Parallelism

Lab 3: Divide and Conquer parallelism with OpenMP: Sorting

Marian Danci & David Valero
Group 42, par4207
Fall 2018-19

Index

Session 1: Task decomposition analysis for Mergesort

1.1 ”Divide and conquer”

During this sessions of Lab3, we work with a sort algorithm, Mergesort, which combines a ”divide and conquer” mergesort strategy that divides the initial list into multiple sublists recursively, a sequential quicksort that is applied when the size of these sublists is sufficiently small, and a merge of the sublists back into a single sorted list.
In this laboratory sessions, we work with the code in *multisort.c* that implements the strategy recursively invoking functions *multisort* and *merge*.

We compiled the sequential version of the program using ‘*make multisort’* and executed the binary and this was the output:

```
./multisort
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
Initialization time in seconds: 0.848534
Multisort execution time: 7.021439
Check sorted data execution time: 0.016682
Multisort program finished
```

The default values when unspecified: *./multisort -n 32768 -s 32 -m 32*

1.2 Task decomposition analysis with Tareador

After executing the sequential version, we started analyzing the task decomposition with Tareador.

First of all, we create a Tareador task for each recursion call in the merge and multisort function, before the call, we put a `tareador_start_task("name_task")` and after the call a `tareador_end_task("name_task")`. So, this strategy generates a different task (with a particular name) everytime a recursive function is called.

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

                tareador_start_task("merge2");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("merge2");
        }
}
```

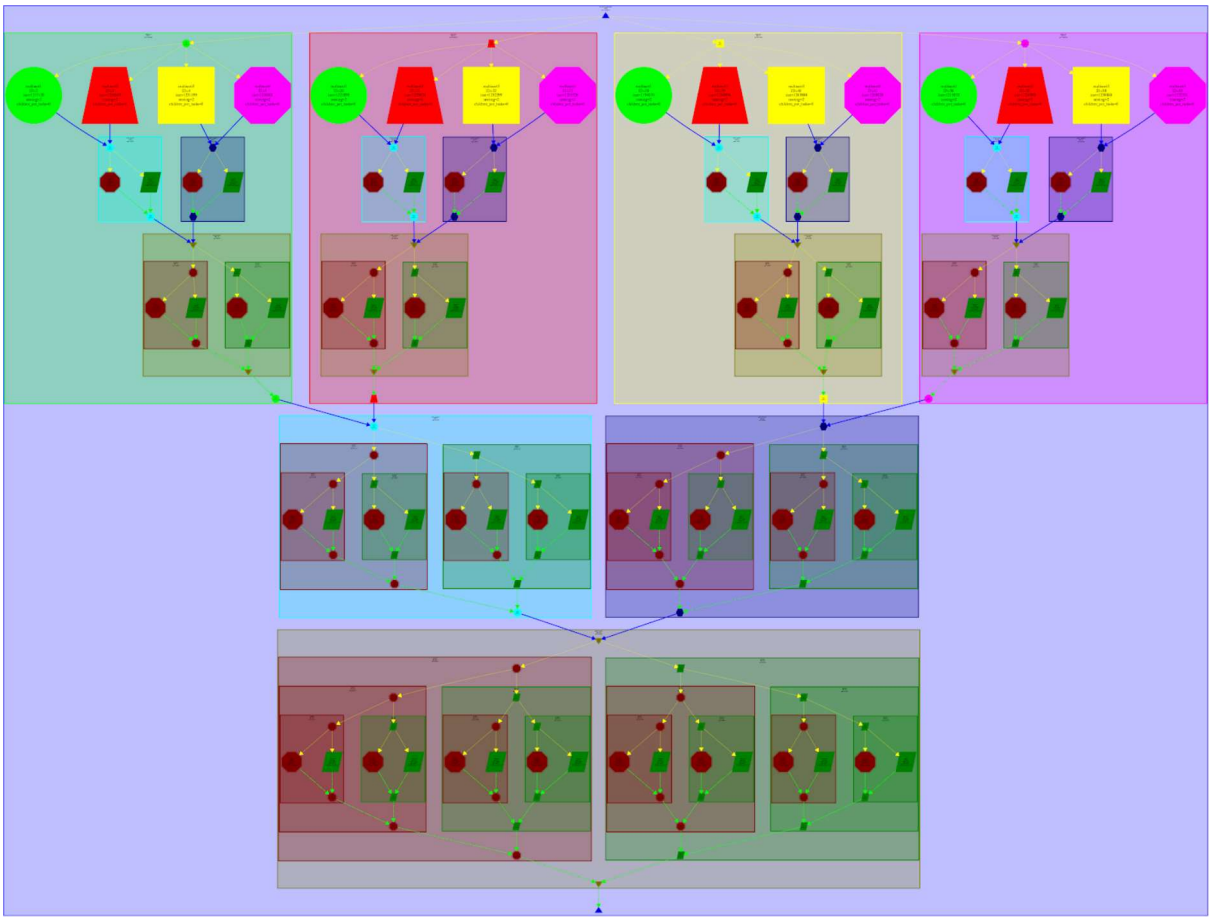
Code1: multisort-tareador.c, merge part.

```
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("multi-merge1");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("multi-merge1");

                tareador_start_task("multi-merge2");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("multi-merge2");

                tareador_start_task("multi-merge3");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("multi-merge3");
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Code2: multisort-tareador.c, multisort part.

Once modified, we compiled using '*make multisort-tareador'* and sent *'run-tareador.sh'* script to the execution queue, this script generates a task graph (picture1).

As we can see in the picture, in the beginning, we have vectors of 32k elements, and then it comes down to 4 parts of 8k elements, these are the first 4 big figures in the graph. The next level, that we have inside these first figures there are calls to *multisort* with a size of 2k.



Picture1: Task dependences graph of multisort-tareador

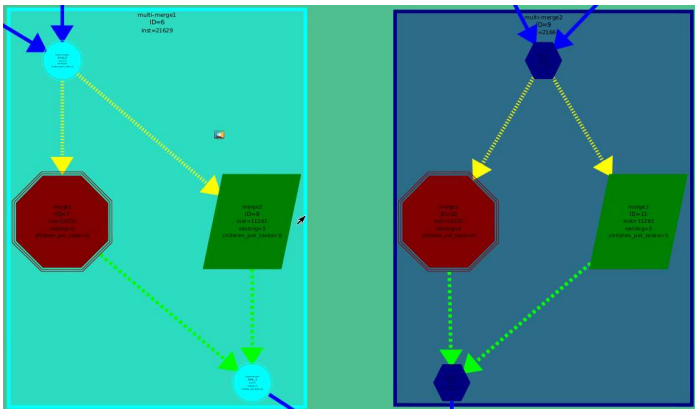
As we can see, the graph follows a patron, that we can see more clearly in Picture2. That patron shows us that the dependencies are created due to:

merge1, depends on *multisort1* and *multisort2*

merge2, depends on *multisort3* and *multisort4*

merge3, depends on *multi-merge1* and *multi-merge2*

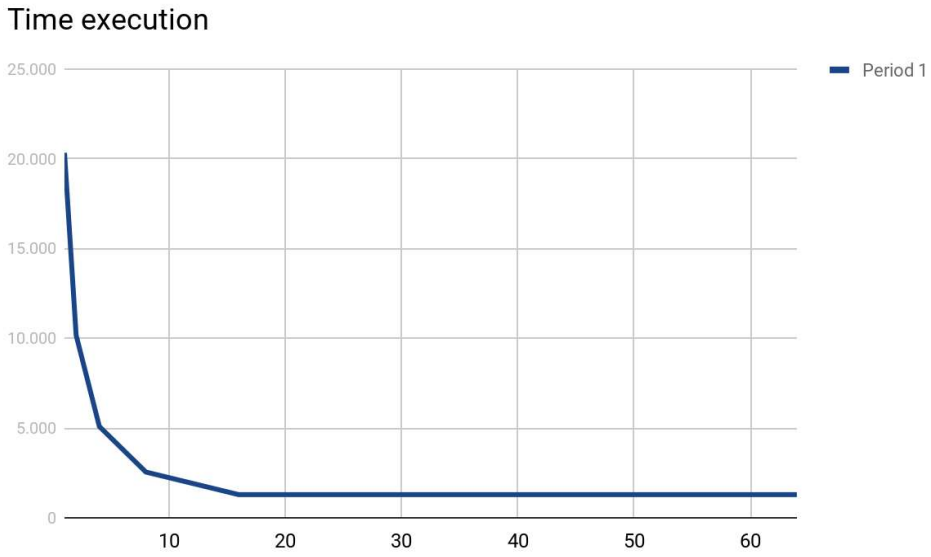
Instead, *basicsort* and *basicmerge* are independent each other.



Picture2: Zoom of the Task dependences graph of multisort-tareador

In order to predict the parallel performance and scalability with a different number of processors, we simulate in Tareador the parallel execution using 1, 2, 4, 8, 16, 32 and 64 processor.
In the following plot, we can see the result of execution in Tareador with a different number of processors, to predict the parallel performance and scalability.

```
1 - 20.334
2 - 10.173
4 - 5.086
8 - 2.550
16 - 1.289
32 - 1.289
64 - 1.289
```



Plot1: The parallel execution using 1, 2, 4, 8, 16, 32 and 64 processors.

As we can see, the execution time decreases by half when we change from one to two, from two to four, from four to eight and from four to sixteen. In the last two changes (from sixteen to thirty-two and from thirty-two to sixty-four) the time does not change, so we can conclude that we have reached the limit of improvement of the execution time.

Session 2: Shared-memory parallelization with OpenMP tasks

After analyzing the task decomposition for Mergesort, we have worked in paralyzing the original sequential code using OpenMP.
From this moment, we have worked in two parallel version: Leaf and Tree:

- In Leaf we defined a task for the invocations of basicsort and basicmerge once the recursive divide–and–conquer decomposition stops.
- In Tree we defined tasks during the recursive decomposition, when invoking multisort and merge functions.

For each version, we followed the same steps: we changed the code, we compiled it, executed it and analyzed it.
First of all, we are going to work with the Leaf Strategy.

2.1 Leaf Strategy

For this version, we put the `#pragma omp parallel` and `#pragma omp single` in the main function, before calling multisort for the first time.

```
        fprintf(stdout, "Arguments (Kelements): N=%ld, MIN_SORT_SIZE=%ld, MIN_MERGE_SIZE=%ld\n", N/BLOCK_SIZE,
MIN_SORT_SIZE/BLOCK_SIZE, MIN_MERGE_SIZE/BLOCK_SIZE);
        fprintf(stdout, "                      CUTOFF=%d\n", CUTOFF);

        T *data = malloc(N*sizeof(T));
        T *tmp = malloc(N*sizeof(T));

        double stamp;
        START_COUNT_TIME;

        initialize(N, data);
        clear(N, tmp);

        STOP_COUNT_TIME("Initialization time in seconds");

        START_COUNT_TIME;

        #pragma omp parallel
        #pragma omp single
        multisort(N, data, tmp);

        STOP_COUNT_TIME("Multisort execution time");

        START_COUNT_TIME;

        check_sorted (N, data);

        STOP_COUNT_TIME("Check sorted data execution time");

        fprintf(stdout, "Multisort program finished\n");
        return 0;
}
```

Code3: Last part of the Main function, where multisort function is called for the first time.

After that, we focus on parallelizing the multisort function. There are two ways to do the tasks wait for each other, one of it is at the If clause put a `#pragma omp taskgroup`, which groups all the recursion calls of multisort in one task it means that they can not execute at the same time in parallel.
And the one which we apply is to put two `#pragma omp taskwait`. Due to of the two calls to the merge function have to wait for the multisort calls to end (as we see in the last chapter), for the same reason, we put a second taskwait before the last merge call.

At the else clause we put a `#pragma omp task`, to generate a task everytime a basicsort is called.
After all these modifications, the multisort function has been modified using leaf strategy.

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                //#pragma omp taskgroup
                //{
                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);
        }
}
```

Code4: Multisort function with leaf strategy.

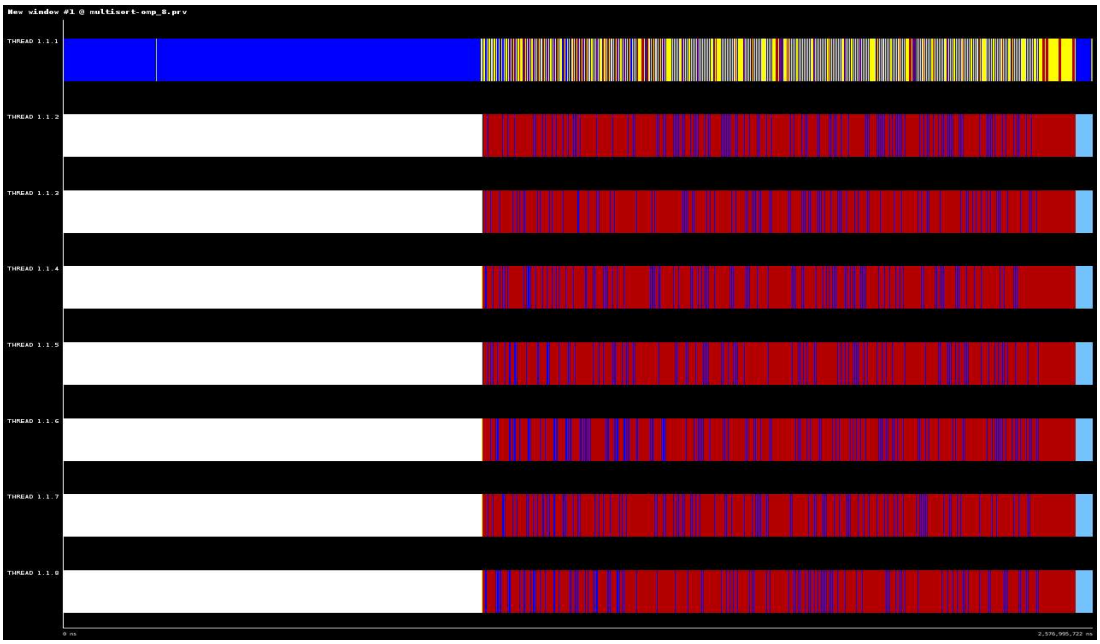
Next, we parallelize the merge function. At the if clause we put one `#pragma omp task`, that creates a task everytime basicmerge is called. Nothing has been done in the else clause.

```
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);
        }
}
```

Code5: Merge function with leaf strategy.

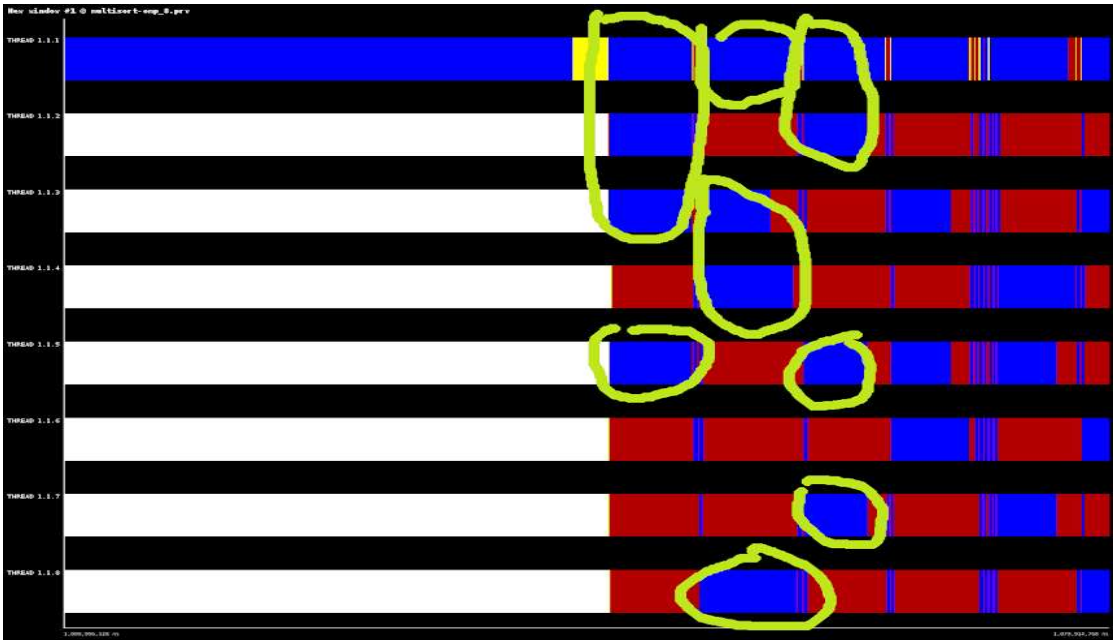
After paralysing all the code as we explain before, we compile using the ‘multisort-omp’ target in the Makefile to generate the executable file and we submit it using the ‘submit-omp.sh’ (specifying 8 processors for the parallel execution).
We submit the ‘submit-omp-i.sh’ script to trace the execution of our OpenMP program.

In order to understand the behaviour of the parallel execution, we use Paraver, that generates the graphic of Picture3.



Picture3: Paraver result with the leaf strategy

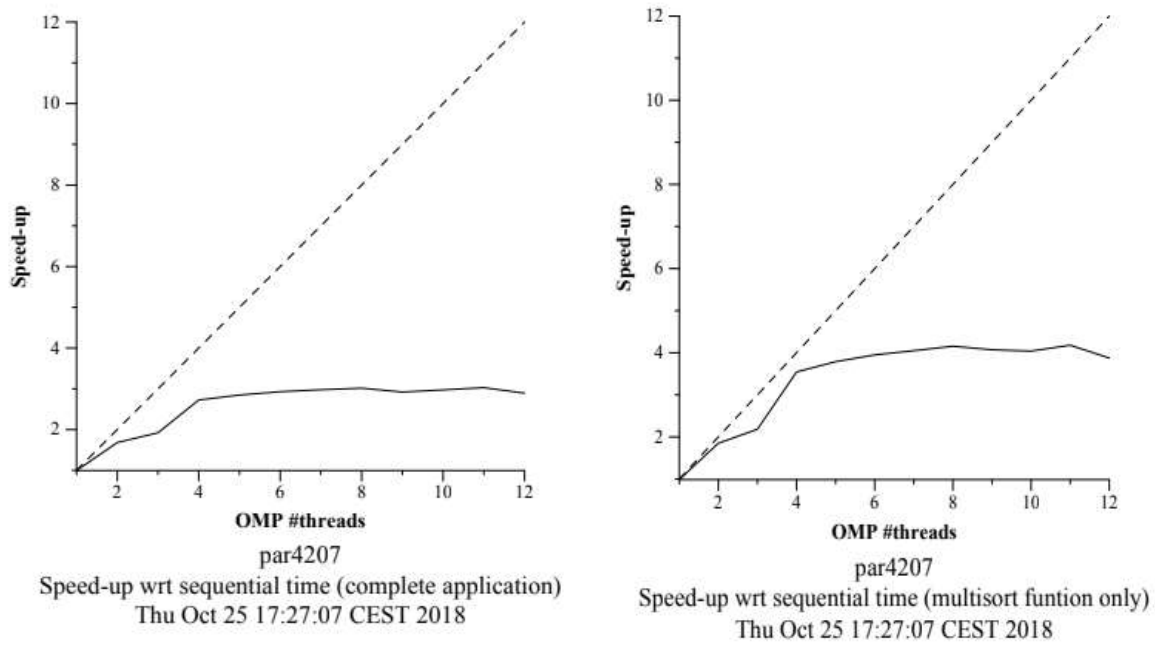
We see that there is a big first sequential period (before multisort call) and after that a paralyzation with the eight processors. The blue zones are zones where the processor is doing tasks and the red zones are periods of time where they are waiting (doing nothing).
At a first look, we see that there are a lot of red zones.
To analyze it in a better way we zoom it, and we obtain Picture4.



Picture4: Paraver result with the leaf strategy, zoom it and mark it some waiting periods of the processors.

In that picture, we can see that there are a lot of periods that the processor is doing nothing because it is waiting for another task finalization. We mark it some blue zones to show that.

This makes us think that maybe the leaf strategy is not the best to parallelize our program, because it generates too many dependencies that it cannot be solved, and the tasks have to wait too much time for task synchronization.
To confirm our thoughts, we analyze the scalability of our parallel program by looking at the two speed–up plots when submitting the 'submit-strong-omp.sh' script:



Plot2: Speed-Up plots of the leaf strategy. left the complete application, right the mutisort function only.

As we see at the plots, the Speed-ups of the complete application and multisort function are increased by increasing the threats to 4, from there it stabilizes.
This is because of how leaf strategy in the multisort works, which, as we saw in Pictures 2 and 3, the $T\infty \simeq T4$.
The main reason for that is the taskwaits of multisort function, where the tasks after them have to wait for the finalization of the task before them to guarantee the good synchronization.

2.2 Tree Strategy

For the tree strategy, as we did in the leaf strategy, we put the `#pragma omp parallel` and `#pragma omp single` in the main function, before calling multisort for the first time.
After that, we focus on parallelizing the multisort function. At the If clause we put a `#pragma omp task`, for each multisort call, take a look that we still have the taskwait like in the leaf strategy.
After their four, we put a `#pragma omp taskwait` in the same way that we did at the leaf strategy.
The next two merge calls, each one has one `#pragma omp task`, making it possible for them to run at the same time.
Before the third and last merge call of the multisort function, another taskwait and `#pragma omp task` are put. So that merge call has to wait until the other merge calls tasks end.
To finalized with the if clause, a taskwait is put to ensure that no task is executed until the previous merge call ends.
Nothing has been done at the else clause.
After all these modifications, the multisort function has been modified using leaf strategy.

```
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);
        }
}
```

Code6: Multisort function with tree strategy.

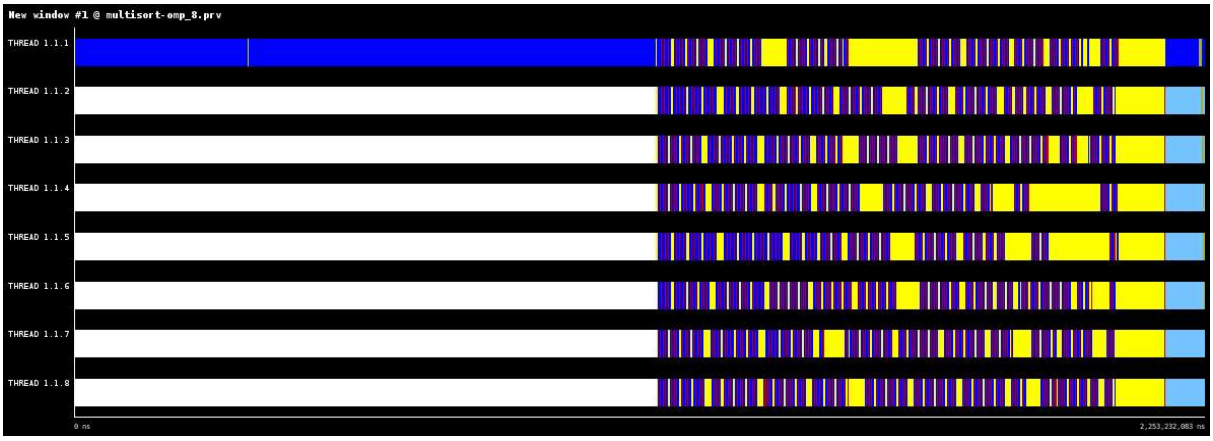
To finalized with the coding of tree strategy, we edit the merge function.

In the if clause nothing has been done but at the else clause we put a `#pragma omp task` before each merge recursivity call and a `#pragma omp taskwait` after each other. This guarantees that the merge call waits until the previous one has finished.

```
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 taskwait

                #pragma omp task
                merge(n, left, right, result, start + length/2, length/2);
                #pragma omp taskwait
        }
}
```

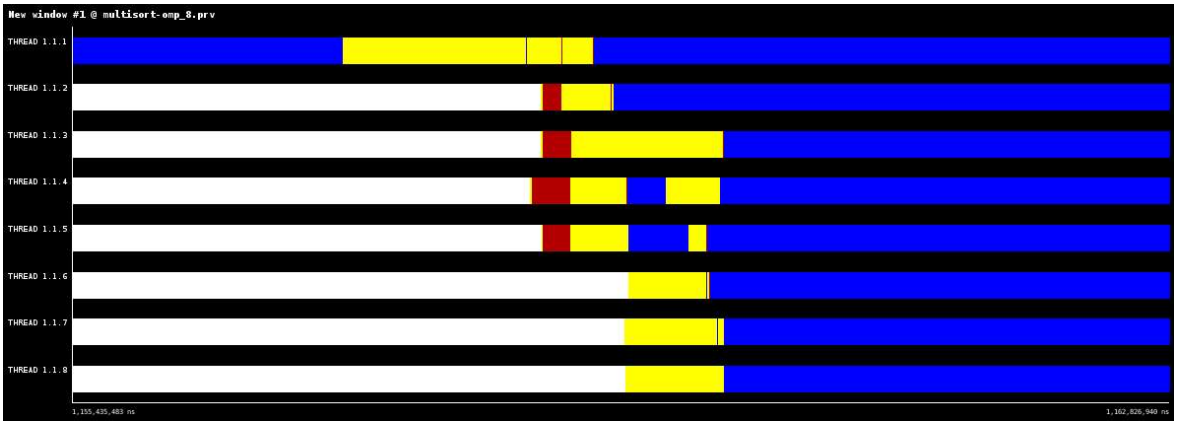
Code7: Merge function with tree strategy.

After paralysing all the code as we explain before, we compile using the 'multisort-omp' target in the Makefile to generate the executable file and we submit it using the 'submit-omp.sh' (specifying 8 processors for the parallel execution).
We submit the 'submit-omp-i.sh 'script to trace the execution of our OpenMP program.
In order to understand the behaviour of the parallel execution, we use Paraver, that generates the graphic of Picture5.
As we see in the leaf strategy, there is the same big first sequential period (before multisort call) and after that a realization with the eight processors.
But this time, there are many more blue areas than red.



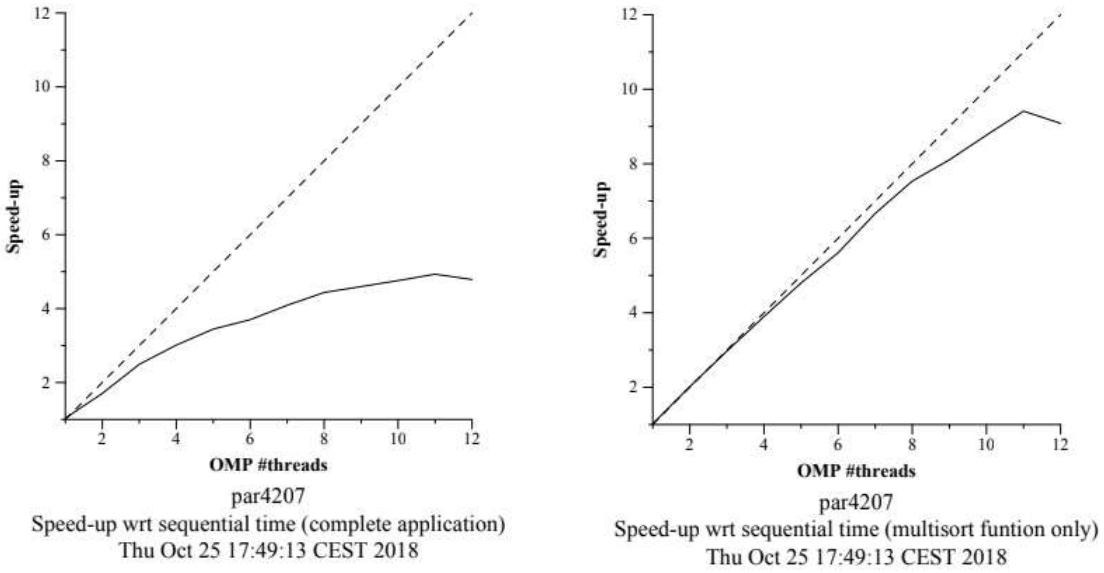
Picture5: Paraver result with the tree strategy

As we did before, we zoom it to analyze it in a better way, we obtain the Picture6.



Picture6: Zoom of the Paraver result with the tree strategy

As we can see, the tree strategy optimize better the eight processors, and they are almost all the time doing tasks.
This makes us think that the tree strategy is better, in that case, to parallelize our program.
As we did before, we analyze the scalability of our parallel program by looking at the two speed–up plots when submitting the 'submit-strong-omp.sh' script:



Plot3: Speed-Up plots of the tree strategy. left the complete application, right the mutisort function only.

As we see at the complete application plot, the Speed-up increase as more threats it has. We also see that the Speed-up stabilizes when the number of threads is larger.
Something different happens at the multisort function plot, where the Speed-up increase much more, and we see that the progress is more homogenous than in the other plot.
This is so because, although the multisort function is getting faster, the period prior to the first call is not paralyzed. We are spending the same time for the first non-paralyzed period.
This means that as the number of threats increases, the percentage of time that the multisort is running will decrease. Concluding, that each thread more, the improvement will be lower.

2.3 Task cut–off mechanism

Finally we modify the parallelization of the Tree version in order to include a cut–off mechanism that controls the maximum recursion level for task generation, based on the use of the OpenMP final and mergeable clauses, to control the number of tasks generated (and their granularity).
For this version, we have not change the main function, is the same as Leaf and Tree versions.
Then, we modify the multisort function. We put one `#pragma omp task final(depth >= CUTOFF) mergeable` before each four first multisort calls. Then, as we saw in the last two version, a taskwait is putted to garantee the good synconization.
For the next two merge calls, we put another `#pragma omp task final(depth >= CUTOFF) mergeable` for each other.
Finally, before the last merge call, we put another taskwait and task final.
As we saw at the tree version, anything is put in the else clause.

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

                                #pragma omp taskwait

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

                                #pragma omp taskwait

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

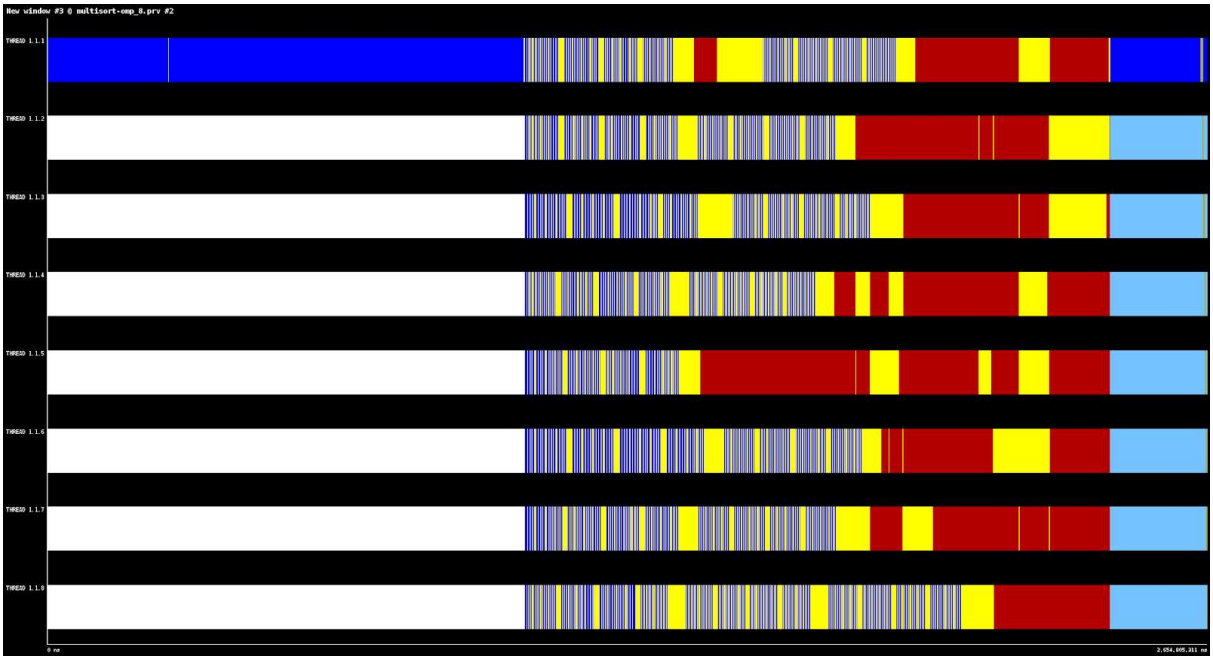
Code8: Multisort function with task cut-off strategy

After modificing the multisort function, we has modified the merge function.
As we see in the last version, we only put pragmas at the else clause. we only add a `#pragma omp task final(depth >= CUTOFF) mergeable` before each merge calls.

```
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
                #pragma omp task final(depth >= CUTOFF) mergeable
                merge(n, left, right, result, start, length/2, ++depth);
                //#pragma omp taskwait

                #pragma omp task final(depth >= CUTOFF) mergeable
                merge(n, left, right, result, start + length/2, length/2, ++depth);
                #pragma omp taskwait
        }
}
```

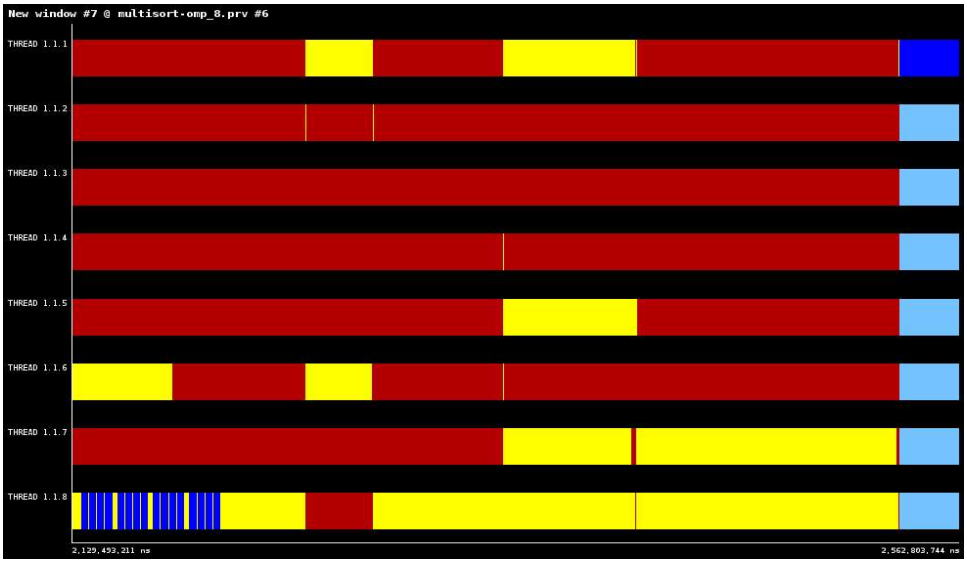
Code9: Merge function with task cut-off strategy.

Once implemented, we generate a trace (using the submit-omp-i.sh script and using the optional argument that specifies the cut–off value) for the case in which we only allow task generation at the outermost level (i.e. level 0) and we visualize the trace with Paraver.



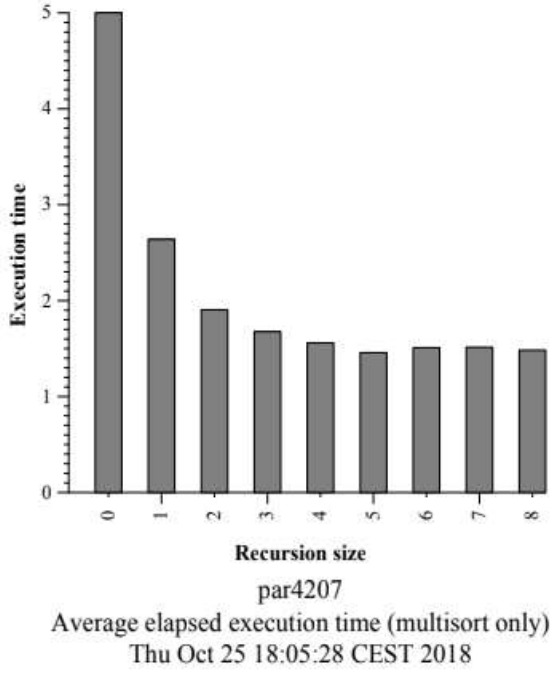
Picture7: Paraver result with task cut-off strategy.

We can see at Picture7 very clear when the different processors arrive at the cut-off time, when they stop doing tasks.
In a more clear way is seen in the Picture7, where we zoom the last period of the paralyzed sector.



Picture8: Zoom of the Paraver result with task cut-off strategy.

Concluding, we can confirm that the cut-off stategy is working good.
After that, we submit the 'submit-cutoff-omp.sh' script to explore different values for the cut–off the argument, trying to observe if there is an optimum value for it.
To obtain the optimum value, we analyze the scalability by looking at the speed–up plot generated when submitting the 'submit-strong-omp.sh' script.



Plot4: Speed-Up plot of the task cut-off strategy.

As we can see, the optimum cut-off value is 4 or 5, after those values, the execution time is stabilized. This could be produced because, after those recursion sizes, the processor has too many tasks that can not be paralysed in a good way.

Session 3: Using OpenMP task dependencies

To finalized with Lab3, we change the Tree parallelization of the previous chapter in order to express dependencies among tasks and avoid some of the taskwait/taskgroup synchronizations.
First of all, the main and merge functions have not changed, the main has the `#pragma omp parallel` and `#pragma omp single` and the merge function the two `#pragma omp task` before each merge call. In the following part of the code, we can see that we did not change the merge function because there are no dependencies between them. However, we have a `#pragma omp taskwait` at the end to ensure that we execute the merge functions before returning to the parent function.

```
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
        }
}
```

Code10: Merge function.

On the other hand in multisort function, to avoid the taskwait/taskgroup clauses we used the depend clause to generate dependencies between each recursion call and their respective variables. However, a taskwait clause is necessary at the end to avoid going on level up if a thread has not finished yet. In the following part of code, you can see the implementation of dependencies.

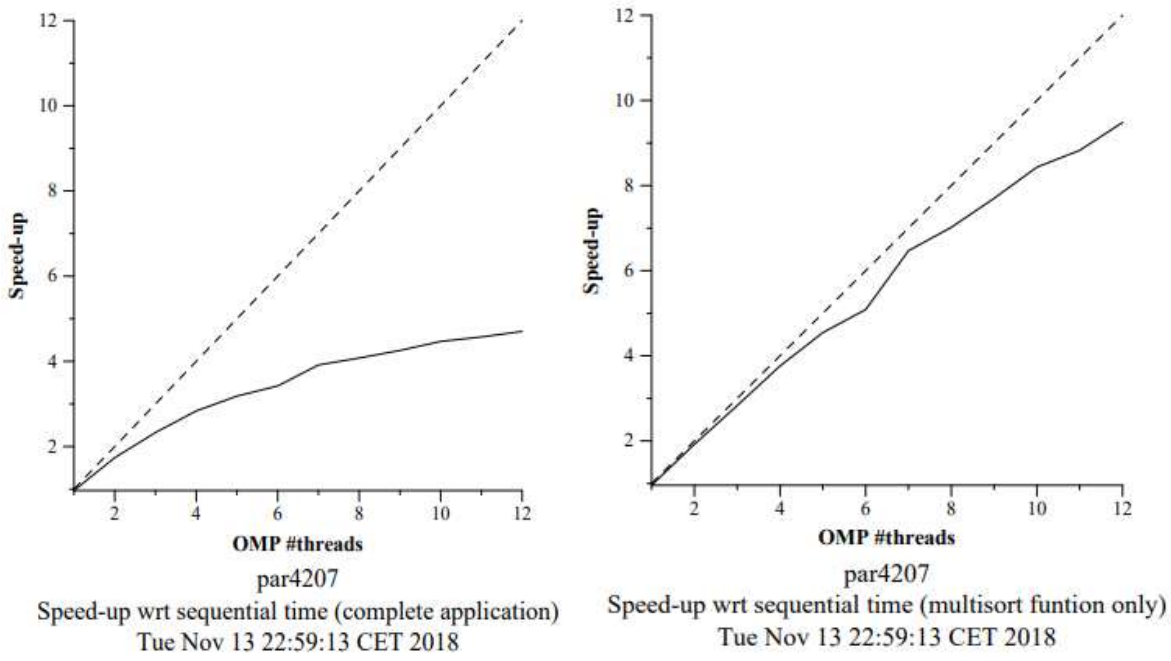
```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                //#pragma omp task final(depth >= CUTOFF) mergeable

                #pragma omp task depend(out: tmp[0])
                multisort(n/4L, &data[0], &tmp[0]);
                #pragma omp task depend(out: data[n/4L])
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                #pragma omp task depend(out: data[n/2L])
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                #pragma omp task depend(out: data[3L*n/4L])
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

                //#pragma omp taskwait

                #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);
                #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);

                //#pragma omp taskwait

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

Code11: Multisort function.

To verify that the program not throw errors about unordered positions we submit to boada with the command: `qsub -l execution submit-omp.sh multisort-omp 8.` With which we obtain a correct output:

```
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=4
Initialization time in seconds: 0.817533
Multisort execution time: 0.883926
Check sorted data execution time: 0.020763
Multisort program finished
```

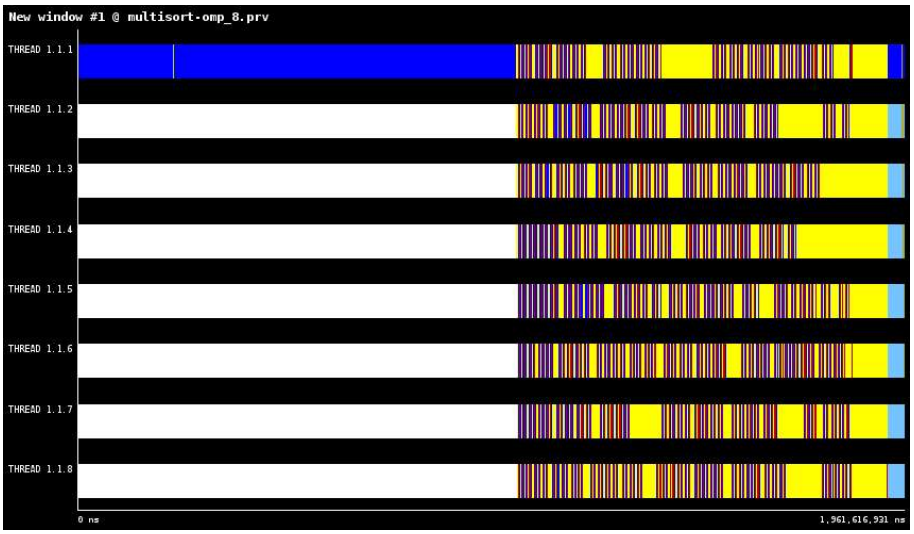
Finally, to analyze the scalability we submit to boada the command: `qsub -l execution submit-strong-omp.sh,` and with which we obtained two speed-up plots, the following ones.



Plot5: Speed-Up plots using task dependencies. left the complete application, right the mutisort function only.

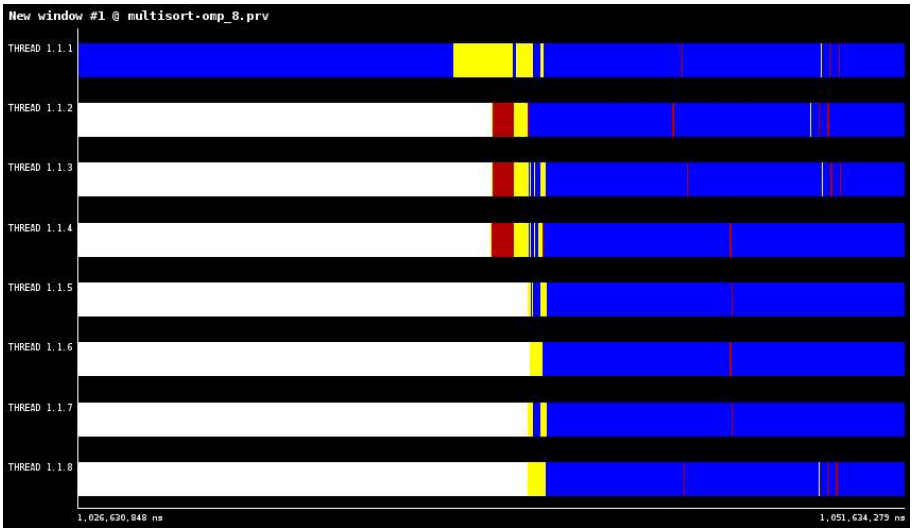
With those plots and the previous of the tree version, we can conclude that there are not significative changes among them, possibly because the parallelism is limited due to the dependencies among son's and parent's tasks.

To finalise with the analysis, we submit the 'submit-omp-i.sh' script to trace its execution and we generate its Paraver file.



Picture9: Trace of Tree version with task dependencies

As we can see, there are a lot of similarities in the non-zoom Picture between the last tree version, to look better the differences we zoom it.



Picture10: Trace of Tree version with task dependencies (zoom)

In the new trace, we can observe that now we do not have four synchronizations at the beginning, when the first thread creates the others, due to of the new dependencies created by the *depend* clause. To conclude, the Paraver file is very similar to the tree strategy one, the two stategis has a very similar result, possibly because the parallelism is limited due to the dependencies among son's and parent's tasks.