ENTREGABLE PARAL·LELISME

TERCER ENTREGABLE

Username: par4105
20 de Novembre de 2018
Tardor del 2018

Àlex Valls
Roger Guasch

Introduction

The aim of this report is to explain the work done in this third laboratory session and also analyze and compare the different versions and results we have studied. Merge sort is one of the fastest sorting algorithms, achieving performances of O(n log n) in worst cases. This performance is achieved by the algorithm recursively dividing the input data into small sets, sorting them and merging the result afterwards. The main objective of this laboratory session would be to analyze its performance if, for instance, we parallelize the merge sort algorithm using different strategies.

In the first part we are going to analyze the results obtained with Tareador program, using the multisort-tareador.c code. In the second part will see how the performance improves over time if we implement a leaf parallelization strategy. Following that, we are going to implement a tree parallel version, with and without cut-off mechanism, in order to compare the performance of both versions with the leaf version. Before finalizing this report, we will see how the tree version of the merge sort behaves if we implement the parallelization strategy using variable dependencies and, in the end, we will complete this report by showing how the different boada's architectures affect the tree's performance version.

Task decomposition analysis for Mergesort

To start this report we are going to analyze how *tareador* represents the different tasks generated during the execution of the program. The graph obtained, after writing the tareador_start_task() and tareador_end_task() before every call to *multisort* and *mergesort*, can be analyzed in Figure 3.1. The first four rectangles represents the four different recursion calls to multisort that, subsequently, call multisort again but with a smaller input size, reducing, in each level, the size of the vector that needs to be sorted. On the bottom of each rectangle the vector is sorted before finishing the execution of the multisort function.
In the middle of the graph, two different calls to merge can be seen, each of them with their respective levels of recursivity. The last rectangle of the graph, the one that is the darkest, is the last call to the function merge that it's function is to merge the result obtained after the two previous calls to the merge function.

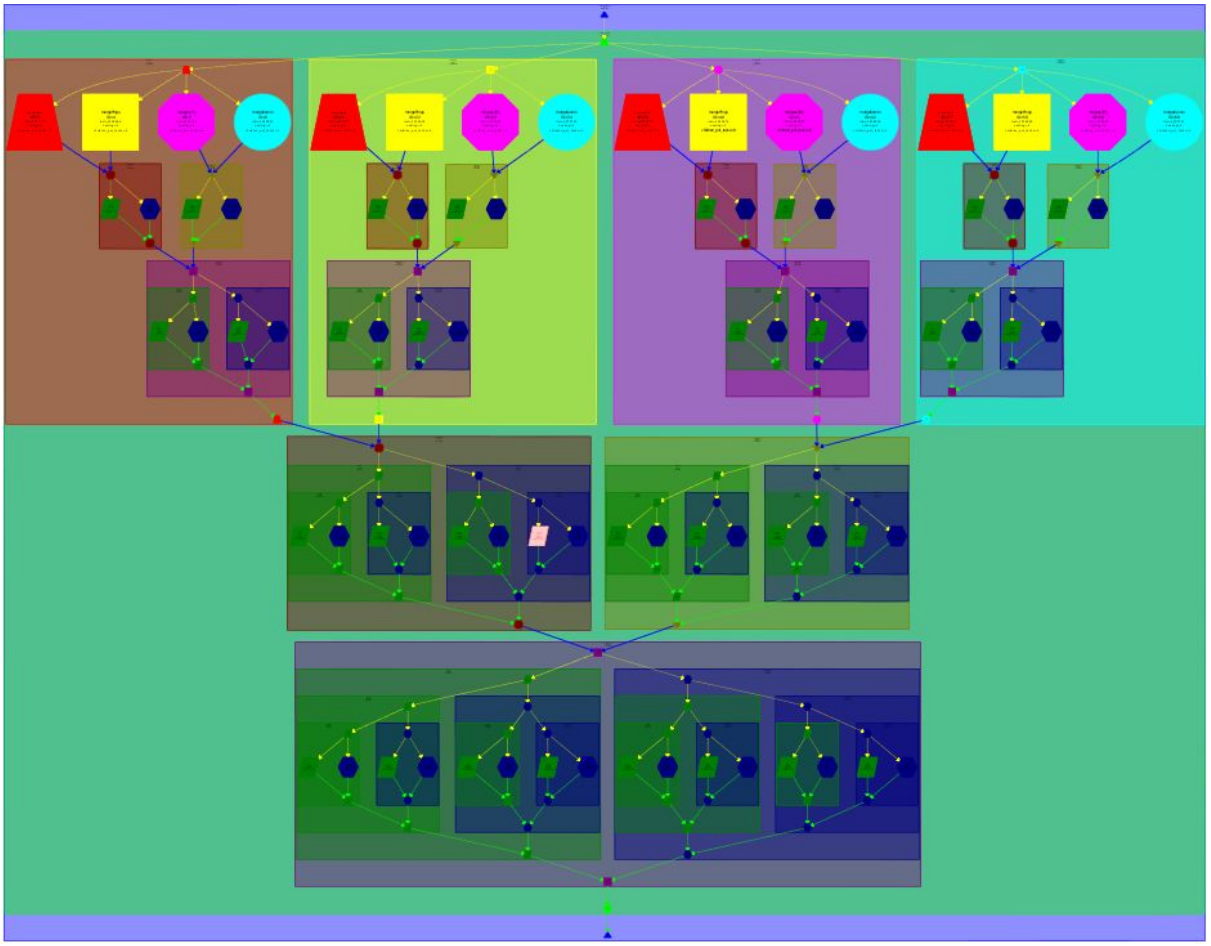


Figure 3.1 Graph obtained using Tareador software

Studying the results that have been obtained we can conclude that the last two mergesort calls have a clear dependency with the multisort calls: mergesort can not start until every previous process have finished. The same happens with the last call to mergesort function inside multisort: It can not be executed until the two previous *mergesort* have ended their execution. Also this analysis can be extrapolated onto the smaller rectangles that represent

either *multisort* and *mergesort* calls: a call to multisort can not be executed until the predecessor task have finished its execution.

In order to get the graph of the Figure 3.1 we have added the calls to tareador_start_task() and tareador_end_task() as seen in Figure 3.2 and Figure 3.3.

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

Figure 3.2 Code snippet of the function merge, showing how tareador functions have been implemented.

```
void multisort(long n, T data[n], T tmp[n]) {
       if (n >= MIN_SORT_SIZE*4L)  {
             // Recursive decomposition
             tareador_start_task("merge3");
             multisort(n/4L, &data[0], &tmp[0]);
             tareador_end_task("merge3");
             tareador_start_task("merge4");
             multisort(n/4L, &data[n/4L], &tmp[n/4L]);
             tareador_end_task("merge4");
             tareador_start_task("merge5");
             multisort(n/4L, &data[n/2L], &tmp[n/2L]);
             tareador_end_task("merge5y");
             tareador_start_task("merge6");
             multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
             tareador_end_task("merge6");

             tareador_start_task("merge7");
             merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
             tareador_end_task("merge7");
             tareador_start_task("merge8");
             merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
             tareador_end_task("merge8");

             tareador_start_task("merge9");
             merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
             tareador_end_task("merge9");
           } else {
                    // Base case
                    basicsort(n, data);
           }
}
```

Figure 3.3 Code snippet of the function multisort, showing how tareador functions have been implemented.

As we know from past studies with *Tareador*, we can simulate the execution time of the program in function of how many processor do we have. The results of this test can be seen in Figure . Knowing the execution time of the program while being executed in different number of threads, we can obtain it's speed up value and the plots shown in Figure and Figure

Threads	Exec Time(ns)	Speedup
1	20.334.411.001	1
2	1.017.373.2001	1,99871699
4	5.086.733.001	3,997538498
8	2.550.610.001	7,972371704
16	1.289.946.001	15,76376917
32	1.289.933.001	15,76392804
64	1.289.933.001	15,76392804

Figure 3.4 Results obtained through tareador simulating the execution of the program with different number of threads.

As it can be appreciated, the execution time of the program is reduced 20 times by just increasing in one the number of processors available. After that, the time is reduced by a half, until we reach 16 threads. Starting from this point, the time will not be reduced although we keep increasing the threads available.

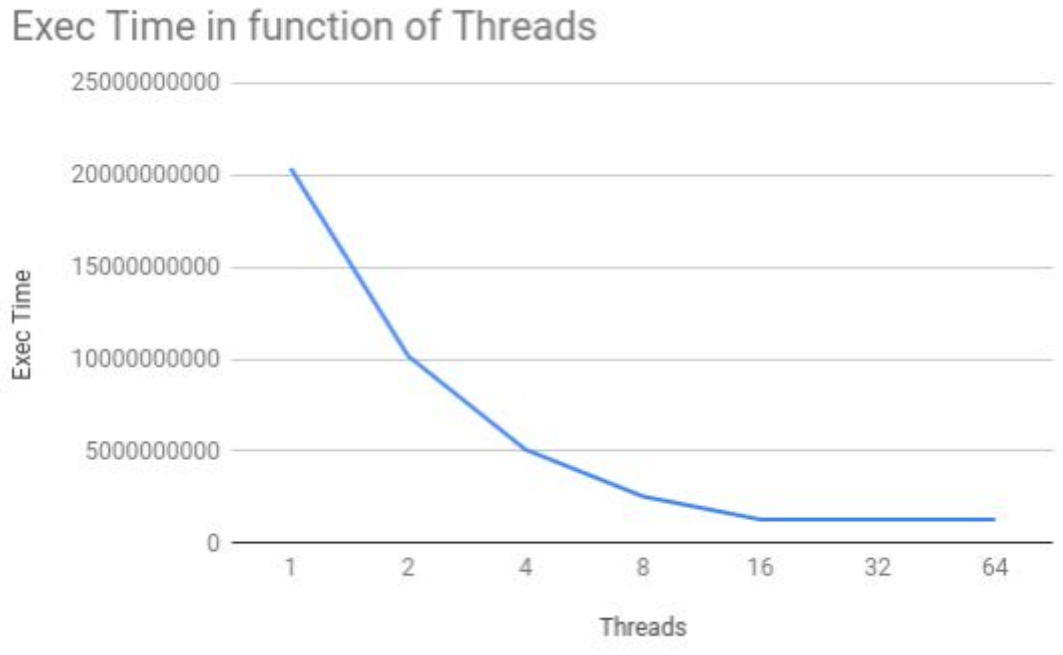


Figure 3.5 Execution time plot obtained with an excel program using the data shown in Figure 3.4.

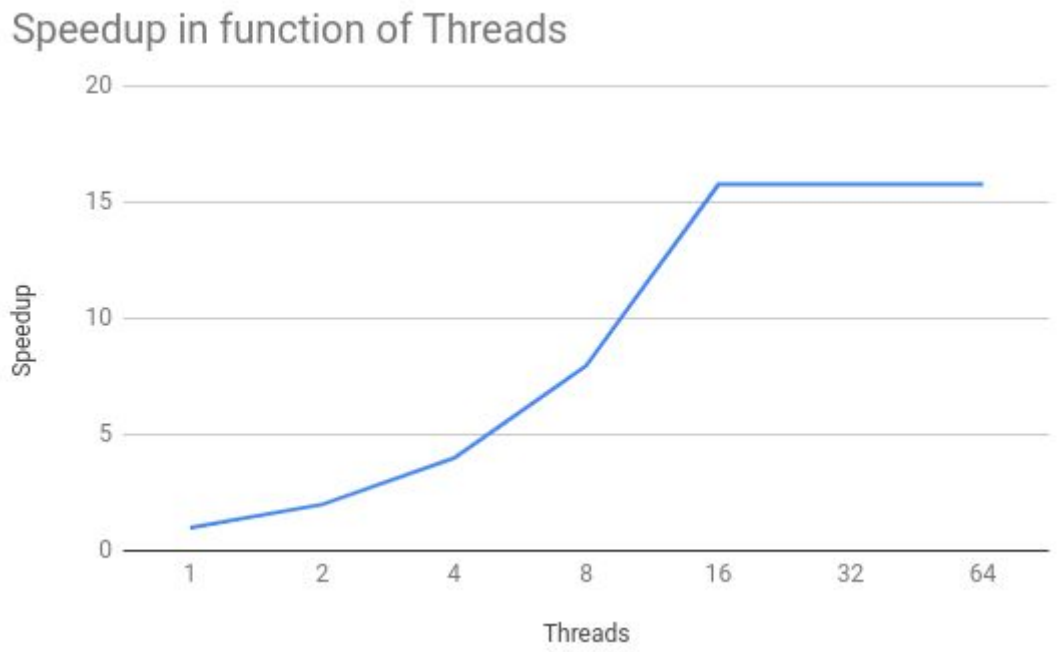


Figure 3.6 Speed up plot obtained with an excel program using the data shown in Figure 3.4.

Shared-memory parallelization with OpenMP tasks

Leaf version

In this section of the report we have tried two different approaches in order to parallelize *multisort* and merge functions. The first option we tried was a Leaf strategy, which its implementation can be found in Figure 3.7, for the details of the merge function, and in Figure 3.8, for the implementation of *multisort* 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
		#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 3.7 Leaf implementation of the merge function

As we can appreciate, **we are only executing the base case of the recursion** -the leaf of the tree- **as a parallel task**. This implementation could be beneficial for trees that their leafs contains an intensive task execution. Although it's not our situation -the execution of *basicmerge* can not be considered as an intensive task- lets see how the execution time of the program behaves.

```
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 3.8 Leaf implementation of the multisort function

Taskwait directive are necessary in order to ensure the correct execution order of the functions, done in parallel.

The statement made before can be demonstrated looking at the plots in Figure 3.9. Improving the parallel function does not has a relatively huge impact on the complete speed-up plot, although there is some improvement on the speed-up of the multisort function only. Increasing the number of threads more than four, does not cause an increase of the speed-up function.

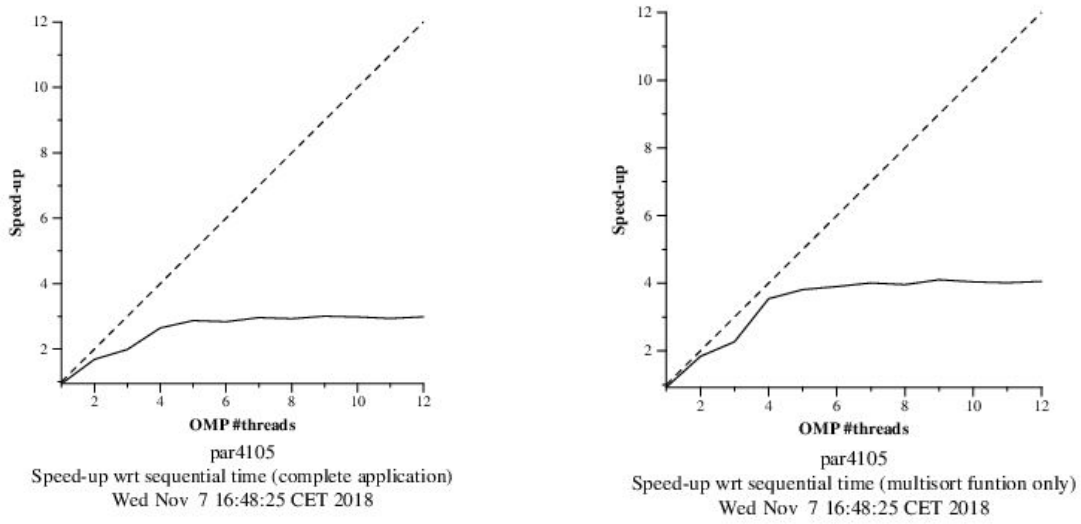


Figure 3.9 Complete speed-up plot and multisort only speed-up plot

The right plot is a clear indicative that, increasing the number of threads to more than four, does not affect the execution. This is caused because **we only have four calls** to *multisort* function.

For paraver analysis. we have submitted the code with the appropriate script, with eight threads, in order to obtain the simulations that can be found in Figure 3.10 and Figure 3.11.

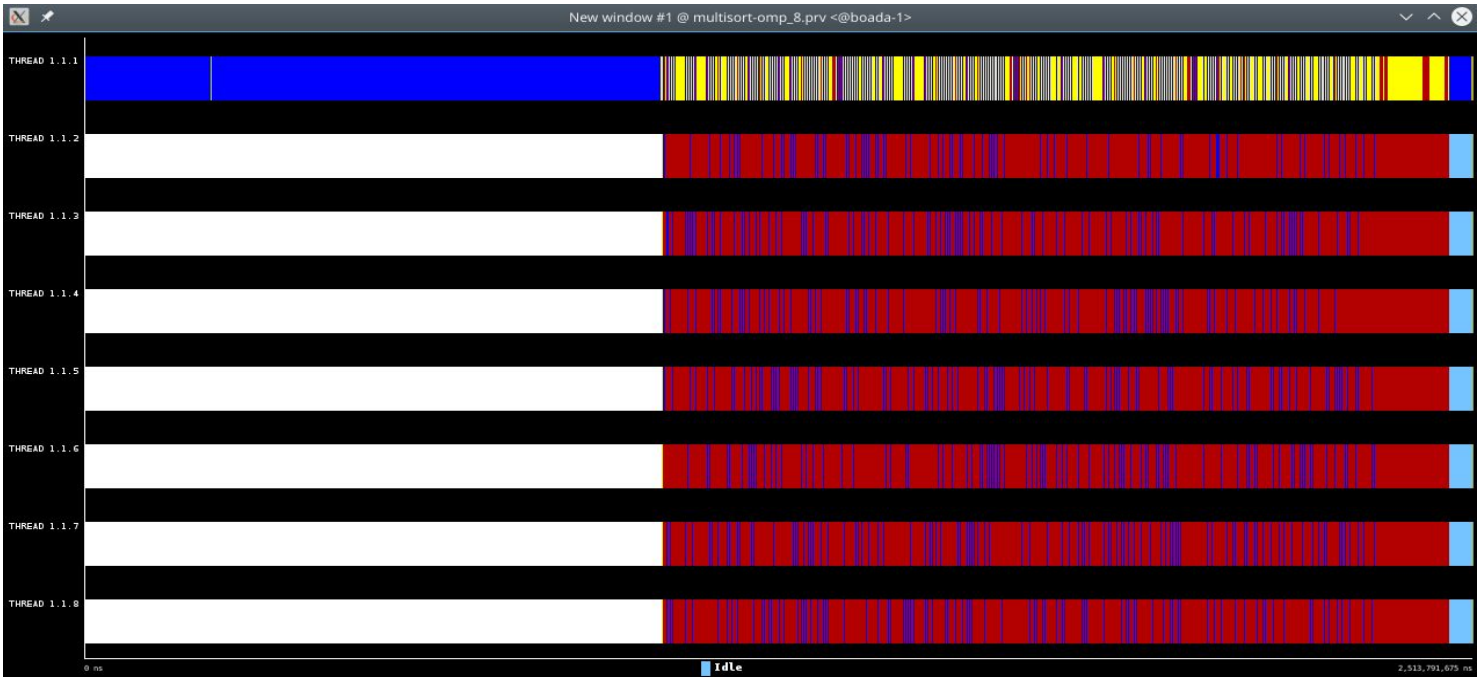


Figure 3.10 Complete paraver result obtained by submitting the leaf version with submit-omp.sh script

For almost half of the time, the code has been executed sequentially, because one thread has to go across every level of the recursion to create all the tasks that needed to be done afterwards.

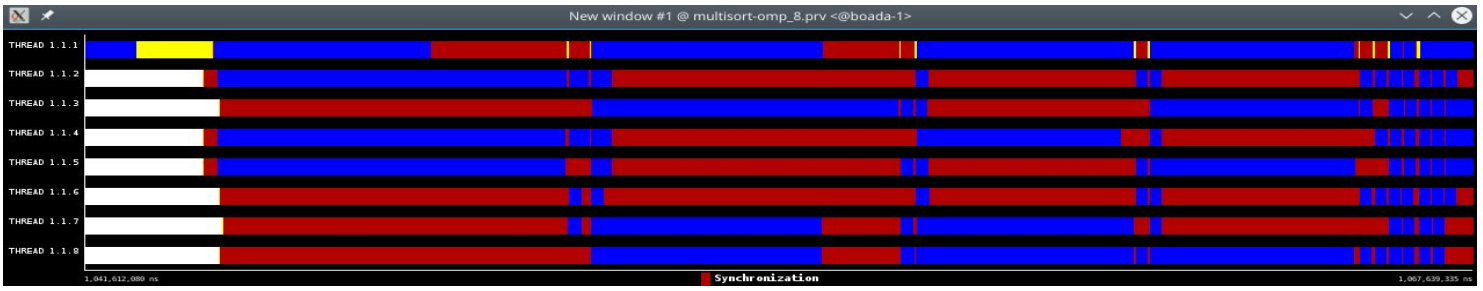


Figure 3.11 Zoom in of the paraver result

The first four blue lines that can be found at the Figure 3.11, starting at the second thread and finishing at the fifth, are the calls of the multisort function. Afterwards, the smaller blue regions, are the calls of the mergesort function.

Tree version

For implementing the tree version of the multisort program we had to made some changes to the code. Now, instead of doing a parallelization for the leafs, the basic case of the recursion, we parallelize all the calls of *multisort* and *basicmerge*. The code is similar as the tree version but changing the small parts, that can be found in Figure 3.12 and Figure 3.13.

```
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);
	}
}
```

Figure 3.12 Modification of the merge function in order to implement a tree parallelization strategy

What will cause this code is that every level of the tree is executed on a parallel node if possible, allowing a fast execution of the recursivity.

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

Figure 3.13 Modification of the multisort function in order to implement a tree parallelization strategy

In order to ensure that any merge call is executed before all the multisort executions, we need to include a taskgroup after the last multisort call. The taskgroup omp directive will force the program to wait until all the executions of multisort have been done and ensuring that the merge has de correct data to produce correct results.

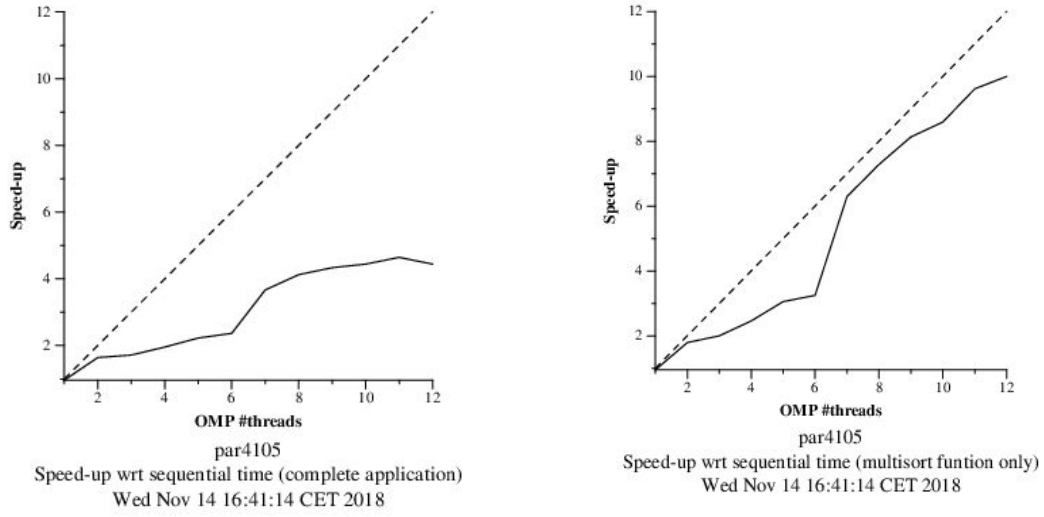


Figure 3.14 Complete speed-up plot and multisort only speed-up plot

It is clear that, by using a tree strategy, we obtain a faster speed-up for all the program. If we

compare the speed-up plot, of the multisort function, obtained for the tree version and the speed-up plot of the leaf function it is clear that the function keeps increasing in function of the number of threads available whereas, in the leaf version, this does not happen. We can state that, by using a tree strategy, we get better results.

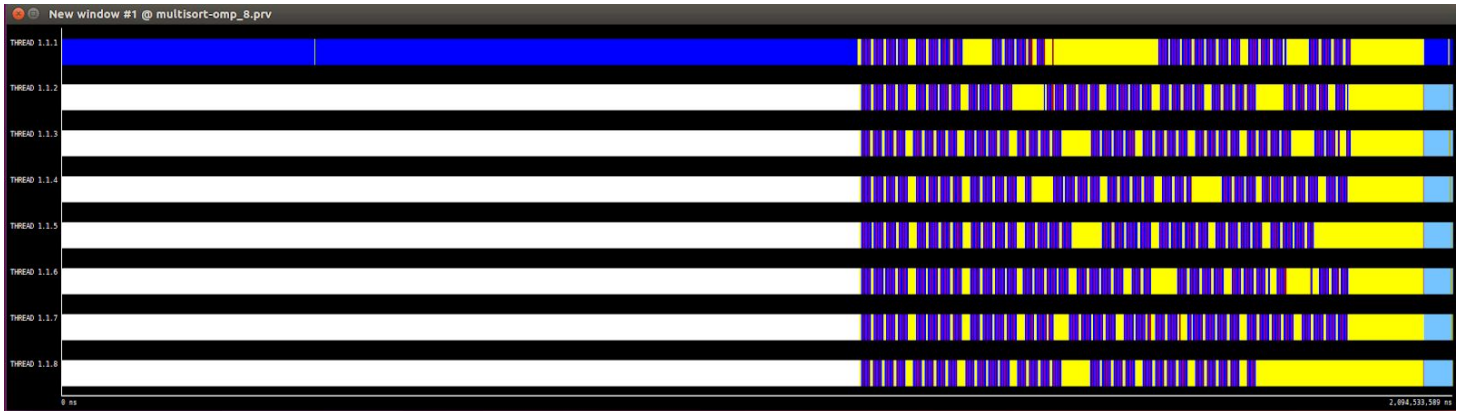


Figure 3.15 Complete paraver result obtained by submitting the tree version with submit-omp.sh script

Although the first part of the paraver result is the same as the leaf version -more than half of the execution of the program is sequential-, the parallel part is smaller causing the total execution time to decrease. Synchronization time -yellow parts- are greater than the previous version, because we need to synchronize and wait until some parallel parts finishes their execution, in order to move up on the execution.

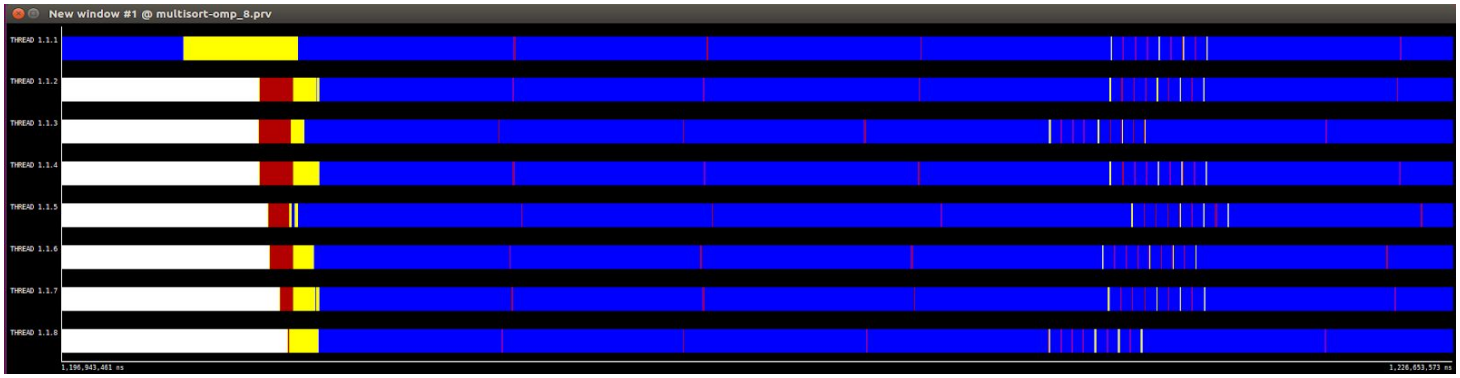


Figure 3.16 Zoom in of the paraver result

Tree version with static cut-off implementation

Cut-off implementation gives us the power to control how many parallel tasks are going to be created. There are two types of cut-off mechanisms: the ones that are static -after reaching some point there are not more creations of parallel tasks-, and the ones that are dynamic -the program controls how many tasks are being executed and decides if it can create more-.

For our concrete version we are going to implement a static cut-off. For doing so we had to implement the code found in Figure 3.17 and Figure 3.18.

```
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+1);
		#pragma omp task final(depth > CUTOFF) mergeable
		merge(n, left, right, result, start + length/2, length/2, depth+1);
	}
}
```

Figure 3.17 Modification of the merge function in order to implement a tree parallelization strategy with cut-off mechanism

The merge and multisort functions now include the correct implementation for a cut-off mechanism. In order accomplish this objective we need an extra parameter for the function, that is going to indicate in which level of the tree we are located. Final clause, before the creation of the parallel task, is going to analyse if the desired level has been reached and, if it happened, stop creating more tasks.

```
void multisort(long n, T data[n], T tmp[n], int depth) {
	if (n >= MIN_SORT_SIZE*4L) {
		// Recursive decomposition
			#pragma omp taskgroup
			{
			#pragma omp task final(depth > CUTOFF) mergeable
			multisort(n/4L, &data[0], &tmp[0], depth+1);
			#pragma omp task final(depth > CUTOFF) mergeable
			multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
			#pragma omp task final(depth > CUTOFF) mergeable
			multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
			#pragma omp task final(depth > CUTOFF) mergeable
			multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
			}
			#pragma omp taskgroup
			{
			merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, 0);
			merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, 0);
			}
		merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, 0);
		} else {
			// Base case
			basicsort(n, data);
		}
}
```

Figure 3.18 Modification of the multisort function in order to implement a tree parallelization strategy with cut-off mechanism.

The same modification has to be done in the multisort function in order to achieve the desired behaviour of the program. Now, instead of using *taskwait*, we need to wait until all child and subsequent tasks finishes before executing the *merge* function. Then we should

use a task group to ensure this desired behaviour.

Knowing the optimal value for the cut-off mechanism is crucial in order to study correctly this version. We could do so by analysing how the execution time behaves while applying different values for the cut-off mechanism. Luckily, this study is done by *submit-cutoff-omp.sh* script, the result which we had obtained can be found in Figure 3.19.

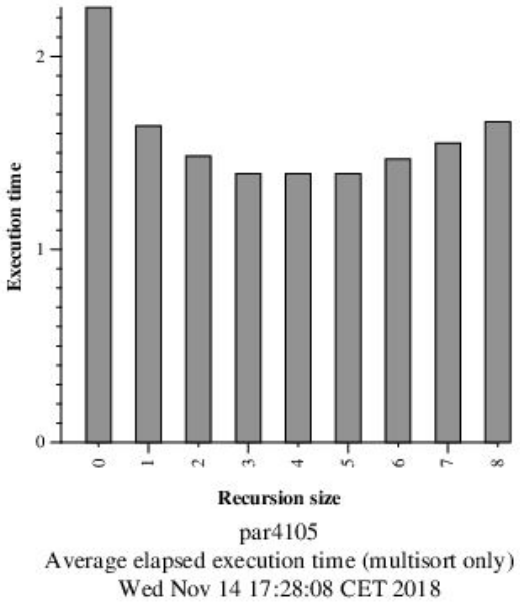


Figure 3.19 Optimal value for the cut-off mechanism of the tree version

After knowing that the optimal cut-off value could be an integer value from three to five, then we can submit again the code in order to obtain the plots to study the speed-up gained by the tree version with cut off. We can argue that we get a smaller size because the generated tree, after traversing all its levels, is significantly small.

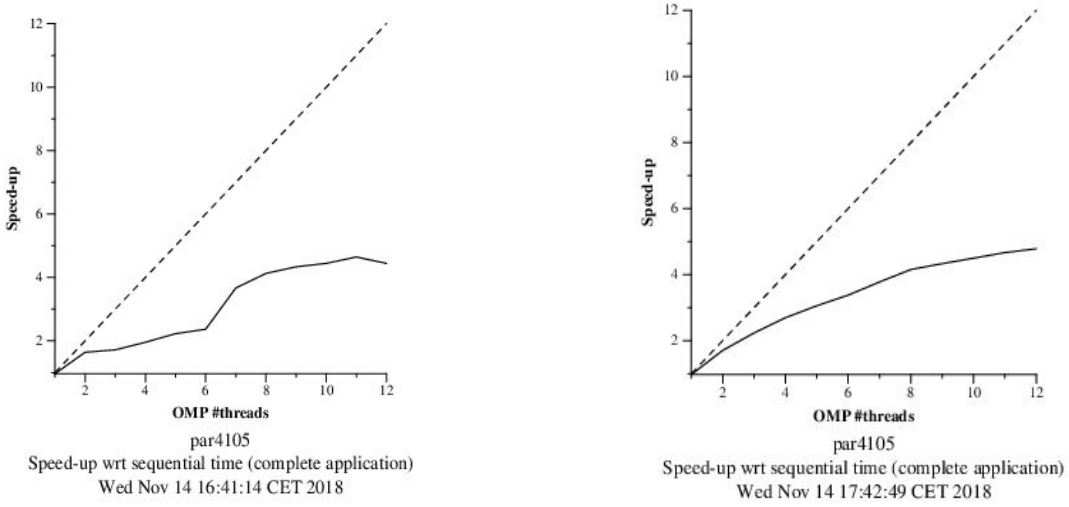


Figure 3.20 On the left we can see the plot for the complete application of the tree version, whereas on the right, we can see the plot obtained for the tree version with a cut-off mechanism

By implementing a cut-off mechanism we ensure not many tasks are created and left unexecuted for a long period of time. That could be one of the reasons that causes the complete application speed-up plot to increase more secuentialy than its predecessor, without a cut-off mechanism.

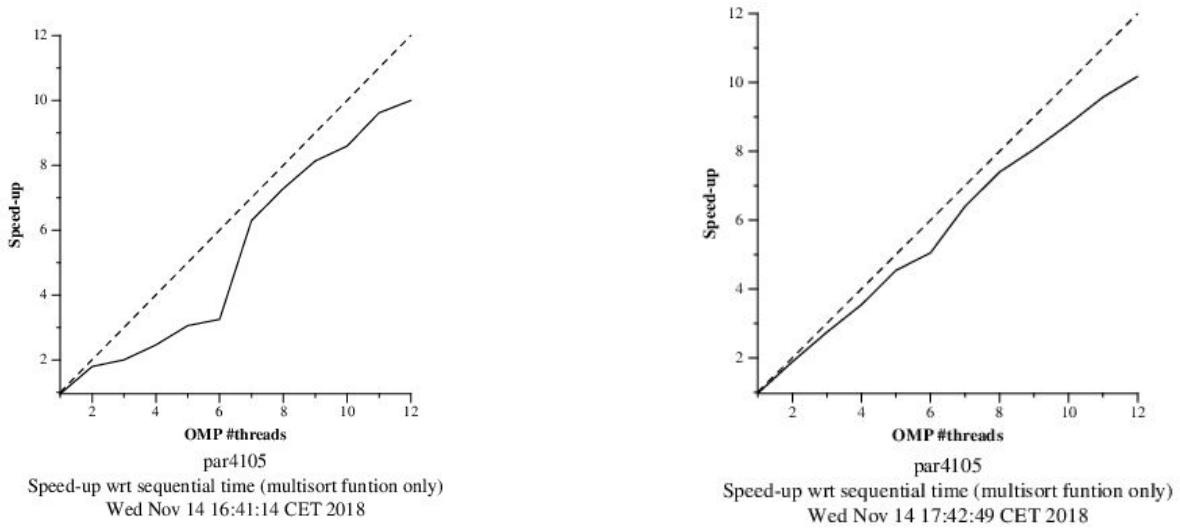


Figure 3.21 On the left we can see the plot for the multisort execution of the tree version, whereas on the right, we can see the plot obtained for the multisort execution with a cut-off mechanism

Although there is no huge difference while using more than ten threads -the plot is near the linear imaginary plot, the distinction comes while executing the program with a smaller number of threads. The plot on the left, that does not implement a cut-off mechanism, is far more of the linear line than the plot that implements a cut off mechanism. This can bring us to conclude that for less number of threads available, the cut-off mechanism has more impact.

Parallelization and performance analysis with dependent tasks

The code we have written in order to implement the *merge* function is pretty straightforward because there are not any dependencies between the different merge recursive call functions.

```
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
	}
}
```

Figure 3.22 Modification of the merge function in order to implement a tree parallelization strategy with dependent tasks

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

		#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 task depend(in: data[0], data[n/2L])
		merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

		#pragma omp taskwait

	} else {
			// Base case
			basicsort(n, data);
	}
}
```

Figure 3.23 Modification of the multisort function in order to implement a tree parallelization strategy with dependent tasks

By using depend clause, before scheduling a parallel task, we ensure that the mentioned task waits until the task that depends on finishes its execution. There is a clear dependency between all the multisort recursive calls and merge calls. In concrete, there is a dependency between the first two calls to multisort with the first merge and also, there is another dependency between the following two calls -the third and the fourth- of the multisort function and the second merge call.
The dependencies are solved by including the correct depend directives, detailing the variables that causes the dependencies inside the directive.

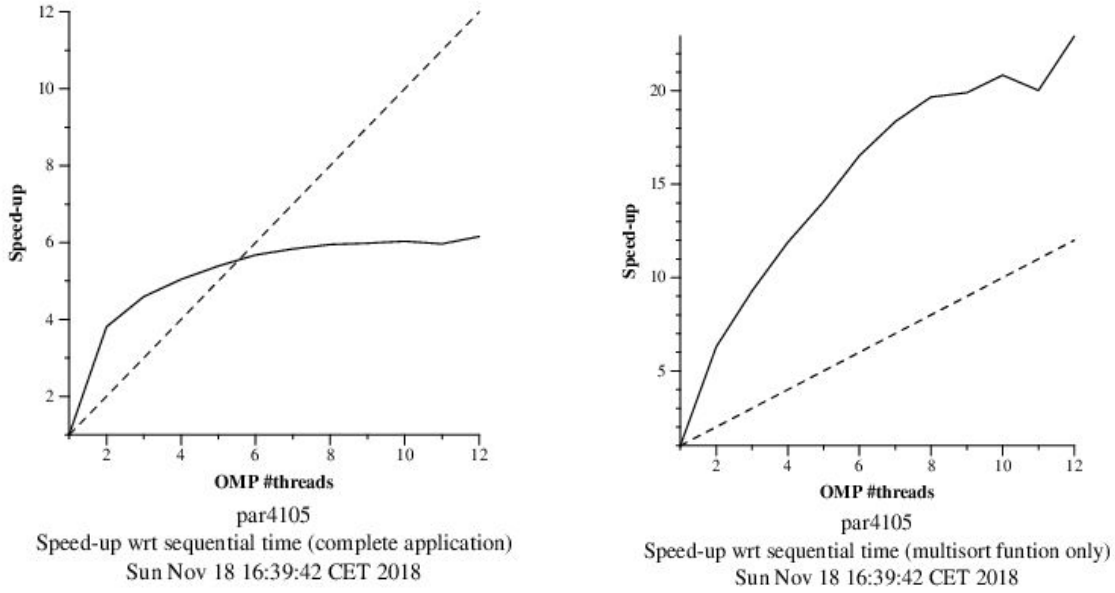


Figure 3.24 On the left we can see the plot for the complete application of the tree version with dependent task, whereas on the right, we can see the plot obtained after analysing only the execution time of the multisort function

The global speed-up of the program can be seen in the left of the Figure 3.24. The improvement for small number of threads available is one of the best performance but, when we increase the number, the speed-up becomes almost linear, achieving better performance than the leaf version but worst compared with the other tree versions. Although this bad result, if we only analyze the *multisort* function, we obtain, by far, the best speed-up in comparison of previous versions. Comparing, for example, this result with the one obtained analyzing the tree with cut-off mechanism version, taking twelve threads as a reference, we can see that the speed up of this version has its speed-up multiplied by a factor of two.

Results obtained by analyzing the trace, produced by submitting the program with the corresponding script, are shown in Figure 3.25 and, more in detail, in Figure 3.26.
By far this version is the one that needs more linear execution time. *Depend* clause has to detail every dependency of the code into a hash table, causing a lot of overheads during its execution.
The analysis of the dependencies is done before any parallel task is executed, which it is the first and large blue line executed in Thread 1.1.1.

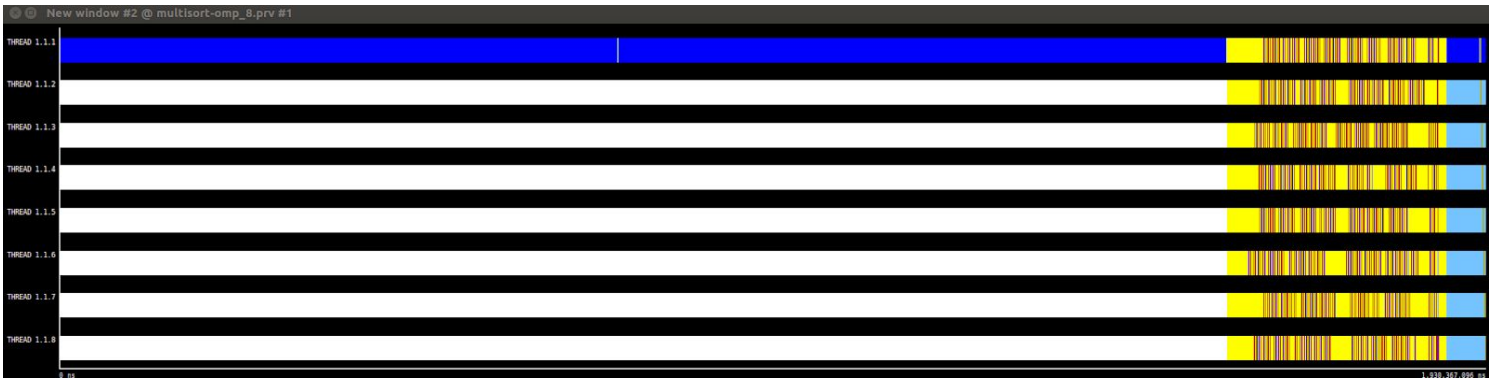


Figure 3.25 Complete paraver result obtained by submitting the tree version with submit-omp.sh script

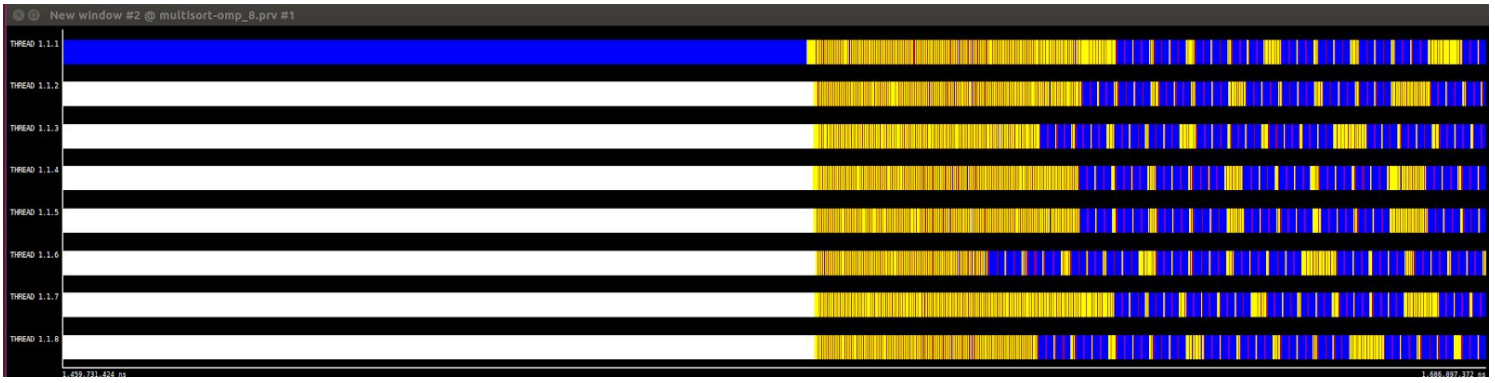


Figure 3.26 Zoom in of the paraver result

Scalability analysis for the Tree version on all the different node types available in boada

In order to study, how the different boada nodes effects on the performance of the multisort, we submitted the tree version on the three different nodes, adapting the *submit-strong-omp.sh* script where necessary. The different values of the variable NMAX, that controls how many processors are available while running the simulation, can be found in Figure 3.26.

Name of the queue	Boada Node	Number of cores per socket	NMAX value
execution	2	6	6
cuda	5	6	6
execution2	7	8	8

Figure 3.26 Table with the different values we stated for the NMAX variable in function of the different nodes of boada.

By doing so, we had obtained the following plots. In order to write a good comparison study in Figure 3.27 we will find the plots we had obtained for the tree version, whereas in following figures the results obtained for this particular optional would be found.

Optional 1: Complete your scalability analysis for the Tree version on the other node types in boada. Remember that the number of cores is different in boada-1 to 4, boada-5 and boada-6 to 8 as well as the enablement of the hyperthreading capability. Set the maximum number of cores to be used (variable np NMAX) by editing submit-strong-omp.sh in order to do the complete analysis. HINT: consider the number of physical/logical cores in each node type and set np NMAX accordingly

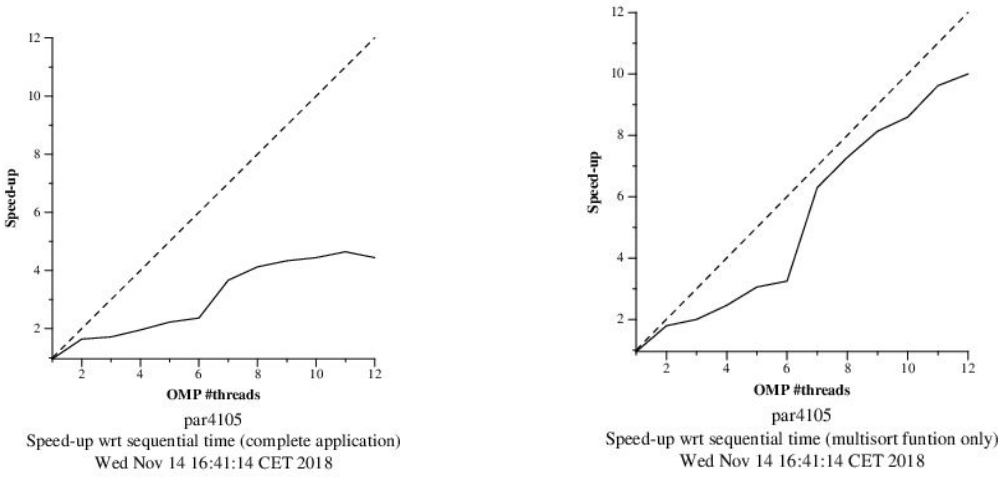


Figure 3.27 Complete speed-up plot and multisort only speed-up plot for the tree version

In Figure 3.28 we can see the results obtained by submitting the code into *execution* queue. Comparing the plots with the ones found in Figure 3.27 it is clear that the complete speed-up is almost the same. The concrete speed-up for the *multisort* function is much better, achieving an speed-up value close to six while using six threads, whereas its predecessor is only able to achieve a speed-up value of, approximately, three.

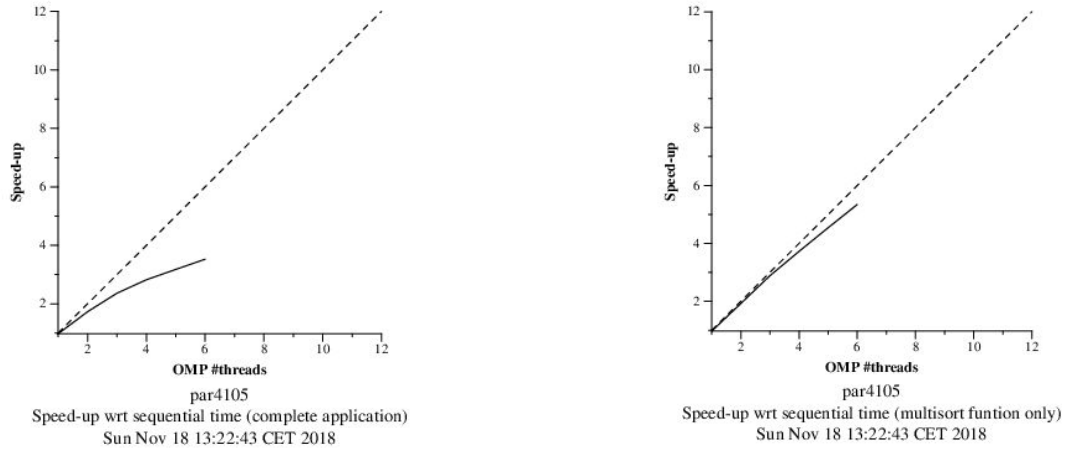


Figure 3.28 Complete speed-up plot and multisort only speed-up plot for the tree version ***submitted in boada-2***

The plot only shows until 6 threads because from *boada2* up to *boada4 there is only six cores per socket.*

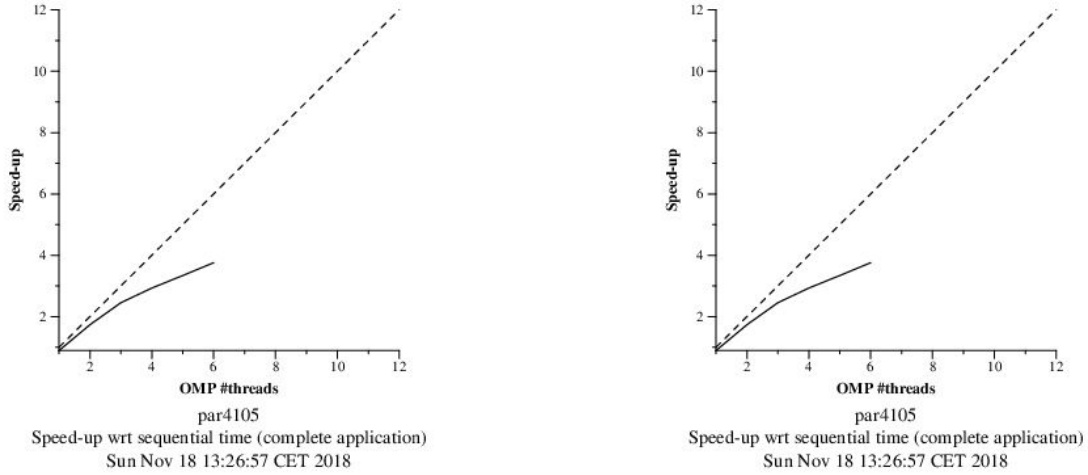


Figure Figure 3.29 Complete speed-up plot and multisort only speed-up plot for the tree version ***submitted in boada-5***

For the fifth boada, we have the same number of sockets per node, what will cause that we are going to obtain a similar result as if we submitted the code into the *execution* queue.

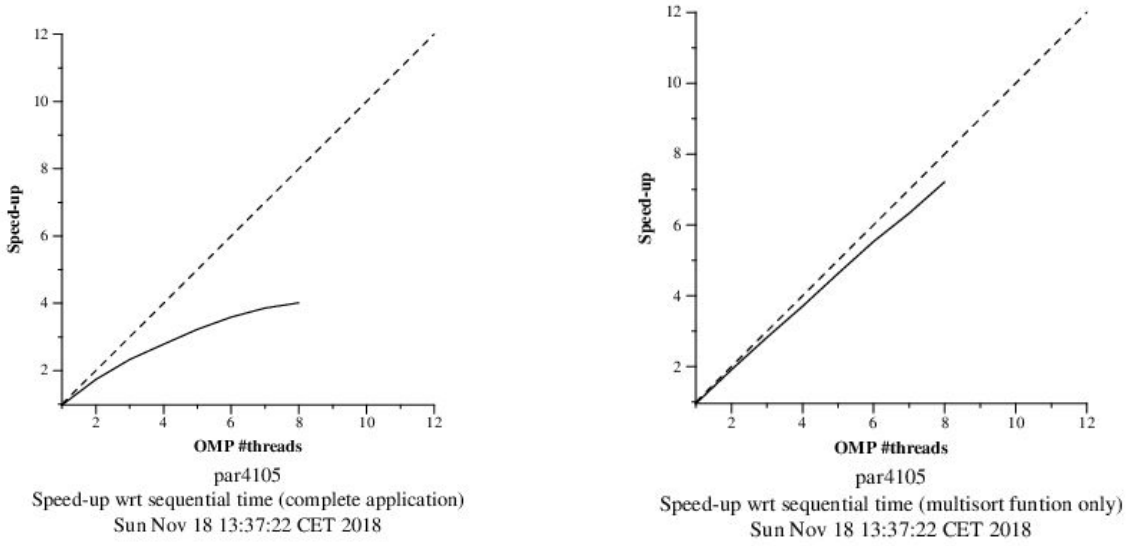


Figure Figure 3.30 Complete speed-up plot and multisort only speed-up plot for the tree version ***submitted in boada-7***

The boada-7 is the one that has more sockets per node: eight in total. As it should be, is the one that we have obtained the highest speed-up value analyzing the *multisort* function only, obtaining a value close to seven, while using eight threads. The general plot achieves almost the same speed-up value. What probably does it mean is that parallelizing the *multisort* and *merge* function does not have a huge impact for the total execution of the program.

Parallelization of the Tree version by parallelizing the two initialize functions

Optional 2: Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors1 . Analyze the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines.

In order to parallelize the initialize function we have used the for clause, that allow us to parallelize a for loop. We have done the same in order to parallelize the clear function, that initialize all the positions of the tmp vector to zero.

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

Figure 3.31 Parallelization of initialize function using for clause

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

Figure 3.32 Parallelization of clear function using for clause

First, we have compared the initialization timing results while executing interactively the code. The results can be found in Figure 3.33, for the tree version without parallelizing the initialization, and in Figure 3.34, for the tree version with a parallelization of the initialization.

```
par4105@boada-1:~/lab3$ ./multisort-omp
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=4
Initialization time in seconds: 0.819058
Multisort execution time: 0.446830
data IS ordered; Check sorted data execution time: 0.021037
Multisort program finished
```

Figure 3.33 Timing results obtained after executing interactively the program, without parallelizing the initialization functions

```
par4105@boada-1:~/lab3$ ./multisort-omp
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
                       CUTOFF=4
Initialization time in seconds: 0.068523
Multisort execution time: 0.386521
data IS ordered; Check sorted data execution time: 0.025114
Multisort program finished
```

Figure 3.34 Timing results obtained after executing interactively the program, parallelizing the initialization functions

By parallelizing this two functions we get an speed-up close to twelve. Multisort execution time is less in the second version, as shown in Figure 3.34. Although that, we think this little reduction does not come as a result of this modification we have implemented rather than other aspects that can not be controlled such as workload of the node, scheduling, etc.

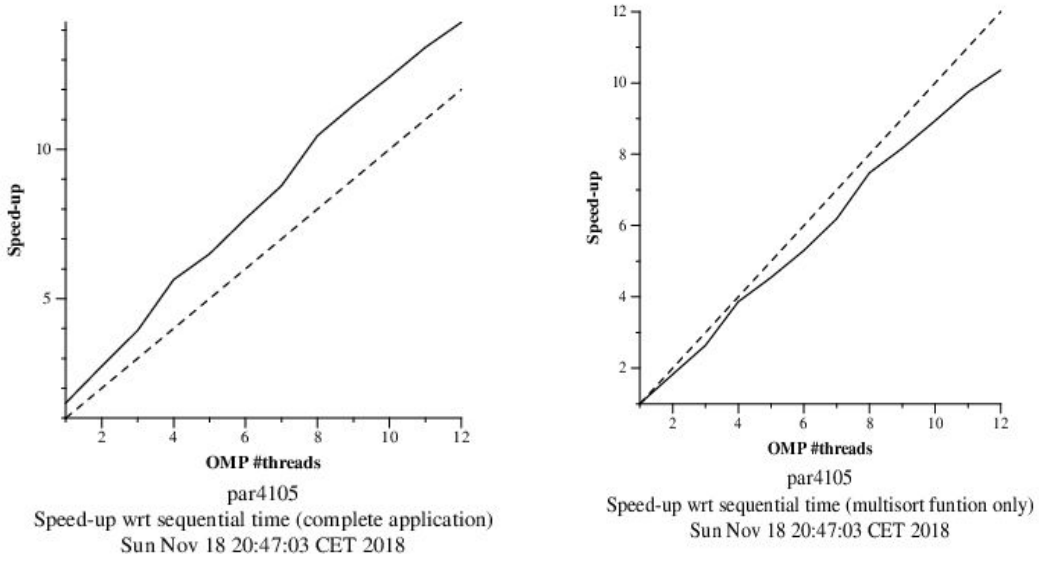
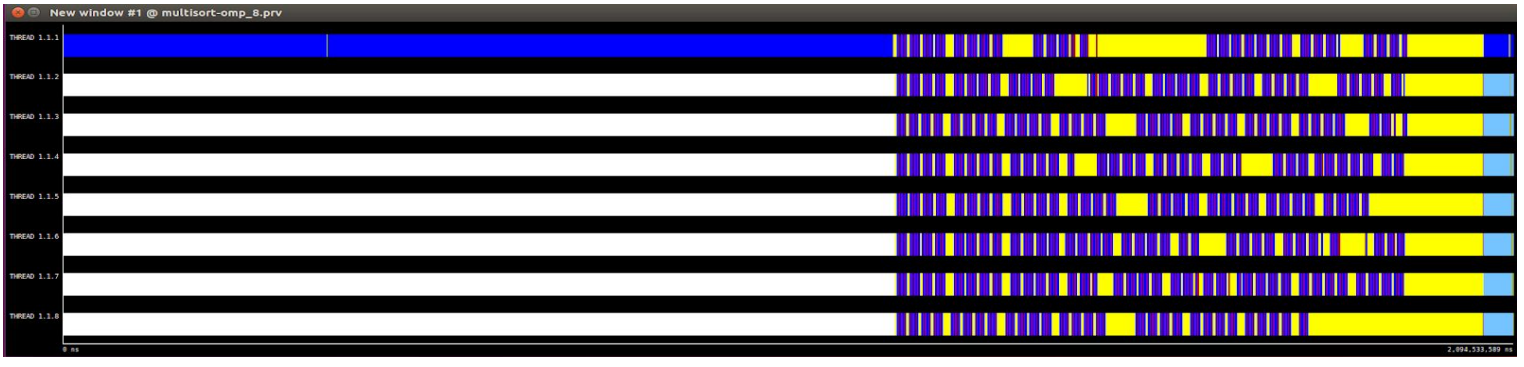


Figure 3.35 Complete application speed-up plot and speed-up of multisort function only.

By parallelizing the two initialization functions we obtain a faster speedup for the complete application, because we are reducing the portion of the code that has to be executed sequentially.

The first tareador result shown in Figure 3.36 is the one that we have obtained in the first version of the tree, while the second result is the tareador result obtained after coding a parallelization of the two different initialization functions.



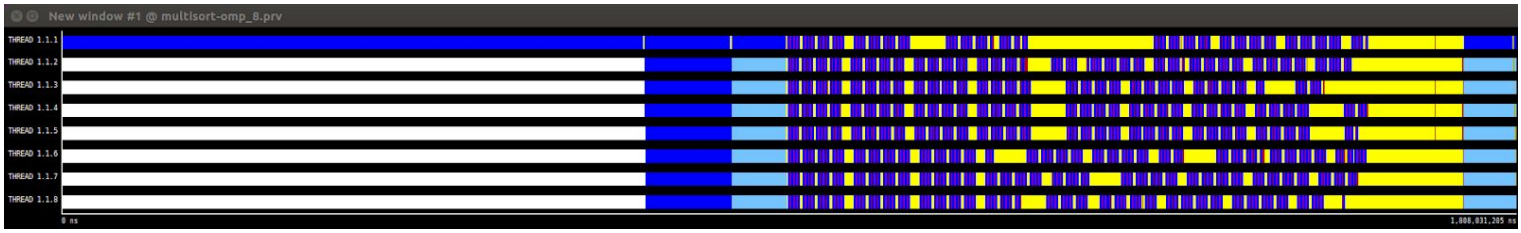


Figure 3.36 The first result is the complete paraver result, obtained by submitting the tree version, without the parallelization of the initialation functions, with submit-omp.sh script. The second result is the complete paraver result with the parallelization of the initialization functions

As it is clear, by doing this optimization, we are able to reduce the sequential time the code needs to be executed.

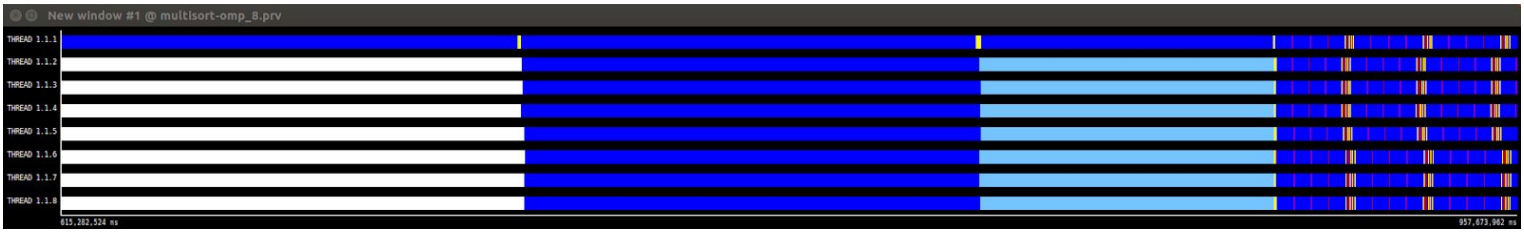


Figure 3.37 Zoom in of the paraver result after doing a parallelization the initialization functions

Conclusion

Before finishing this document, we want to share some conclusions about the study we have carried with *multisort* and *merge* functions. In this report, we have studied some strategies in order to implement good parallelization into our code. As we know, implementing a good parallelization is crucial to improve our execution timings. we have focused into the merge sort algorithm, a basic one that uses a divide and conquer approach to achieve good performance.

We can conclude that the tree parallelization strategy is the best, compared with the leaf strategy, if our recursive function does not have an intensive task at its basic case. Also implementing a cut-off mechanism, and testing our code before choosing the right value, is a must.

Also we have to take into a consideration the architecture that will have the computer that is going to run our code, because will influence positively or negatively to the overall performance.

Although it is important to implement a parallelization strategy to a portion of the code, we have to take into account the part that is not being parallelized and study how could be improved, as we have seen doing the second optional.