm works:

Firstly, the program will create 2 vectors, tmp and data. They are created and initialized for two reasons, the *data* one stores all the elements which gradually its elements are being sorted, and the vector called tmp is used as a local variable between merge functions to store the provisional order of data elements.

Next, the program calls a multisort function and divides the vector recursively to 4 vectors. In his base case it uses another function, basicsort.

Secondly, after the multisort calls finish, starts the merge calls. These algorithms combine 2 subparts of the vector *data* into the vector *tmp*.
There are merge calls in the function, 2 to merge the subvectors 2 by 2 and a final one to combine the final subvectors merged before.

Finally, this vector is iterated by a function which checks if the vector is correct or if it is not, and outputs all the execution parameters.

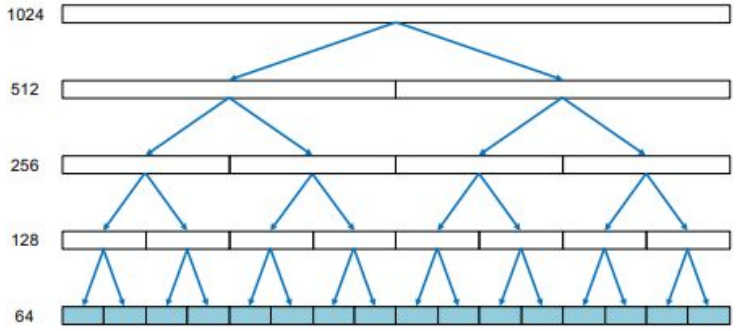


Figure 1: Divide-and-conqueer for N=1024, MIN_SIZE=64

2. Parallelization Strategies

There are 2 main different parallelization strategies (Leaf and Tree), and in this point we will explore it's differences. For it, we are going to use Tareador, one graph for each strategy (this code is available in multisort-tareador.c, we made a global variable named *leaf* and if we want to execute the leaf version, that variable is on 1, 0 otherwise).

On one hand, leaf strategy, the main particularity is, as we can see at the graph in the figure 2 that the parallelization is made in the leafs of the graph. What means that the same task is doing sequentially all the divide processes, and just the calls to basicsort have been parallelized.

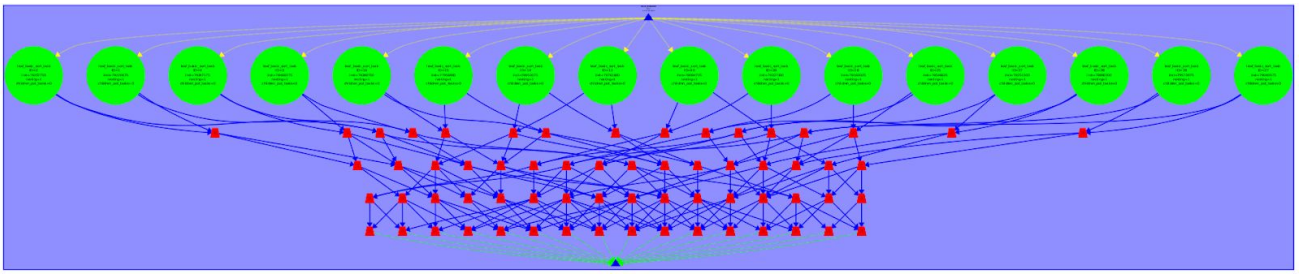


Figura 2: Graph generated by Tareador for leaf strategy

This strategy is improvable, because we are not taking advantage of the parallelization, the main parts of the program are not being parallelized and are made in a sequential way. In our code, as we can see in the figures 3 and 4, we are making a task every time the basicsort and basicmerge are going to execute. So we are creating tasks for each base case.

```
if(leaf) tareador_start_task("leaf basic merge task"); //leaf strategy
    basicmerge(n, left, right, result, start, length);
if(leaf) tareador_end_task("leaf basic merge task"); //leaf strategy
```

Figure 3 : Code in method merge of multisort-tareador.c when we choose leaf strategy

```
if(!leaf)// tareador_end_task("tree merge task"); //tree strategy
} else {
    // Base case
    if(leaf) tareador_start_task("leaf basic sort task"); //leaf strategy
        basicsort(n, data);
    if(leaf) tareador_end_task("leaf basic sort task"); //leaf strategy
}
```

Figure 3 : Code in method multisort of multisort-tareador.c when we choose leaf strategy

On the other hand, we have the other version of the leaf strategy, which uses parallelization all time it is possible. That is the tree strategy.
The main key of this strategy is the contrary of the leaf one. Tree strategy are creating tasks every time a recursive call is done, but in the base case. So this strategy has his workload at the end, not like the leaf one, who has that workload at the beginning, since we don't arrive at the base case.
In Figure 4 we can see the tree strategy graph generated by *Tareador.* As we can see, the first calls are parallelized as a task, then in the next level we have the same task creation, and same for each call. After all this calls, when we arrive at the base case, we do not create specific tasks for this, there are the same tasks that made the previous tasks. At the end, there are the different merge calls, which are parallelized as a tasks because there are not the basic tasks.

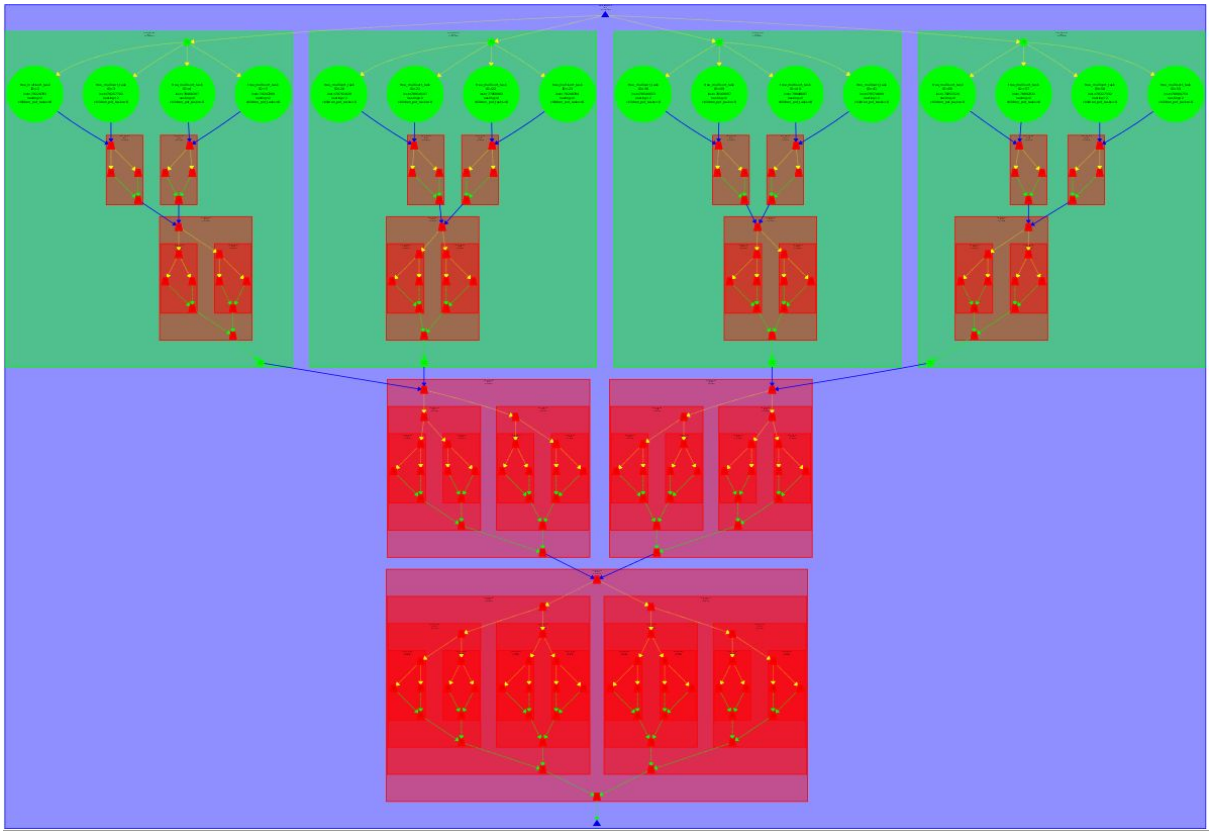


Figure 4: Task graph for tree strategy done by Tareador

We can see how it is done in the next code fragments in the figures 5 and 6, where tasks are created every non-base case calls.

```
// Recursive decomposition
if(!leaf)tareador_start_task("tree merge task"); //tree strategy
    merge(n, left, right, result, start, length/2);
if(!leaf)tareador_end_task("tree merge task"); //tree strategy

if(!leaf)tareador_start_task("tree merge task"); //tree strategy
    merge(n, left, right, result, start + length/2, length/2);
if(!leaf)tareador_end_task("tree merge task"); //tree strategy
```

Figure 5 : Code in method merge of multisort-tareador.c when we choose tree strategy

```
if (n >= MIN_SORT_SIZE*4L) {
    // Recursive decomposition
if(!leaf)tareador_start_task("tree multisort task"); //tree strategy
    multisort(n/4L, &data[0], &tmp[0]);
if(!leaf)tareador_end_task("tree multisort task"); //tree strategy

if(!leaf)tareador_start_task("tree multisort task"); //tree strategy
    multisort(n/4L, &data[n/4L], &tmp[n/4L]);
if(!leaf)tareador_end_task("tree multisort task"); //tree strategy

if(!leaf)tareador_start_task("tree multisort task"); //tree strategy
    multisort(n/4L, &data[n/2L], &tmp[n/2L]);
if(!leaf)tareador_end_task("tree multisort task"); //tree strategy

if(!leaf)tareador_start_task("tree multisort task"); //tree strategy
    multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
if(!leaf)tareador_end_task("tree multisort task"); //tree strategy

if(!leaf)tareador_start_task("tree merge task"); //tree strategy
    merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
if(!leaf)tareador_end_task("tree merge task"); //tree strategy

if(!leaf)tareador_start_task("tree merge task"); //tree strategy
    merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L); //tree strategy
if(!leaf)tareador_end_task("tree merge task"); //tree strategy

if(!leaf)tareador_start_task("tree merge task");
    merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n); //tree strategy
if(!leaf) tareador_end_task("tree merge task"); //tree strategy
```

Figure 6 : Code in method multisort of multisort-tareador.c when we choose tree strategy

Finally, we want to calculate the execution time and speedup of each strategy for a different number of processors each time.

Processors	Execution time(s)	Speed up
1	1263,26	1
2	631,66	1,99
4	315,82	3,99
8	158,45	7,97
16	79,909	15,80
32	79,909	15,80
64	79,909	15,80

Figure 7: Execution time table for n processors in a leaf strategy

As we saw in figure 7, for the leaf strategy, there is a great performance when we added more processors, but when we have 16 or more, the performance is practically the same. The best SpeedUp is in 16 processors or more, so the best performance is reached with these 16 processors because there is the maximum parallelism possible. In order to not make this document very extensive, we are just showing the best performance histogram, in the figure 8.

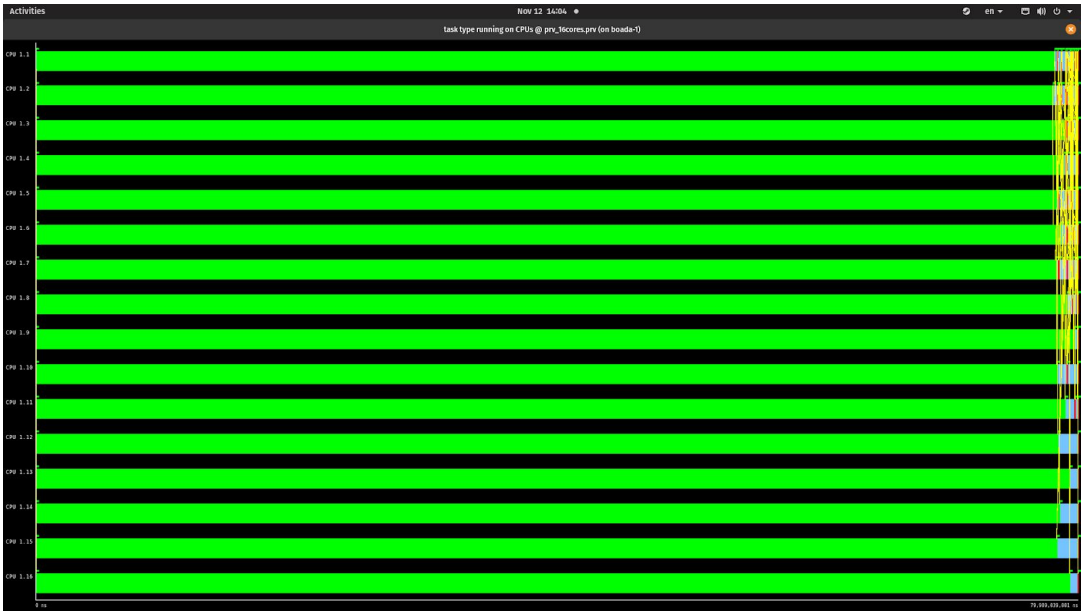


Figure 8: Execution time histogram for 16 processors in a leaf strategy

In Figure 9, we can see the same table but for the tree strategy. We can observe that the results are quite similar in both tables, the speedUp is practically the same, but at 16 processors or more, there is best execution time (not significant).

Processors	Execution time(s)	Speed up
1	1263,26	1
2	632,01	1,99
4	315,83	3,99
8	158,90	7,95
16	79,908	15,80
32	79,908	15,80
64	79,908	15,80

Figure 9: Execution time table for n processors in a tree strategy

Like in the previous case, we are going to show in Figure 10 the histogram for the best case of tree strategy, which is 16 too. As we can see, the main difference is that the workload is balanced quite better in the tree strategy, but is not significant.
For the moment, we cannot make conclusions because with *Tareador* when we have a ‘ready’ task, we can execute her, but in a real execution that does not happen.

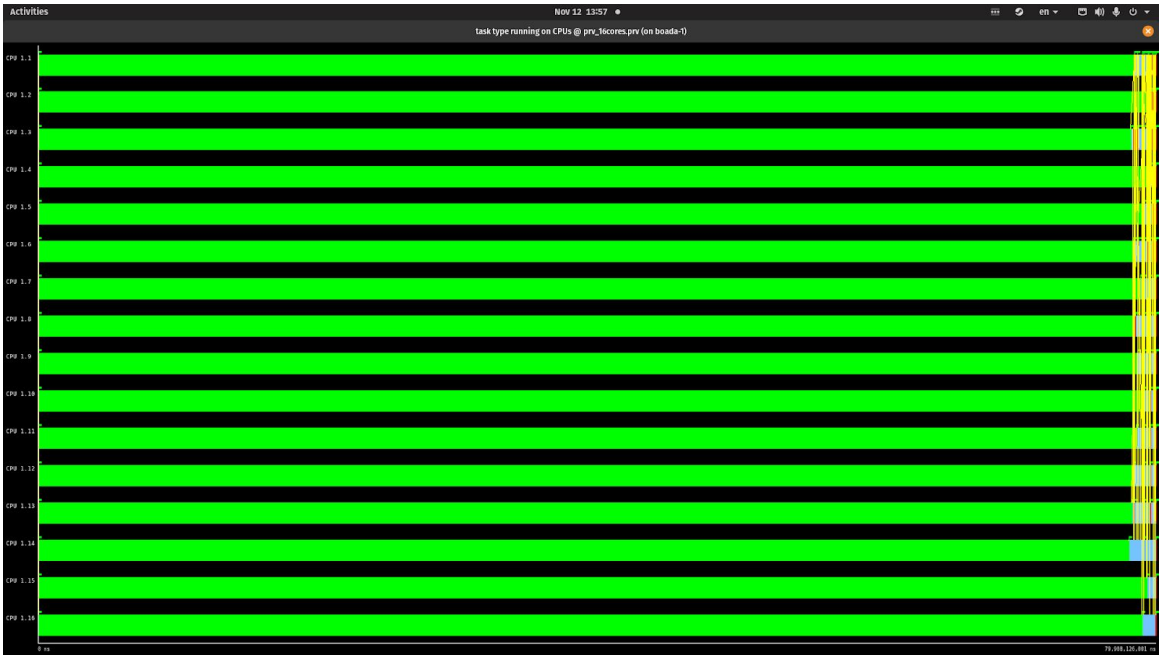


Figure 10: Execution time histogram for 16 processors in a leaf strategy

3. Performance analysis

Leaf Strategy

For this strategy, and as we explained in the previous block, we implemented the idea of task creation in each base case of the recursive divide and conquer algorithm. This code is available in leaf.c code. We can see what we did in Figures 11 and 12.

```
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);
    }
}
```

Figure 11: Code from multisort method of leaf.c

```
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);
    }
}
```

Figure 12: Code from merge method of leaf.c

Next, we will see the Task execution histogram of this strategy, at Figure 13. We can observe that most of the time spent on the execution is running in a single thread. That is the initialization and clear functions.

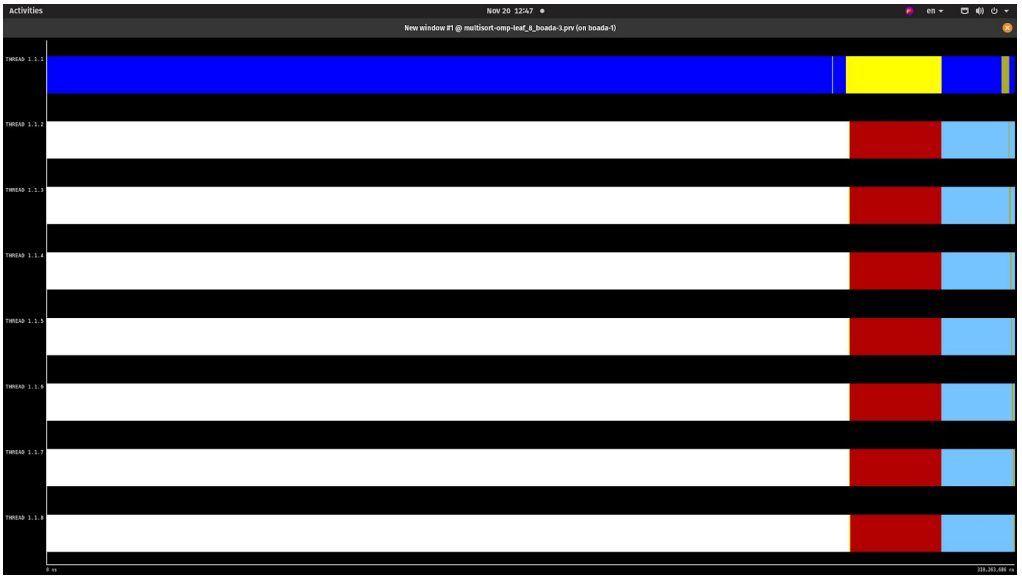


Figure 13: Tasks execution with paraver for the leaf strategy

Finally, at Figure 14 we can see that the strong scalability is not good, this is because merge calls can not start until all the multisort calls have finished, and also to the number of tasks created. As we can see, best performance is on 16 processors.

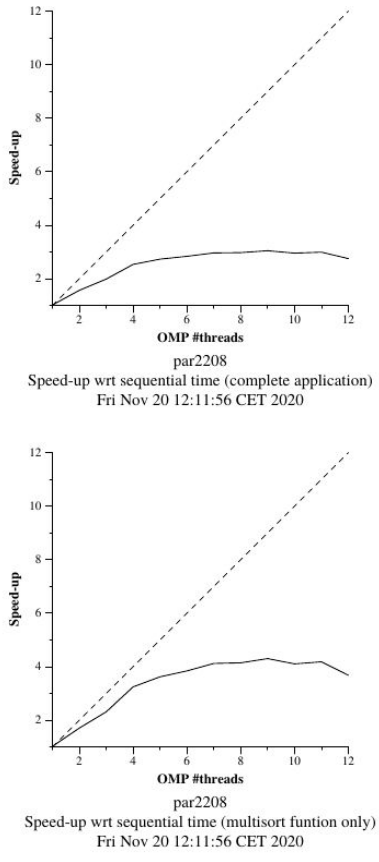


Figure 14. Scalability and speed up plots for leaf strategy (complete application and multisort only)

Tree Strategy

As we have seen before, leaf strategy is not so good in terms of scalability, so we are going to study tree strategy. Which is the contrary of the leaf, as we explained in the *Parallelization Strategies*. This code is available in tree.c code. We can see what we did in Figure 15.

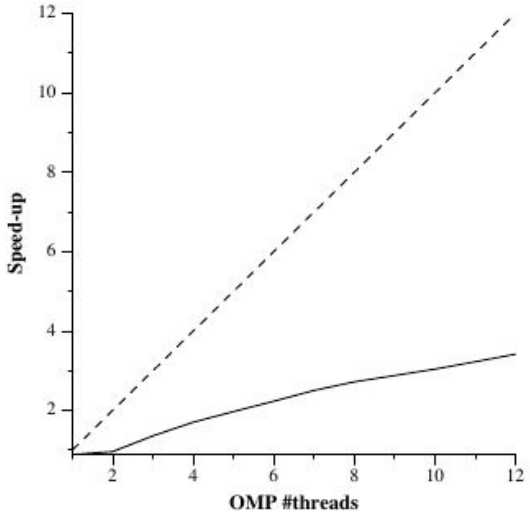
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
    #pragma omp task
        merge(n, left, right,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

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);
    }
}
```

Figure 15: Code of tree version

Next, we are going to analyse the histogram of the execution of our tree function. In the figure 16, we can see that the performance is quite better than in the leaf, the execution time is lesser and workload is balanced between threads. That is because all threads help on task creation because each recursive call is a task.

Figure 16:Task execution paraver for tree.c

Last, we are going to observe the figure 17. In this figure we can see the plots of the speedup of our code execution. As we can see, the speedup for all the application is better than the leaf one, because it never goes down, and the speedup is growing. If we focus on the second one, the multisort function only plot is notably better too, because, like in the complete application case, is permanently growing.
However, the speedup is not as good as we expected, because it is far to be perfect. For this reason we are going to explore other ways to program it, like in the next section, tree function with cutoff.



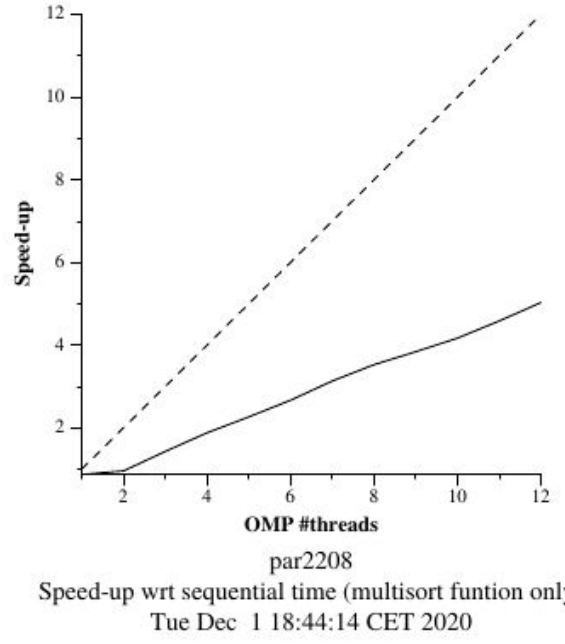


Figure 17: Plots for scalability of tree.c code (complete application and multisort only)

Tree Strategy with Cut-Off

In order to reduce the overhead obtained due to task creation, we program a cut-off mechanism to limit the number of tasks created depending on the recursion level. We can see our code in Figure 18, and in the code in tree-cutoff.c.

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

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

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

Figure 18: Code for tree-cutoff.c

First, we executed the code of version 0 of the cut-off with 8 processors, as we can see at Figure 19. There are only 4 threads doing work, and just one is doing task creation in the first level of recursion. This level of cut-off seems like it is not the best one, because it does not help us very much.

Figure 19: Execution of the program with cutoff = 0

Now, we will see the cutoff mechanism but with the value 1, that way we can compare both values and it may help us to understand how it works.
As we can see at Figure 20, the execution will change the threads who make the work, but now, there are more than 4 threads doing things, so workload is balanced between threads. As in the previous one, just one thread is creating tasks. Both paraver histograms look similar, we cannot observe a lot of differences.

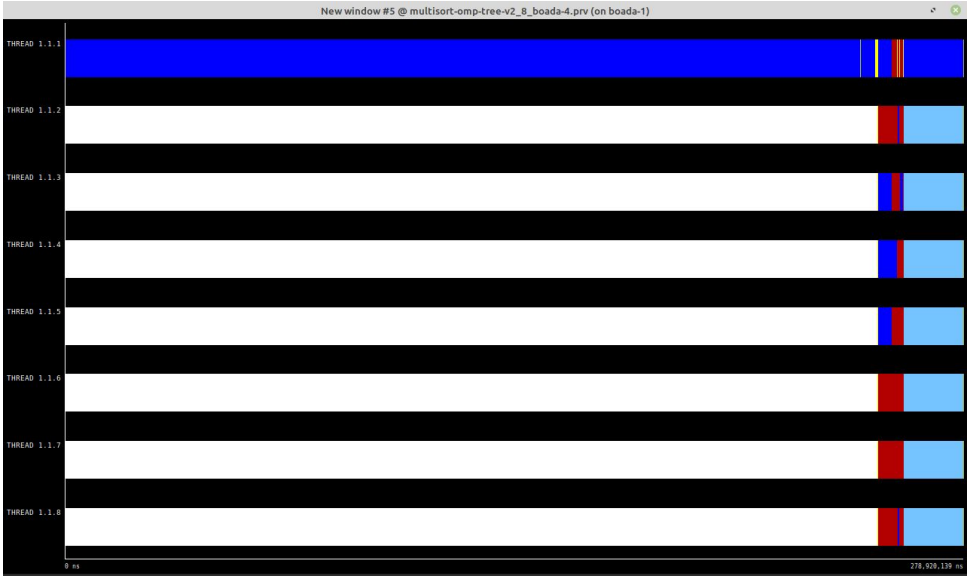


Figure 20: Execution of the program with cutoff = 1

To improve the performance and in order to reach the best cutoff value, we can observe the table on the figure 21. As we can see, the best cutoff value would be 4 or 6, if we try these values, we would see a better performance and workload per thread.

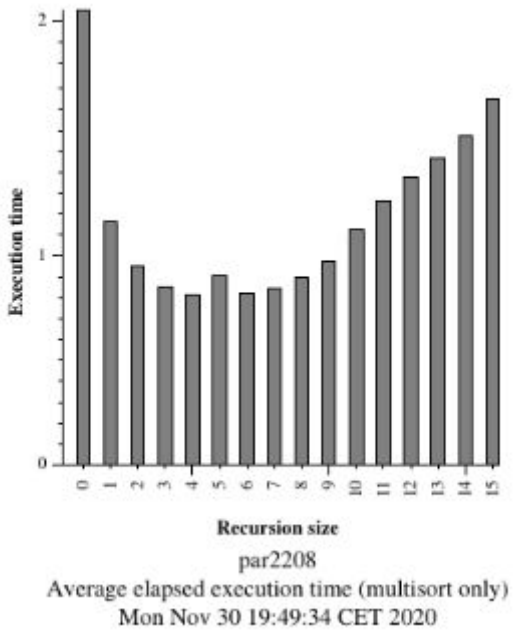


Figure 21. Execution time of the program depending on the level of recursion where we apply the cutoff strategy

Now, we will discuss the speedup plots obtained for the tree strategy with cutoff. We can see at the figure 22 that the multisort function, the ones who are interesting for our work, is practically perfect, so we reach what we want, a better performance of the tree plots.

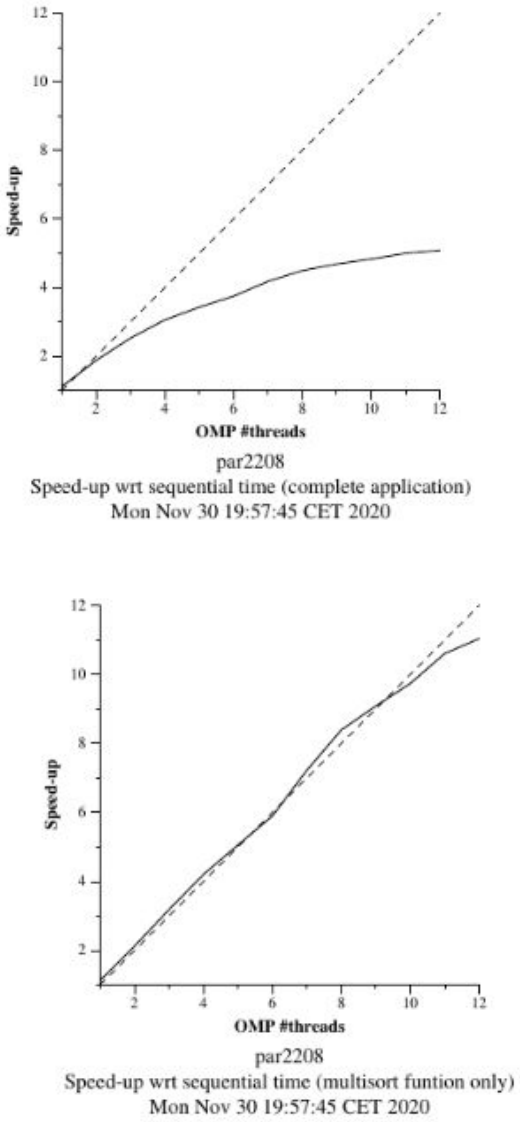


Figure 22: Speedup plots for the cutoff execution

For the **first optional exercise**, we will comment the speedup plots for a big number of threads, for instance 24. We can observe in Figure 23 that the speedup for the multisort is almost perfect too, because it is so difficult that the speedup maintains the same value in all the plot, but it almost does that. Performance is growing because cores can have multiple threads.

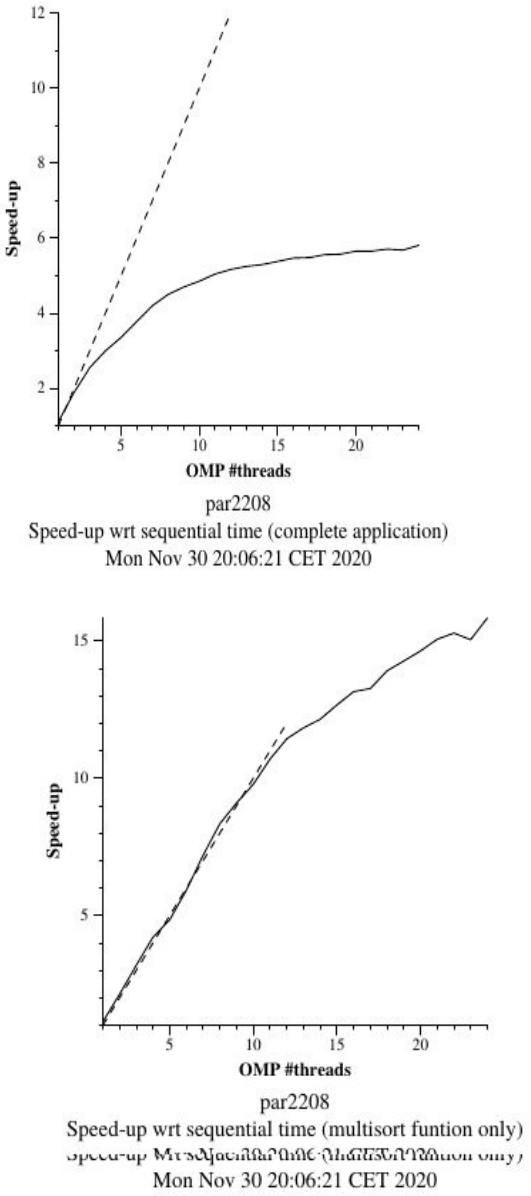


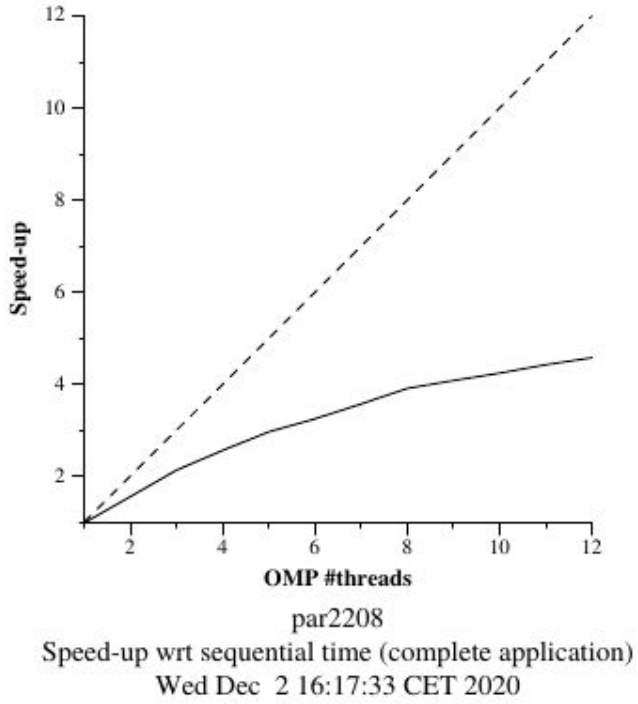
Figure 23: Speedup plots for the cutoff execution with a lot of threads

Task dependency

To end our work, we will try to implement another strategy type in order to parallelize the sorting algorithm. This strategy is task dependency. Code for this block is on tree-dependencias.c.
As we can see in Figure 24, the task creation on Paraver shows that the execution time is similar on the tree strategy, that is because in tree strategy we use task group, which is a similar clause than dependency, in practice. So both histograms are similar.

Figure 24: Task execution paraver for tree-dependencias.c

Finally, we are going to explain the plots of Figure 25. As we can observe, the speedup of the complete application is similar to the tree one, for the reasons we explained before. For multisort, the speedup is good, and similar to tree too.



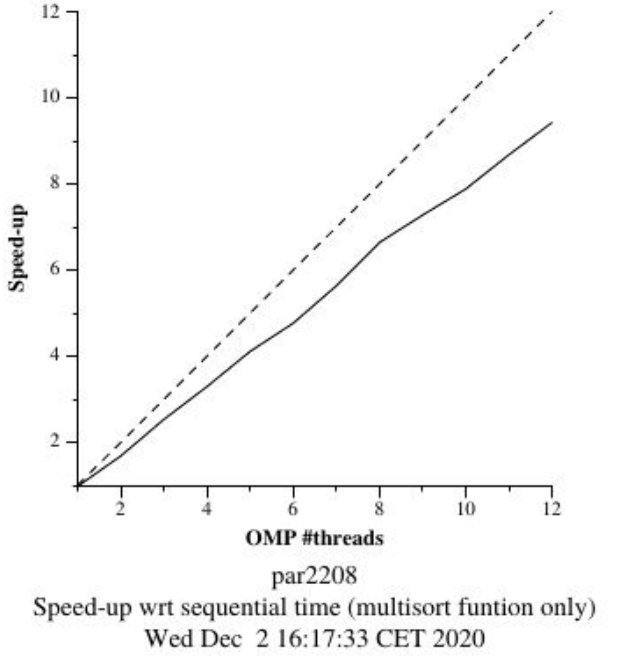


Figure 25: Speedup plots for the execution of tree-dependencias.c

For the **second optional exercise**, we are going to parallelize the functions that initialize data and tmp, in code pot2.c. In Figure 26 we can see the relevant code.

```
static void initialize(long length, T data[length]) {
    #pragma omp taskloop grainsize(length/omp_get_max_threads())
    for (long i = 0; i < length; i++) {
        if (i==0) {
            data[i] = rand();
        } else {
            data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
    }
}

static void clear(long length, T data[length]) {
    #pragma omp taskloop grainsize(length/omp_get_max_threads())
    for (long i = 0; i < length; i++) {
        data[i] = 0;
    }
}
```

Figure 26: Code for opt2.c

Now, we are going to explain the histogram of execution of the opt2.c. In figure 27 we can see the entire histogram, but in order to get more precision, we made a zoom on the important parts, in figure 28. We observe that the plots are not so different from the original parallelization, but in the zoom zone we can evidence that the task creation is more balanced and workload is better for all threads.

Figure 27: Execution histogram with paraver from opt2.c

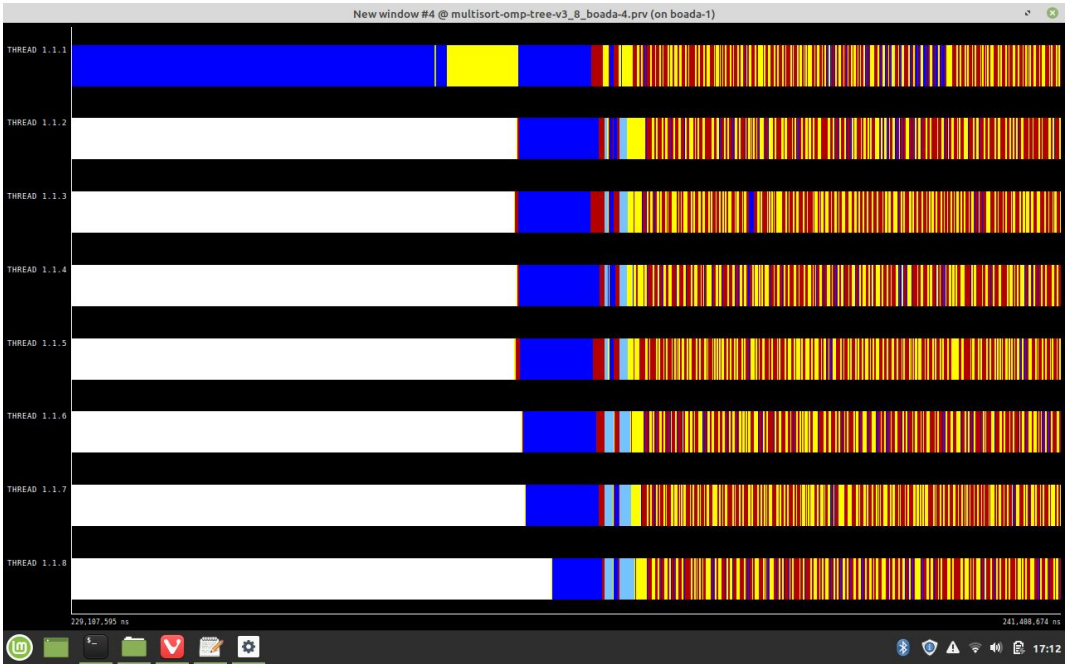


Figure 28: Execution histogram with paraver from opt2.c

To end with this part, we are going to explain speedup plots, on Figure 29, which are obviously better than the normal parallel one. As we can see, the complete application has approximately the same speedup as the multisort, which makes sense if we parallelize the initialize functions.

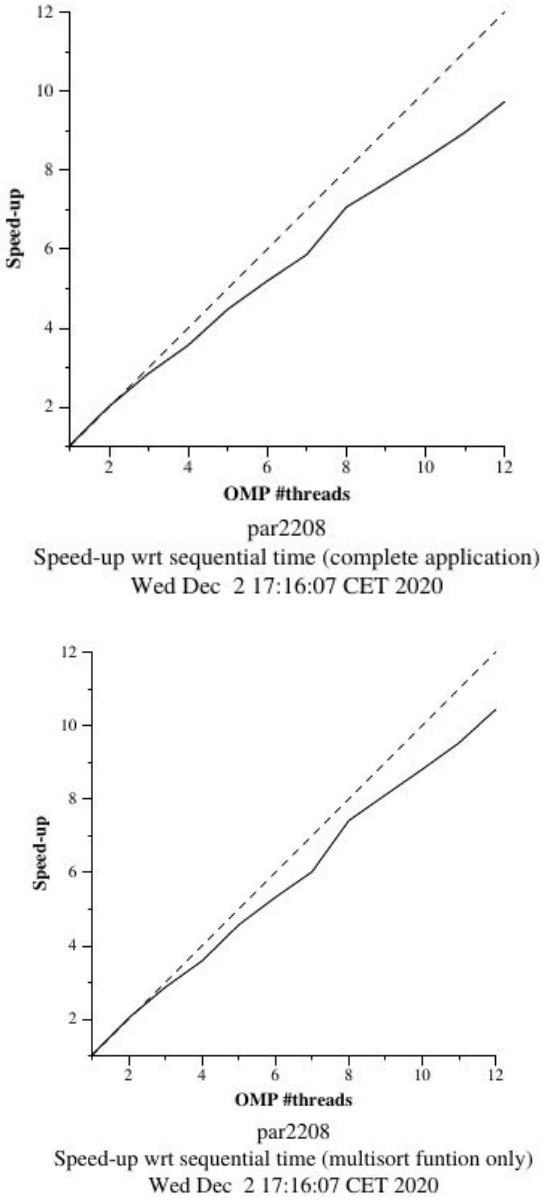


Figure 29: Speedup plots for the execution of tree-dependencias.c

4. Conclusions

With *Tareador*, the dependences are easy to see, and it gives us an idea about how each strategy works, but as we said in the second part of the work, we cannot make conclusions because with *Tareador* when we have a ‘ready’ task, we can execute her, but in a real execution that does not happen.

So it makes us go ahead and try the strategies, and the result, as in the theory, saids that tree strategies are better in order to parallelize this type of recursion functions in the majority of cases.
For this reason, we went further on the study of the tree parallelization adding cutoff, where the overhead was improved, but speedup was improved too, and it is more evident that the noise of the overhead.

With the parallelization of the function initialize and clear of the multisort, the speed-up was big as expected. The parallelizable part was bigger and this did have an effect of a faster execution, which we could predict thanks to Amdahl’s Law.