Fourth deliverable

Álvaro Martínez Arroyo

Daniel García Romero

par2303

Fall 2015-2016

INDEX

1. Analysis with Tareador

1. Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.

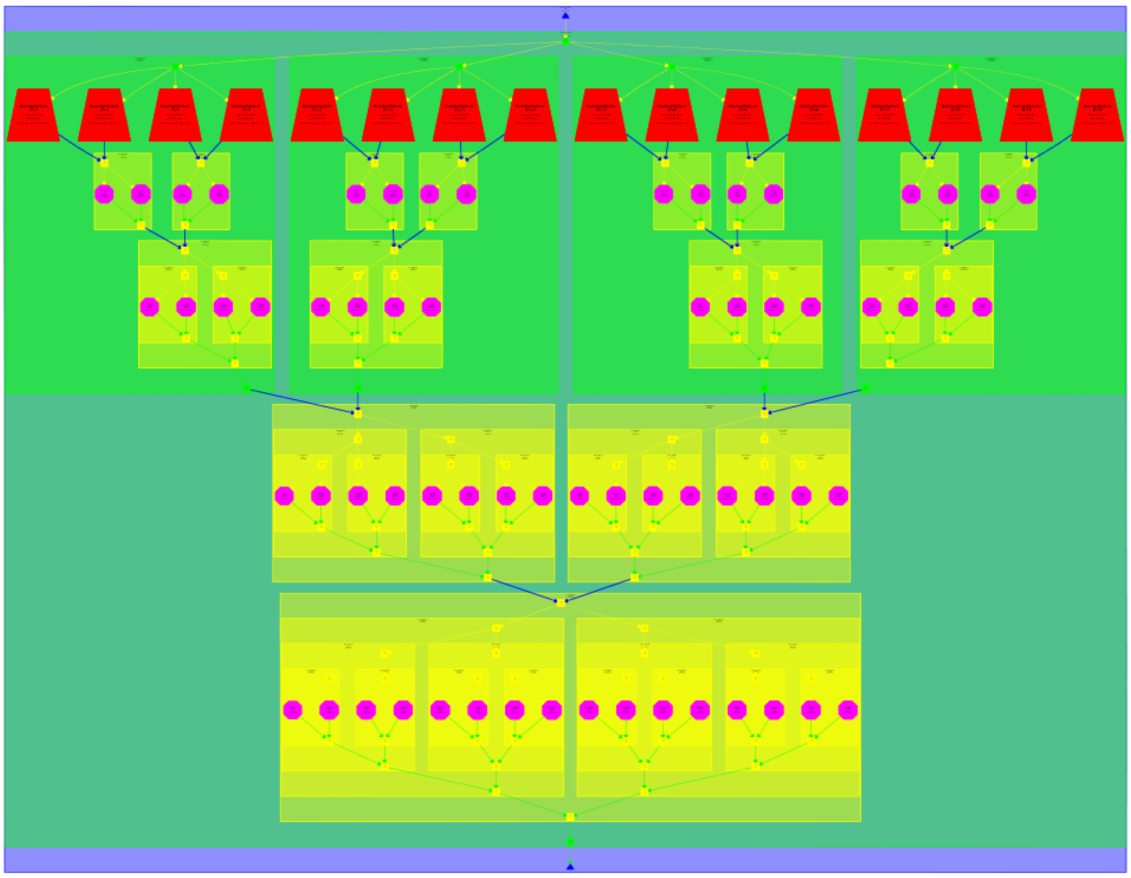


Figure 1. Dependences of multisort-tareador task graph

In the multisort function, recursive decomposition is done by dividing the *data* vector into four parts and each of these parts in other four parts, until we reach leafs (base case), where we call the basicsort function. During the recursive decomposition, we also merge the sorted parts two by two.

In order to see the potentially parallelizable parts of the code, we have defined a task for each region of code that involves the recursive decomposition or the base case.

The dependences that appear in the task graph of Figure 1 are caused by the parts of *data* vector, because they must be sorted before we can merge them.

2. Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

Number of processors	Execution time	Speed-up
1	20,334,421 ns	1.0000
2	10,173,679 ns	1.9987
4	5,087,963 ns	3.9966
8	2,548,394 ns	7.9793
16	1,289,949 ns	15.7637
32	1,289,949 ns	15.7637
64	1,289,949 ns	15.7637

Figure 2. Table with execution time and speed-up for multisort-tareador

Using more processors, we obtain less execution time and better speed-up, unless we use 16 or more processors. In that case, as we can see in the figure 2, we get the maximum parallelization and we will not obtain neither better execution time nor better speed-up.

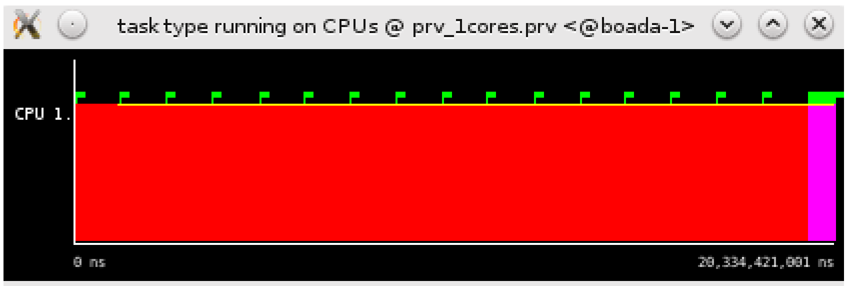


Figure 3. Paraver trace for multisort-tareador with 1 thread

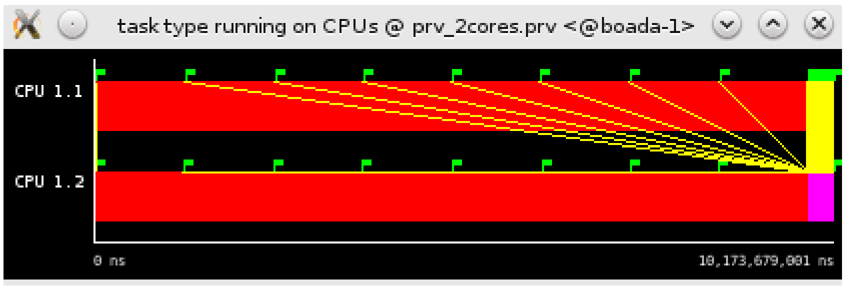


Figure 4. Paraver trace for multisort-tareador with 2 threads

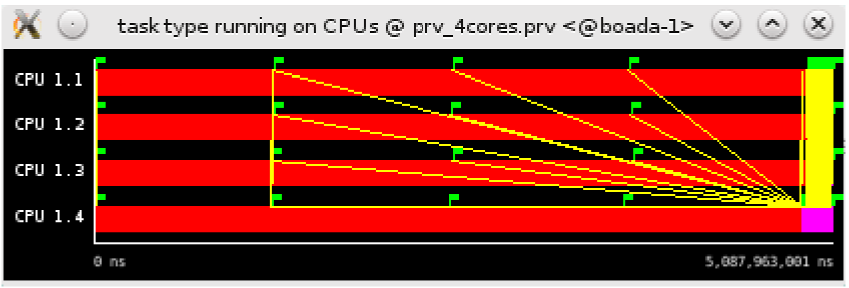


Figure 5. Paraver trace for multisort-tareador with 4 threads

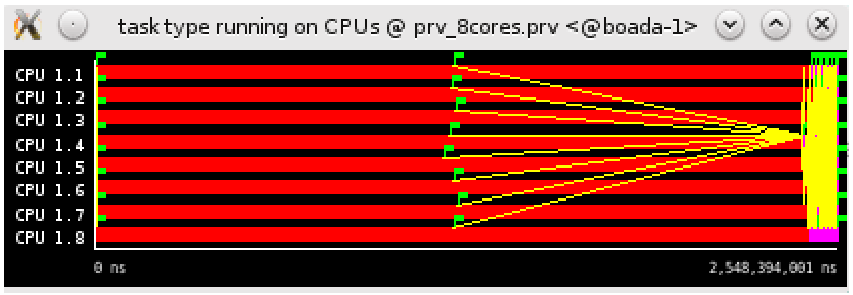


Figure 6. Paraver trace for multisort-tareador with 8 threads

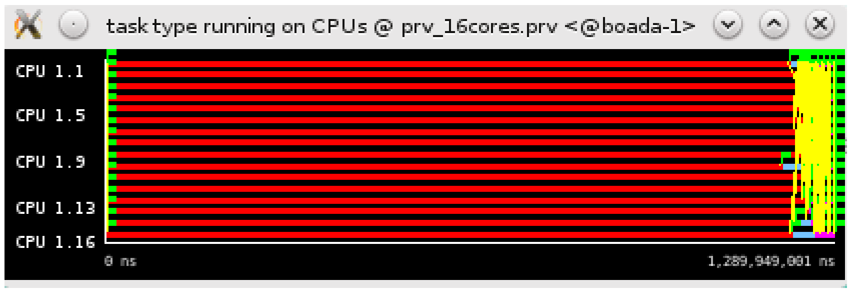


Figure 7. Paraver trace for multisort-tareador with 16 threads

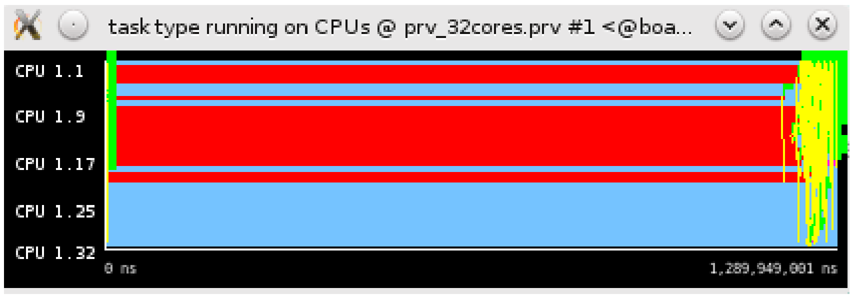


Figure 8. Paraver trace for multisort-tareador with 32 threads

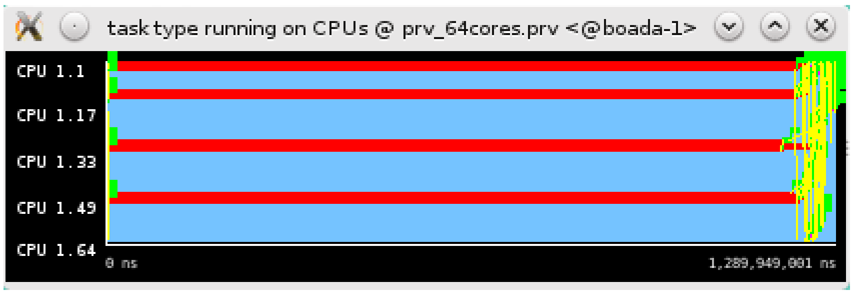


Figure 9. Paraver trace for multisort-tareador with 64 threads

2. Parallelization and performance analysis with tasks

1. Include the relevant portion of the codes that implement the two versions (*Leaf* and *Tree*), commenting whatever necessary.

Leaf version

The *#pragma omp parallel* creates the parallel region, the threads and one task for each thread and the *#pragma omp single* indicates that only one of these created threads will call the multisort function.

In the multisort function, we add the directive *#pragma omp task* in the base case in order to create a task that will call the basicsort function and we add the *#pragma omp taskwait* after the recursive multisort calls because we want to guarantee that the merge calls will be made correctly, so these calls need the vector *data* values calculated in multisort. The last merge call, by the same reason as before, has to wait for the two previous merge functions have been executed.

In the merge function, we add the directive *#pragma omp task* in the base case in order to create a task that will call the basicmerge function.

Tree version

The *#pragma omp parallel* creates the parallel region, the threads and one task for each thread and the *#pragma omp single* indicates that only one of these created threads will call the multisort function.

In the multisort function, we assign a task for each call to multisort and merge functions with *#pragma omp task* and we add *#pragma omp taskwait* before the first and the last merge call and after the last merge call to ensure that the previous calls have been finished.

In the merge function, we assign a task for each auto-invocation with *#pragma omp task* and we add *#pragma omp taskwait* after the second recursive merge call to ensure that the previous calls have been finished.

It is also worth mentioning that in the leafs (base case) of both functions, unlike the *Leaf* version, we don't define a task for the calls to basicsort and basicmerge functions once the recursive divide decomposition stops.

2. For the *Leaf* and *Tree* strategies, include the speed-up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

Steps to obtain the plots and the Paraver traces:

1. In the multisort-omp.c, we have the code of *Leaf* / *Tree* version.

2. We execute `make` to generate the executable file and we submit it using the submit-omp.sh and submit-strong-omp.sh scripts to obtain the plot. If we want to see it (figure 10), we should execute `gs multisort-omp-strong.ps`.

3. In order to get the Paraver trace, we submit the submit-omp-i.sh script, and if we want to see it (figures 11 and 12), we should execute `wxparaver` and load the trace.

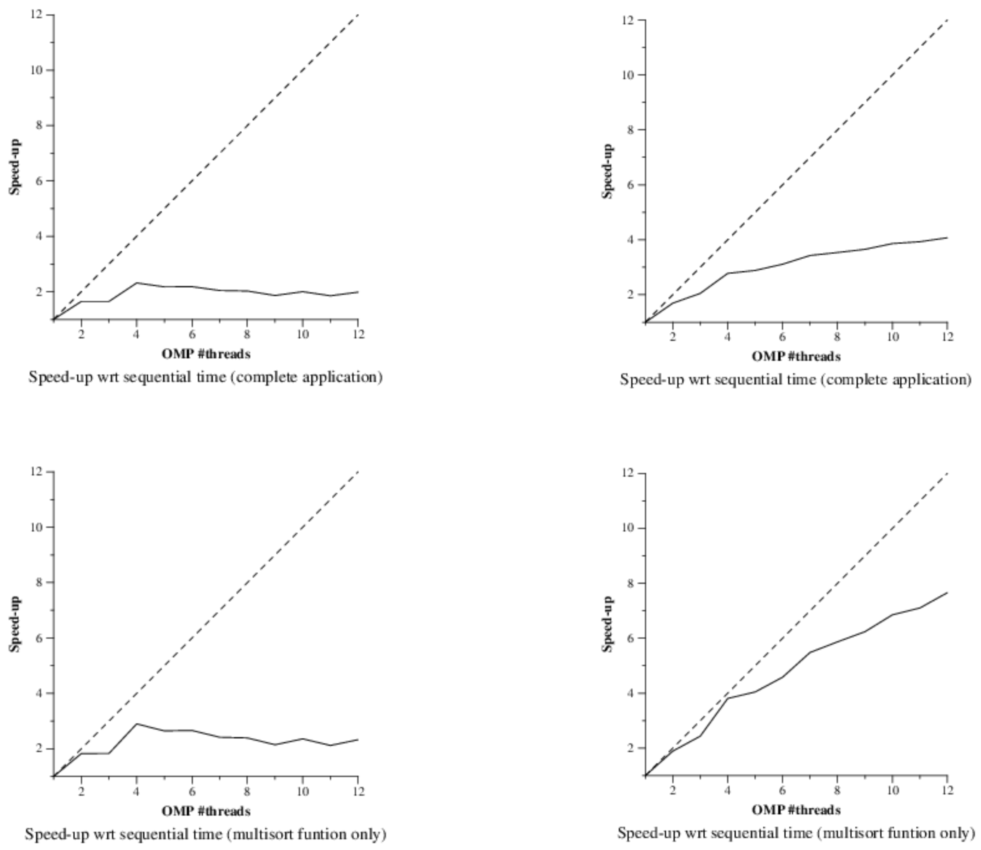
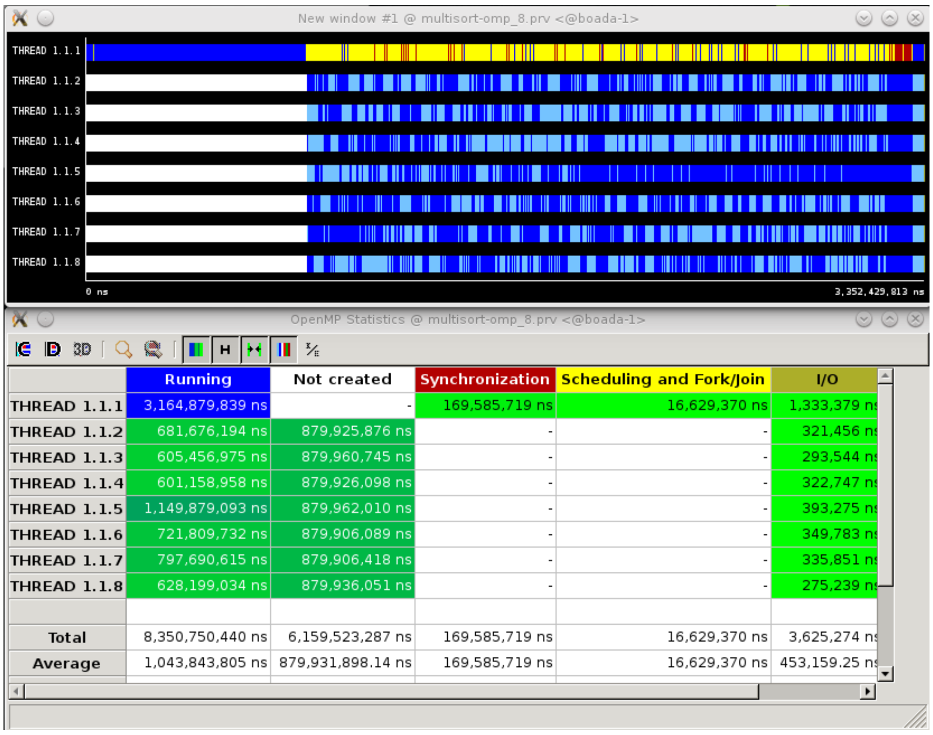
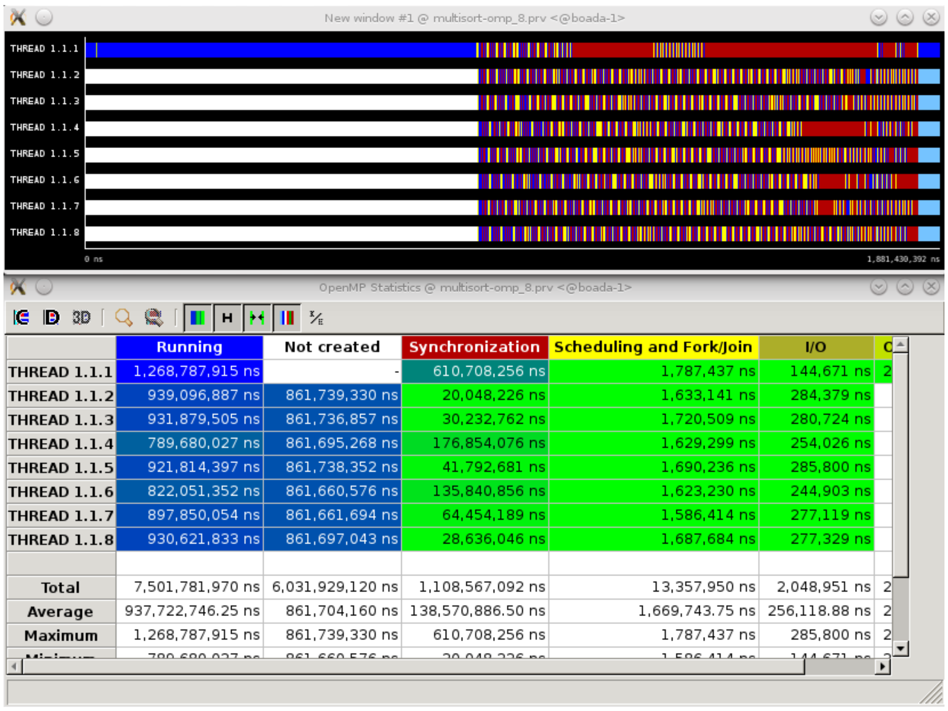


Figure 10. Speed-up plots of Leaf and Tree versions, respectively



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	3,164,879,839 ns	-	169,585,719 ns	16,629,370 ns	1,333,379 n
THREAD 1.1.2	681,676,194 ns	879,925,876 ns	-	-	321,456 n
THREAD 1.1.3	605,456,975 ns	879,960,745 ns	-	-	293,544 n
THREAD 1.1.4	601,158,958 ns	879,926,098 ns	-	-	322,747 n
THREAD 1.1.5	1,149,879,093 ns	879,962,010 ns	-	-	393,275 n
THREAD 1.1.6	721,809,732 ns	879,906,089 ns	-	-	349,783 n
THREAD 1.1.7	797,690,615 ns	879,906,418 ns	-	-	335,851 n
THREAD 1.1.8	628,199,034 ns	879,936,051 ns	-	-	275,239 n
Total	8,350,750,440 ns	6,159,523,287 ns	169,585,719 ns	16,629,370 ns	3,625,274 n
Average	1,043,843,805 ns	879,931,898.14 ns	169,585,719 ns	16,629,370 ns	453,159.25 n

Figure 11. Paraver trace for Leaf version



New window #1 @ multisort-omp_8.prv <@boada-1>

OpenMP Statistics @ multisort-omp_8.prv <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	C
THREAD 1.1.1	1,268,787,915 ns	-	610,708,256 ns	1,787,437 ns	144,671 ns	2
THREAD 1.1.2	939,096,887 ns	861,739,330 ns	20,048,226 ns	1,633,141 ns	284,379 ns	
THREAD 1.1.3	931,879,505 ns	861,736,857 ns	30,232,762 ns	1,720,509 ns	280,724 ns	
THREAD 1.1.4	789,680,027 ns	861,695,268 ns	176,854,076 ns	1,629,299 ns	254,026 ns	
THREAD 1.1.5	921,814,397 ns	861,738,352 ns	41,792,681 ns	1,690,236 ns	285,800 ns	
THREAD 1.1.6	822,051,352 ns	861,660,576 ns	135,840,856 ns	1,623,230 ns	244,903 ns	
THREAD 1.1.7	897,850,054 ns	861,661,694 ns	64,454,189 ns	1,586,414 ns	277,119 ns	
THREAD 1.1.8	930,621,833 ns	861,697,043 ns	28,636,046 ns	1,687,684 ns	277,329 ns	
Total	7,501,781,970 ns	6,031,929,120 ns	1,108,567,092 ns	13,357,950 ns	2,048,951 ns	2
Average	937,722,746.25 ns	861,704,160 ns	138,570,886.50 ns	1,669,743.75 ns	256,118.88 ns	2
Maximum	1,268,787,915 ns	861,739,330 ns	610,708,256 ns	1,787,437 ns	285,800 ns	2
Minimum	789,680,027 ns	861,660,576 ns	20,048,226 ns	1,586,414 ns	144,671 ns	2

Figure 12. Paraver trace for Tree version

In the *Leaf* version, we create the tasks in the leafs, and as a consequence of the recursive calls, the tasks are waiting on the finalization of their child tasks, so the further away we are from the leafs, more time we will have to wait, and this causes that we won't obtain a good speed-up as compared to the *Tree* version, where we create a task for each recursive call and each task creates other tasks until we reach leafs, so the speed-up obtained is better because the threads create tasks and they only have to wait the tasks that each one has generated.

3. Analyze the influence of the recursivity depth in the *Tree* version, including the execution time plot, when changing the recursion depth and using 8 threads. Reason about the behavior observed. Is there an optimal value?

We have submitted the submit-depth-omp.sh script, which performs a number of executions changing the recursion depth using 8 threads and generates a plot showing how the execution time changes with that parameter. If we want to see it (figure 13), we should execute `gs multisort-omp-8-depth.ps`.

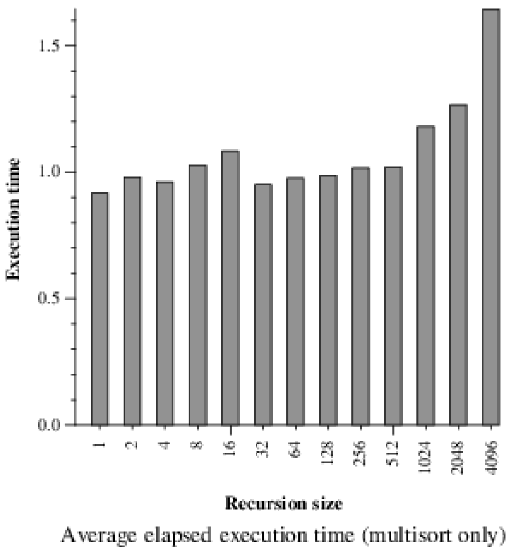


Figure 13. Execution time plot of Tree version

In the *Tree* version, we create a task for each recursive call. The more recursive calls we have, the more tasks are created, and each task has to wait for the tasks created by itself have been executed. Therefore, the more recursive calls we have, the more execution time we obtain.

In our case, the optimal value of recursion size is 1.

3. Parallelization and performance analysis with dependent tasks

1. Include the relevant portion of the code that implements the *Tree* version with task dependencies, commenting whatever necessary.

In order to express dependencies among tasks and avoid some of the *taskwait* synchronizations that we have, we can use the following task definition:

#pragma omp task depend (in: …) depend (out: …)

Using this directive, we are specifying that our task can't be executed until the task that generates the values in *in* finishes. When our task finishes, it notifies other tasks waiting for the values in *out* that it has already finished.

2. Reason about the performance that is observed, including the speed-up plots that have been obtained different numbers of processors and with captures of Paraver windows to justify your reasoning.

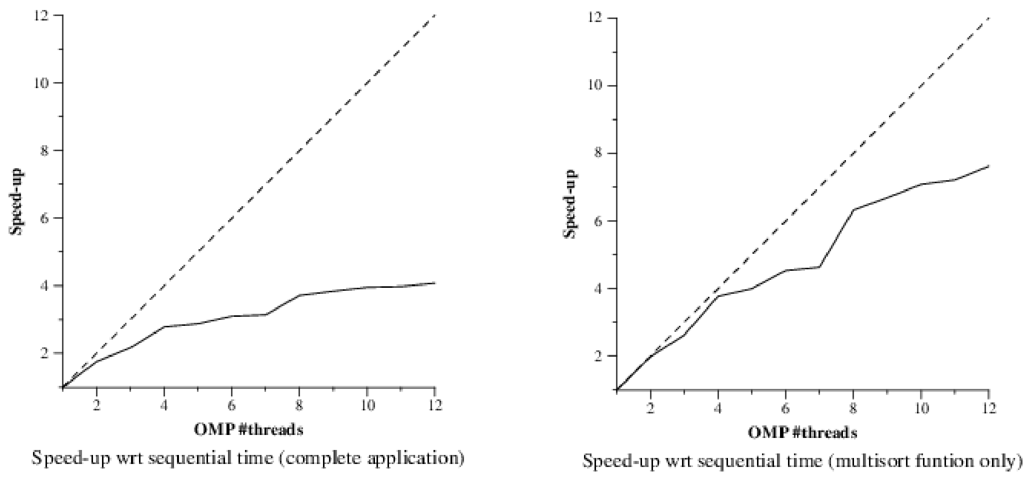
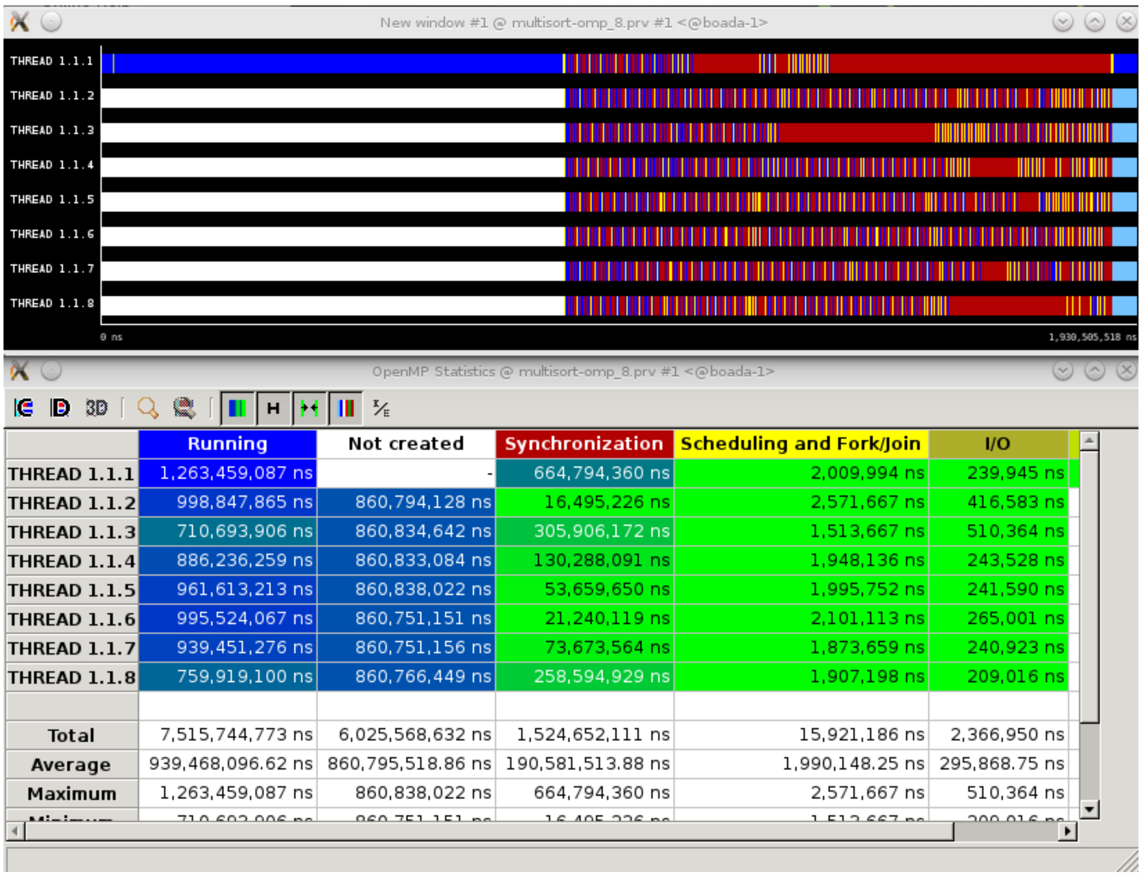


Figure 14. Speed-up plots of Tree version with task dependencies

If we compare the speed-up plots and the Paraver traces of the figures 10, 12 and 14, 15, we can see that the obtained results are very similar, but we think that we should obtain better results with explicit definition of task dependencies because the tasks just have to wait for the tasks that generate their values in *in* have been finished, while in the *Tree* version without explicit definition of task dependencies, the *taskwaits* suspend the tasks waiting on the finalization of their child tasks.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	1,263,459,087 ns	-	664,794,360 ns	2,009,994 ns	239,945 ns
THREAD 1.1.2	998,847,865 ns	860,794,128 ns	16,495,226 ns	2,571,667 ns	416,583 ns
THREAD 1.1.3	710,693,906 ns	860,834,642 ns	305,906,172 ns	1,513,667 ns	510,364 ns
THREAD 1.1.4	886,236,259 ns	860,833,084 ns	130,288,091 ns	1,948,136 ns	243,528 ns
THREAD 1.1.5	961,613,213 ns	860,838,022 ns	53,659,650 ns	1,995,752 ns	241,590 ns
THREAD 1.1.6	995,524,067 ns	860,751,151 ns	21,240,119 ns	2,101,113 ns	265,001 ns
THREAD 1.1.7	939,451,276 ns	860,751,156 ns	73,673,564 ns	1,873,659 ns	240,923 ns
THREAD 1.1.8	759,919,100 ns	860,766,449 ns	258,594,929 ns	1,907,198 ns	209,016 ns
Total	7,515,744,773 ns	6,025,568,632 ns	1,524,652,111 ns	15,921,186 ns	2,366,950 ns
Average	939,468,096.62 ns	860,795,518.86 ns	190,581,513.88 ns	1,990,148.25 ns	295,868.75 ns
Maximum	1,263,459,087 ns	860,838,022 ns	664,794,360 ns	2,571,667 ns	510,364 ns
Minimum	710,693,906 ns	860,751,151 ns	16,495,226 ns	1,513,667 ns	209,016 ns

Figure 15. Paraver trace for Tree version with task dependencies