Lab4: Divide and Conquer parallelism with OpenMP: Sorting

Q2 2021-22

Integrantes:
Pol Pérez Castillo - par2116
Cristian Sánchez Estapé - par2118

Fecha de entrega: 10-05-2022

Task decomposition analysis with Tareador

From the task dependence graphs that are generated for leaf and tree, do you observe any major differences in terms of structure, types, number and granularity of tasks...

In terms of structure, the first that must be noted is the creation of the tasks: *leaf strategy* presupposes that a task *only* corresponds to the computation of a base case, which means that one thread will be responsible for the generation of *all* tasks, and the remaining threads of executing them. However, *tree strategy* considers recursive calls as tasks, which means that, during the first iteration, a thread will be responsible for the creation (and part of the computation) of their subsequent tasks. In that manner, *tree strategy* has an explorative nature which is not found in the *leaf strategy*, for the former parallelizes the exploration and computation of the whole structure of the task tree, while the latter only takes into consideration the "main" workload of the program itself.
In regards to the types of tasks, whereas for the *leaf strategy* each task corresponds to the computation of the base case, for the *tree strategy* each task corresponds to the computation of the base case plus the intermediate tasks, that is, the recursive calls to the used functions.
For the number of tasks, we comparatively find that *tree strategy* creates and executes more tasks than *leaf strategy*, for a task corresponds to a recursive call to a function, a feature that includes both base cases along with the creation of tasks.
In terms of granularity, given that both end up executing the same base cases, even though *tree strategy* has a bigger number of tasks to execute, the granularity would stay very much the same for both programs. This granularity ought to change, however, if we took into consideration a different type of task for the *leaf strategy*, for it could very well change the number of base case tasks both strategies compute.
Figures 1 and 2 are graphical representations of all the information stated above.

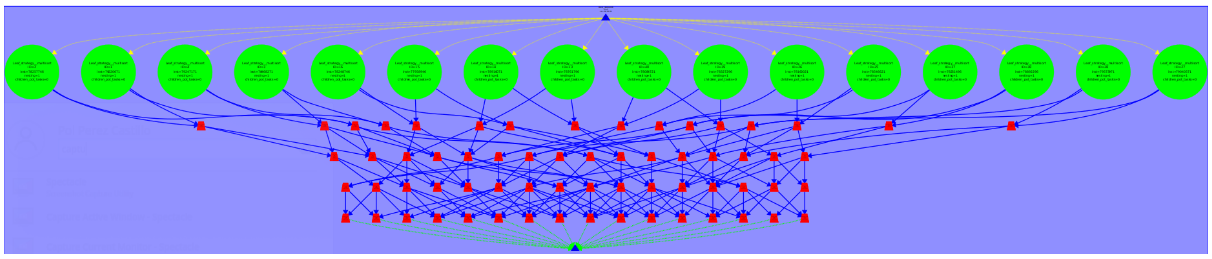


Figure 1: TDG for the leaf strategy

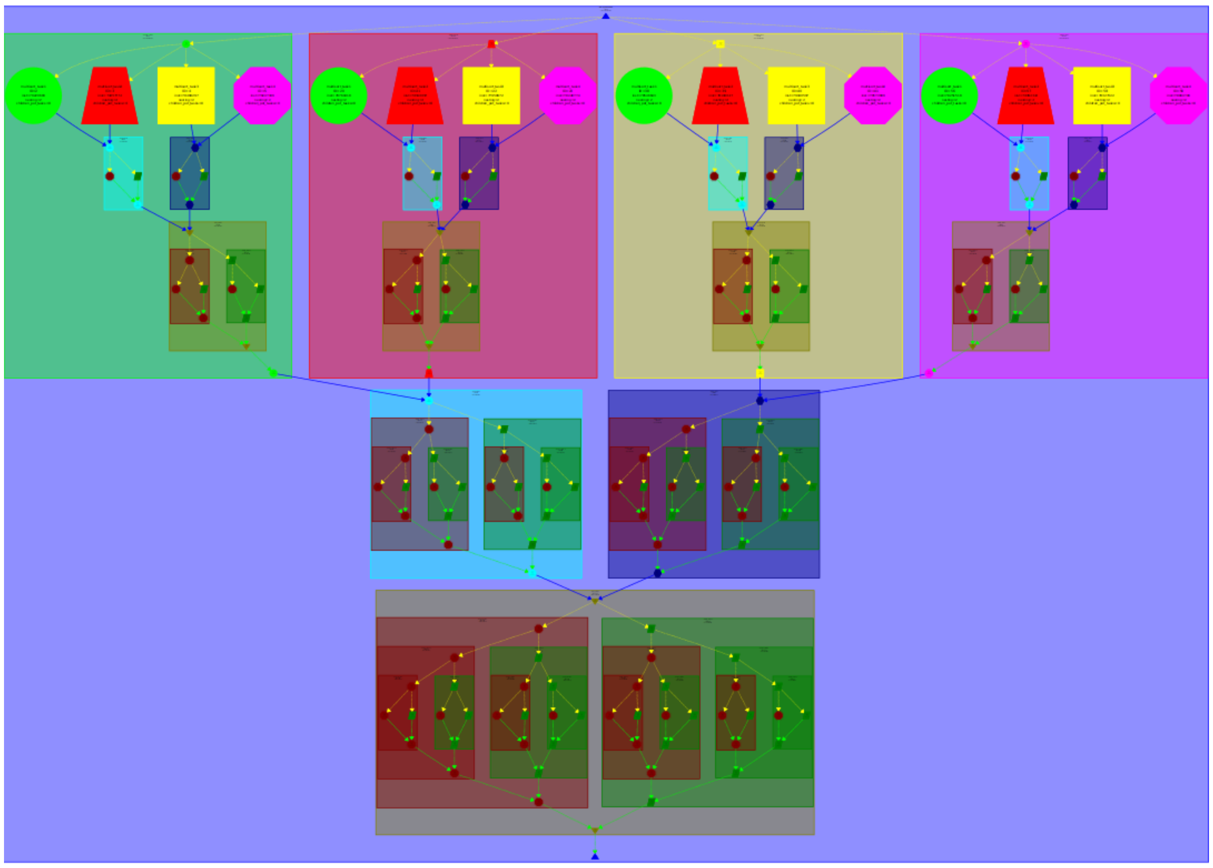


Figure 2: TDG for the tree strategy

Continue the analysis of the task graphs generated in order to identify the task ordering constraints that appear in each case and the causes for them, and the different kind of synchronisations that could be used to enforce them.

In regards to the *leaf strategy*, it is necessary to insert two *taskwait* directives: the first one located after the last call to *multisort*, for all *merge* calls depend on it; the second one would be located after the third and last call to *merge* (in the *multisort* function), for it needs the first two *merge* calls to have finished to sort the vector correctly.
In regards to the *tree strategy*, the two *taskwait* directives previously stated would be located in the exact same place, along with a third *taskwait* directive located after the second and last *merge* call (in *merge* function) to ensure that the function sorts correctly.

From the Paraver windows, do you notice any differences in terms of how and when tasks doing computation are generated?

Regarding the *tree strategy*, in figure 4 it can be observed how the first 4 tasks are created precisely at the same time, along with the fact that these will be the ones from which the subsequent 4 tasks will be recursively generated (colors indicate the relation between the tasks: the first 4 of them indicate their respective childs). It must be stated that this process will take place until each leaf has been finally computed.
As opposed to this strategy, in the *leaf strategy* each leaf is created once the thread (that single-handedly creates all tasks) has reached them, which is the main reason why, in figure 3, threads don't start the computation of leaf tasks at the same time: at the beginning of the execution, there still exist leafs that haven't been reached yet.

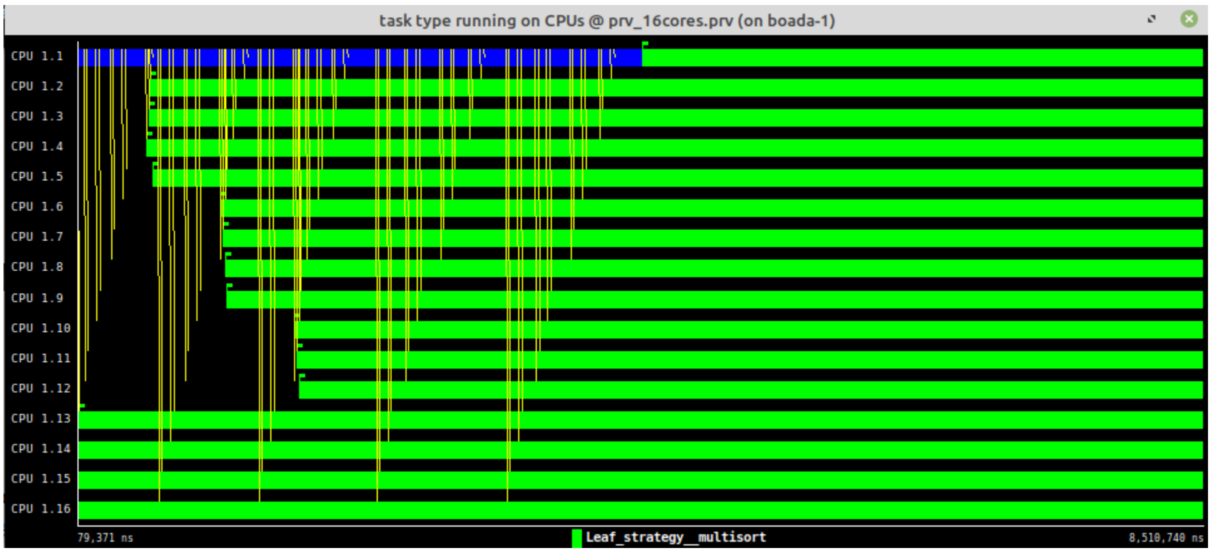


Figure 3: task creation for the leaf strategy

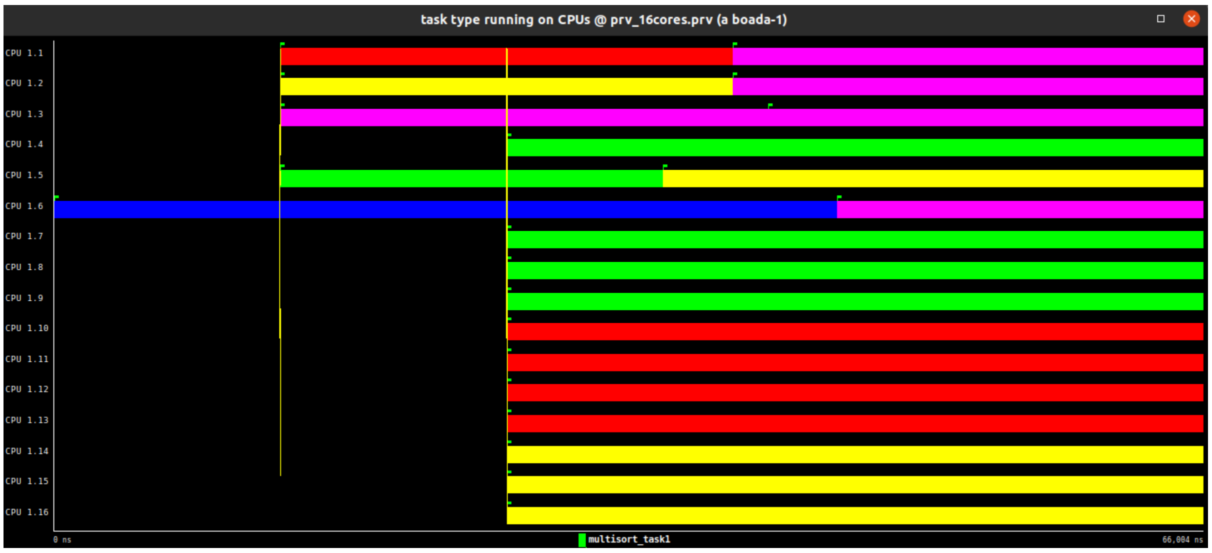


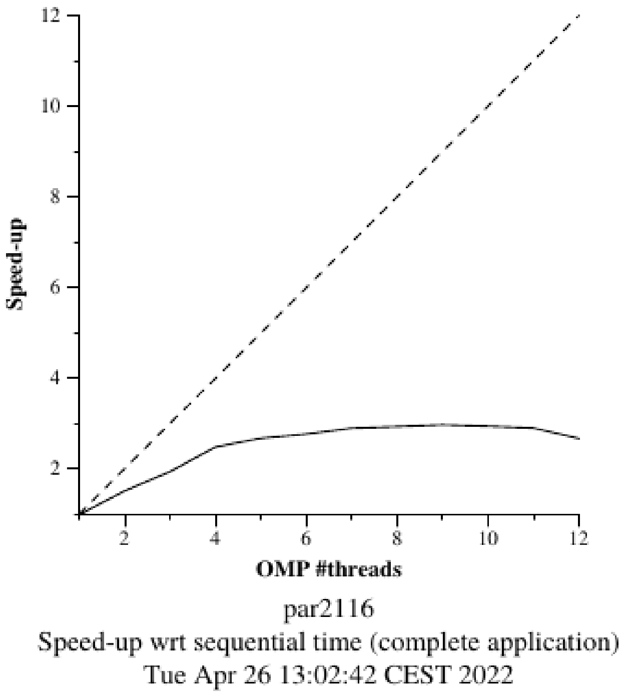
Figure 4: task creation for the tree strategy

Parallelisation with OpenMP tasks

1. Leaf strategy in OpenMP

Analyze the scalability of your parallel implementation by looking at the two speed–up plots. Is the speed–up achieved reasonable?

As it can be seen in figure 5, the obtained strong scalability is far from ideal (both in terms of the execution of the main program as well as the multisort function). The cause of such phenomena is to be found in the fact that there's only one thread exploring the tree task, thus causing inefficiencies in the overall execution. Moreover, increasing the number of threads can't solve the problem by itself given that the increase itself might imply the non-usage of some of them, hence increasing even more the inefficiency. Consequently, everything stated above leads to the conclusion that the obtained speed-up is not what one would expect, meaning that these problems could be solved using a different approach, such as the *tree strategy*.



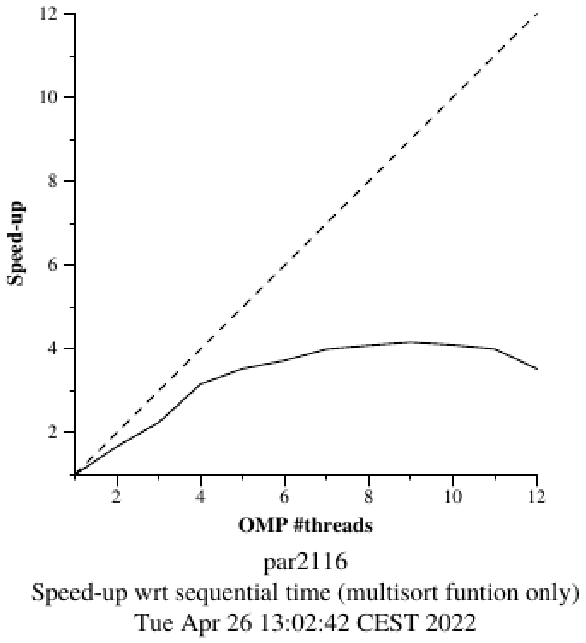


Figure 5: Strong scalability plots of leaf strategy implementation

Try to conclude why your parallel implementation is not scaling as you would expect. For example, is there a big sequential portion in your parallel execution? Is the program generating enough tasks to simultaneously feed all processors? How many tasks simultaneously execute?

The implementation of the *leaf strategy* doesn't scale as one would expect due to its task decomposition: given the fact that this strategy considers a base case as a task, the traversal of the task tree is done by one single thread, which means that, for an unreasonable amount of levels, we'll have *one* thread going over all its different branches to get to their leafs. This implies that, in the end, we'll have a big sequential portion of execution time, which is unavoidable given the nature of the decomposition itself. Moreover, it also means that, if we relate the present number of threads of the execution with the chunk of vector positions one has to sort and merge, this means that, for a big amount of threads, we'll have even more base cases, which means that, if we initially had a little bit of speed-up, now we'll incur in inefficiencies due to the fact that *one* thread that traverses the task tree spends too much time in it. Now, given all of these stated facts, the task generation becomes slow and, thus, threads spend a lot of time waiting for the next task to execute, which translates into a considerable amount of time dedicated to synchronizations. Finally, as for the amount of tasks simultaneously executed, one could argue that it will hardly ever be more than four tasks executing parallely.

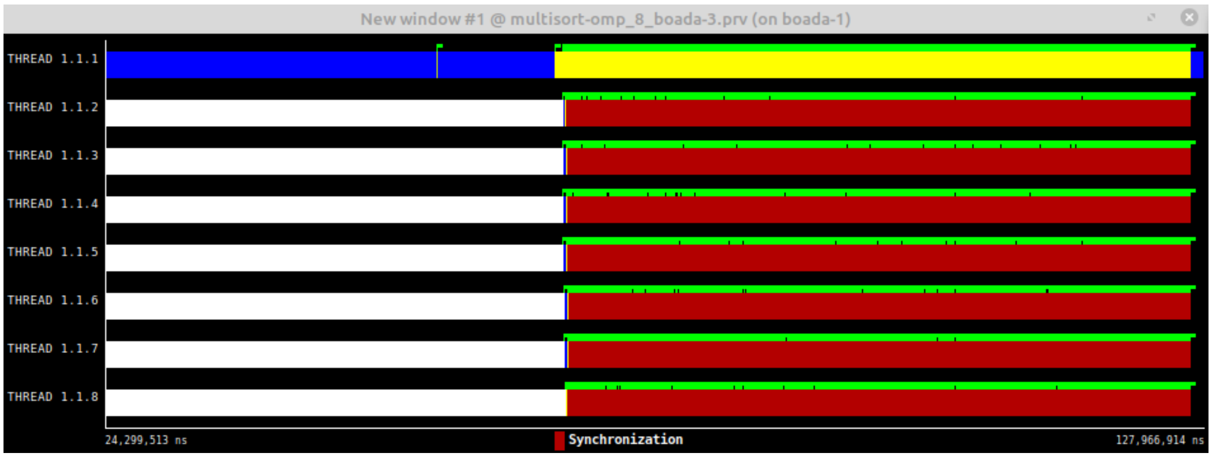
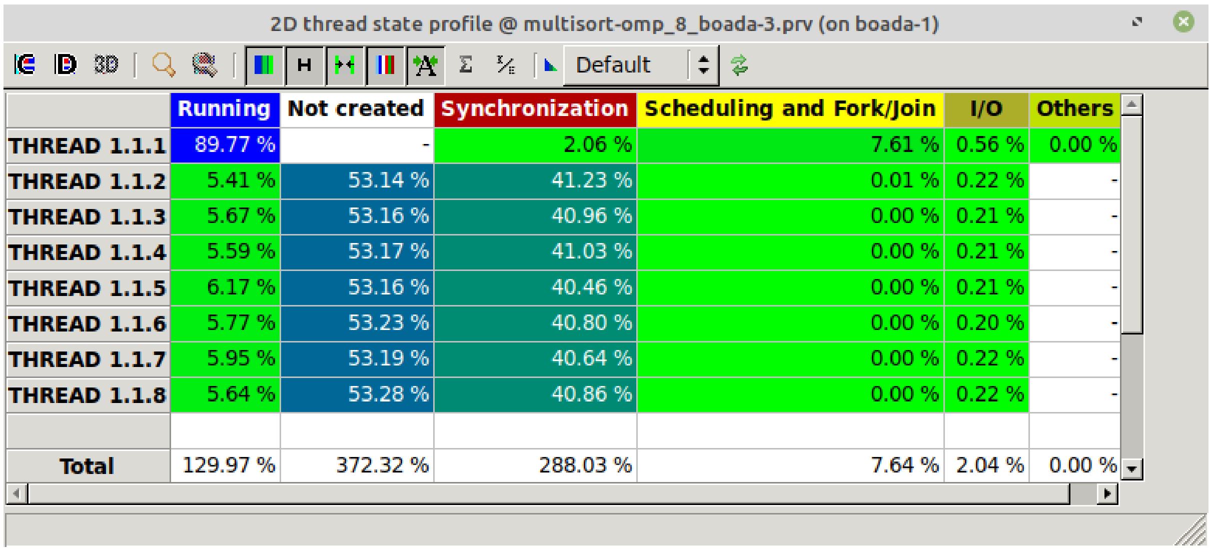


Figure 6: Paraver window for leaf strategy



2D thread state profile @ multisort-omp_8_boada-3.prv (on boada-1)

Default

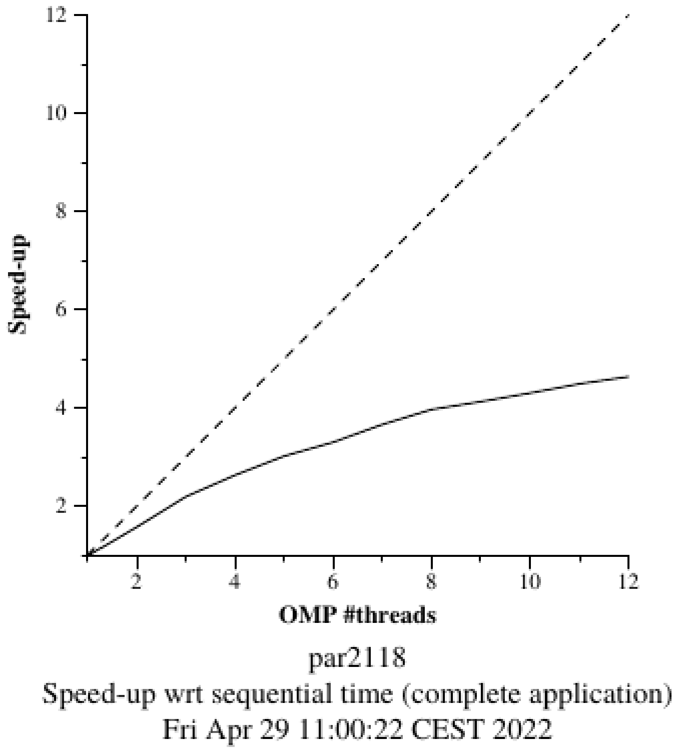
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	89.77 %	-	2.06 %	7.61 %	0.56 %	0.00 %
THREAD 1.1.2	5.41 %	53.14 %	41.23 %	0.01 %	0.22 %	-
THREAD 1.1.3	5.67 %	53.16 %	40.96 %	0.00 %	0.21 %	-
THREAD 1.1.4	5.59 %	53.17 %	41.03 %	0.00 %	0.21 %	-
THREAD 1.1.5	6.17 %	53.16 %	40.46 %	0.00 %	0.21 %	-
THREAD 1.1.6	5.77 %	53.23 %	40.80 %	0.00 %	0.20 %	-
THREAD 1.1.7	5.95 %	53.19 %	40.64 %	0.00 %	0.22 %	-
THREAD 1.1.8	5.64 %	53.28 %	40.86 %	0.00 %	0.22 %	-
Total	129.97 %	372.32 %	288.03 %	7.64 %	2.04 %	0.00 %

Figure 7: Histogram of leaf strategy

2. Tree strategy in OpenMP

3. Analyze the scalability of your parallel implementation by looking at the two speed–up plots. Is the speed–up achieved reasonable?

Just as we had thought, the *tree strategy* achieves a greater speed-up and strong scalability than its counterpart, which is due to the fact that both *multisort* and *merge* functions are being fully parallelized (and not only their base cases). Subsequently, an increase in the number of threads reduces the overall execution time given the fact that the recursive calls are adequately distributed, thus achieving an almost-ideal speed-up.



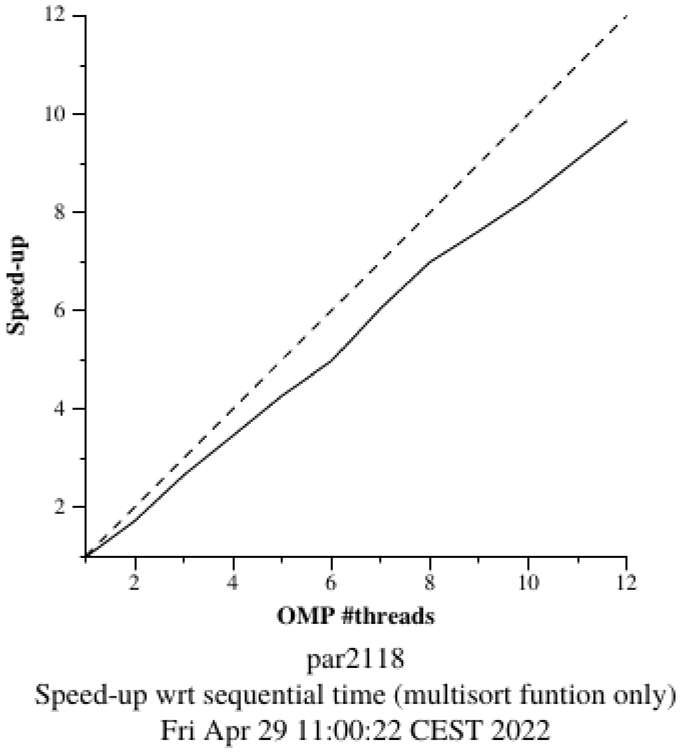


Figure 8: strong scalability plots of tree strategy implementation

4. Is there a big sequential portion in your parallel execution? Is the program generating enough tasks to simultaneously feed all processors? How many tasks simultaneously execute?

With the *tree strategy* implementation, what was once a sequential portion of execution time has now pivoted to a fork portion of execution time: synchronizations have been transmuted into forks, and thus no sequentiality has been maintained (although synchronizations are still needed, of course). Thus, with tree strategy not only no sequentiality has been preserved, but also no thread is found at the expense of any other: all threads spend time in task creation and execution, and thus there is no need to worry about the chances that one thread might be waiting for the creation of a new task (since all of them participate in such endeavor). Finally, now the number of tasks executed simultaneously increases up to 8

(since we have 8 threads making computations), which is comparatively the double as in the *leaf strategy*.
Figure 9 represents the pivoting from synchronizations to forks (bear in mind that, between all those yellow stripes, synchronizations and executions still take place), and figure 10 shows the exact % spent in each type of process. It might be stated that running time has increased, and that synchronizations have decreased in contrast to the *leaf strategy*, although forks and “not creation” now are bigger than they were.

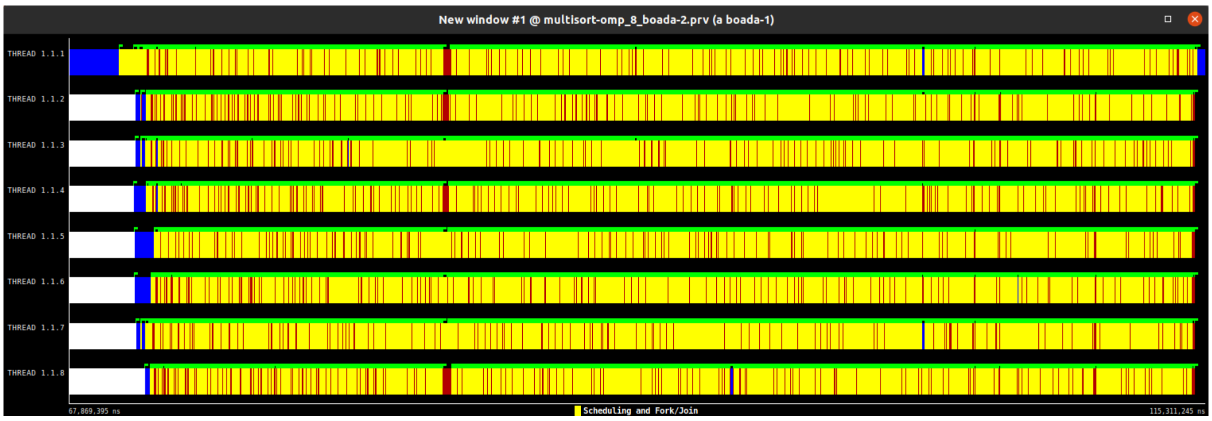
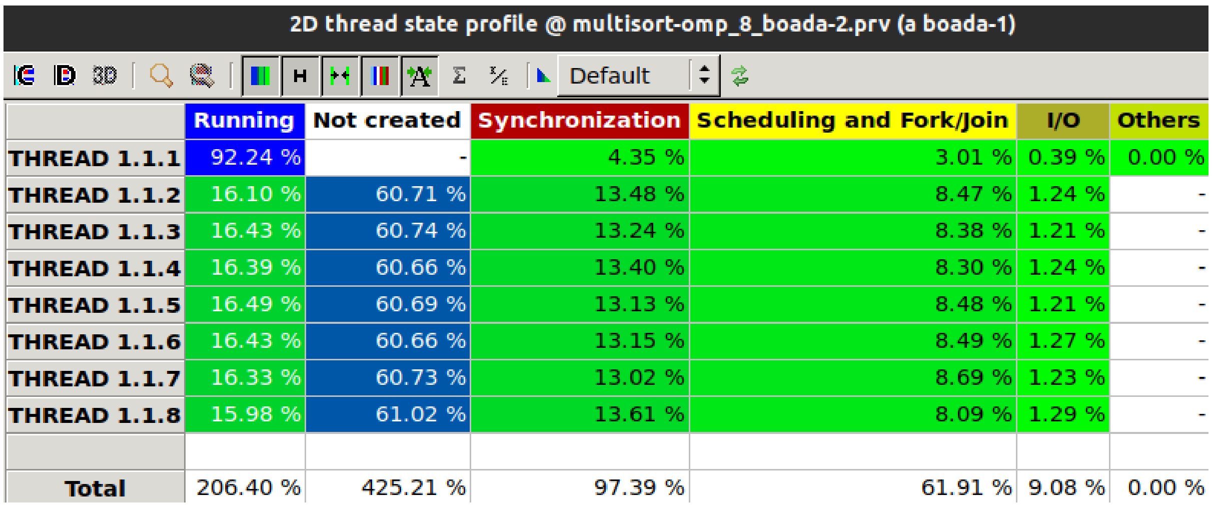


Figure 9: Paraver window for tree strategy


2D thread state profile @ multisort-omp_8_boada-2.prv (a boada-1)

Default

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	92.24 %	-	4.35 %	3.01 %	0.39 %	0.00 %
THREAD 1.1.2	16.10 %	60.71 %	13.48 %	8.47 %	1.24 %	-
THREAD 1.1.3	16.43 %	60.74 %	13.24 %	8.38 %	1.21 %	-
THREAD 1.1.4	16.39 %	60.66 %	13.40 %	8.30 %	1.24 %	-
THREAD 1.1.5	16.49 %	60.69 %	13.13 %	8.48 %	1.21 %	-
THREAD 1.1.6	16.43 %	60.66 %	13.15 %	8.49 %	1.27 %	-
THREAD 1.1.7	16.33 %	60.73 %	13.02 %	8.69 %	1.23 %	-
THREAD 1.1.8	15.98 %	61.02 %	13.61 %	8.09 %	1.29 %	-
Total	206.40 %	425.21 %	97.39 %	61.91 %	9.08 %	0.00 %

Figure 10: Histogram of tree strategy

3. Task granularity control: the cut–off mechanism

Comment the results that you obtained.

Through the *cut-off* mechanism, we're not only capable of checking the proper functioning of our implementation, but also (as we will imminently comment) of improving the efficiency of the strategy itself. For a *CUTOFF* = 0, as it can be seen in figure 11, we detect the creation and execution of 4 main tasks which are a direct result of the first call to the *multisort* function. Moreover, it can also be noted how the remaining threads, in absence of tasks to execute, maintain themselves in a *synchronization* state, along with the fact that the only threads that will ever get to access part of the main workload will be the first 4 threads that pick up those 4 main tasks. It must be noticed that the 3 remaining tasks (executed in threads 4, 6 and 8) correspond to the tasks created by the calls to the *merge* function, which have also been properly parallelized.
In regards to the implementation itself, when we execute the program with *CUTOFF* = 1, just as it can bee seen in figure 12, now all threads get to execute some of the tasks created in the *multisort* function. Moreover, it is also possible to observe the presence of the 16 main tasks, which correspond to the *multisort* calls within the function itself, along with the smaller tasks which, just as we've previously mentioned, correspond to the *merge* calls.
Based on these figures, it is noticeable that more parallelization will be achievable for each new level we traverse, thus allowing us to increase the efficiency of the overall execution of the program as well as its speed-up.

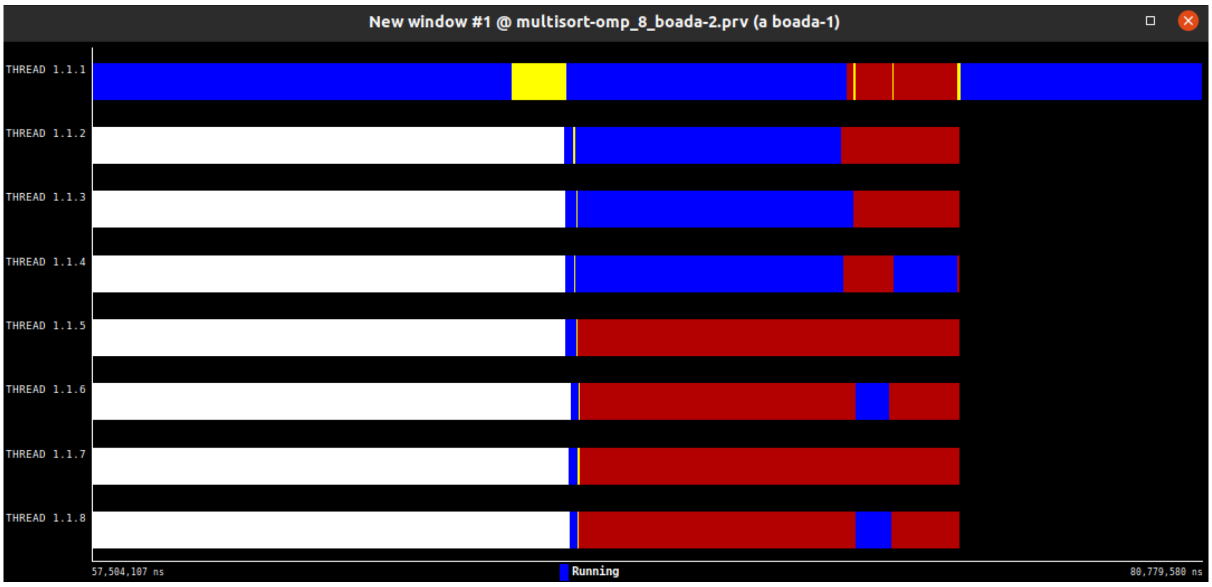


Figure 11: Paraver window of the tree strategy with CUTOFF = 0

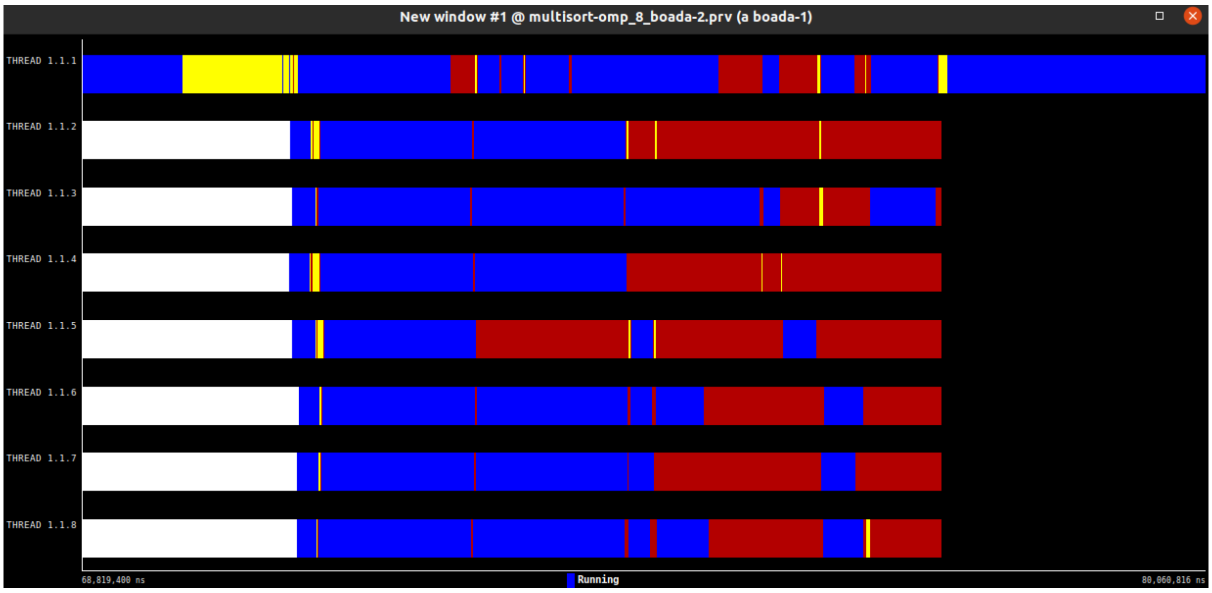


Figure 12: Paraver window of the tree strategy with CUTOFF = 1

According to what's been stated, once the nature of the *cut-off* mechanism has been explained and the proper functioning of our implementation has been adequately evaluated, we've executed the script *submit-cutoff-omp.sh* for 8 and 16 threads:
Figure 13 represents the execution times for the former case. Through this plot, we can observe that, preliminarily, the execution time decreases considerably with each increase of the *CUTOFF* value, and posteriorly this considerable decrease loses its initial inertia and finally stalls for *CUTOFF* = 7. From that moment on, inefficiency manifests and the execution time increases.
Figure 14, on the other hand, represents the execution for the latter case. This plot implies that the augment of threads translates into an increase in the performance of the execution of the program, leading to a reduction in its execution time to almost a halve of the previous case (it must be noted, however, that it's not that much given the present overheads).
Once we've analyzed both plots, it can be noticed that the optimum value for the recursion size, which is the same for *both* executions (and that we expect to hold true for any other one might want to do), oscillates between 6 and 7. Said value will be explained and used in the following section to obtain the strong scalability plot.

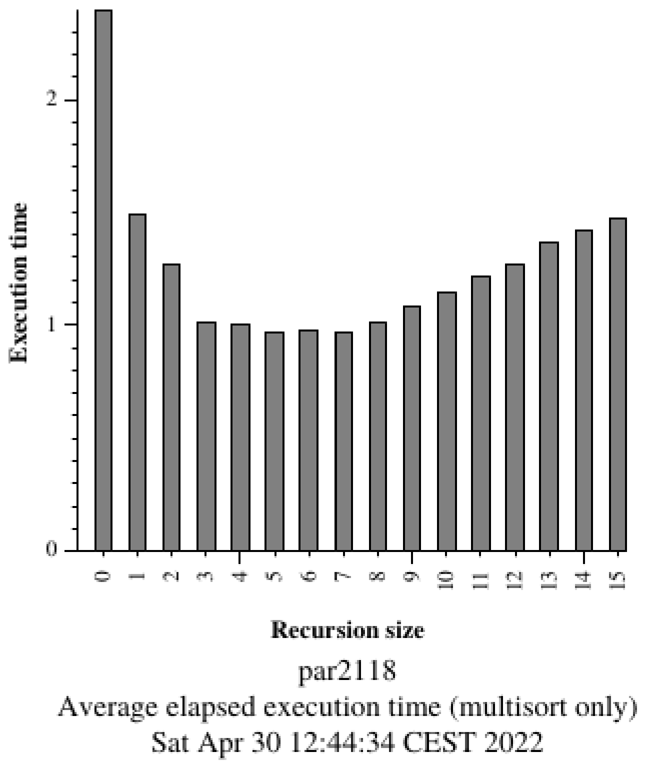


Figure 13: average time for the tree strategy execution with 8 threads

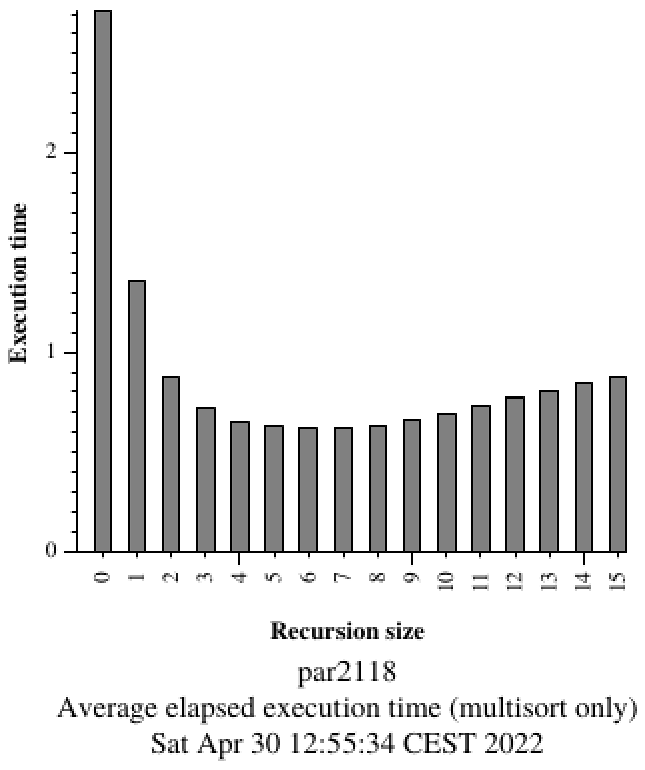


Figure 14: average time for the tree strategy execution with 16 threads

According to what's been stated above, the *strong scalability* of the present implementation has been studied, leading to the following results: for an execution with a *CUTOFF* = 16, it can be noticed that the scalability is relatively linear to the number of threads present in the execution of the program, which implies that the efficiency and speed-up achievable increases with each level of depth of the task tree that is explored. However, as figures 12 and 13 have demonstrated, the optimum level for stopping said exploration corresponds to a *CUTOFF* = 7, which leads us to think if said value translates into an even bigger scalability.
Once this doubt has arised, evaluating said phenomena with the *submit-strong-omp.sh* gives place to figure 16, from which one can observe how the scalability is bigger and the obtained speed-up is more linear in relation to the present number of threads during the execution. With this evidence, we can conclude that, even though the *CUTOFF* generally implies an increment in the efficiency of the execution of the program, not all *CUTOFF* values will make such scalability possible, and for the particular case that we're studying, a *CUTOFF* of 7 is the maximum value that one should use, since it prevents any inefficiency as well as allowing an adequate scaling of the execution itself.

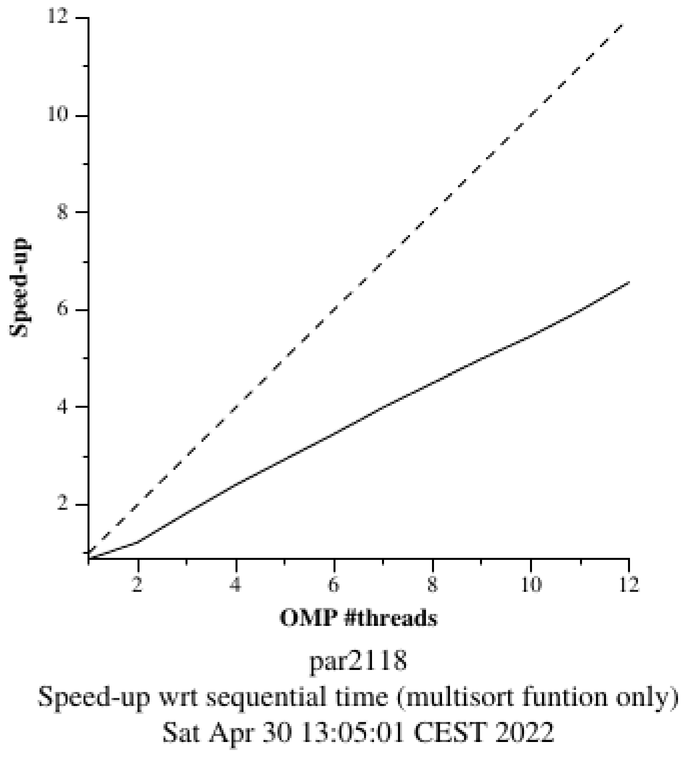


Figure 15: speed-up of the tree strategy execution with CUTOFF = 16

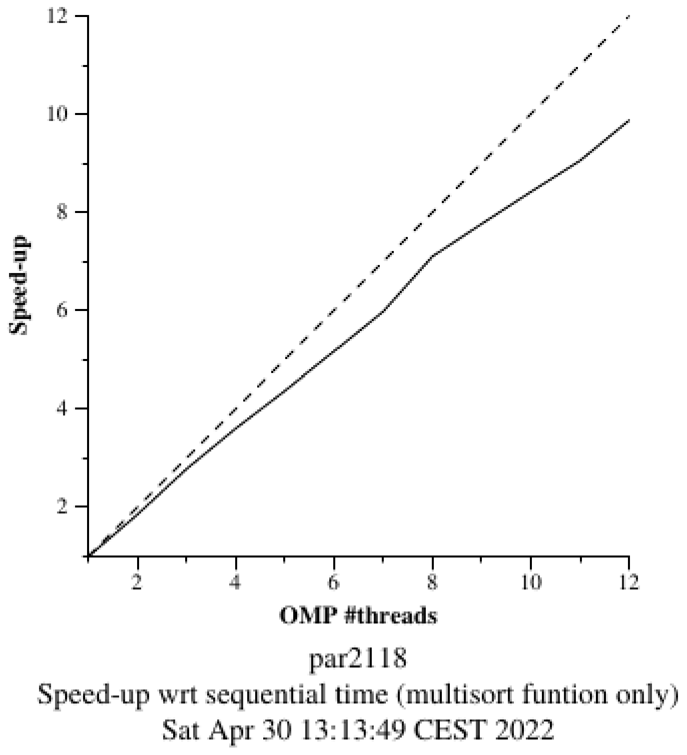


Figure 16: speed-up of the tree strategy execution with CUTOFF = 7

Optional 1

Have you explored the scalability of your tree implementation with cut–off when using up to 24 threads? Why is performance still growing when using more than the 12 physical cores available in boada-1 to 4?

Just as we'd studied in the 1st lab session, each one of the 12 available CPU's, from boada-1 to boada-4, are made of 2 PUs, which implies the existence of a grand total of 24 CPU's, capable of assuming the execution of 24 threads maximum and, thus, allowing an improvement of the performance of the program. If figure 17 represents the tendency of *multisort*'s performance up until 24 threads, figure 18 represents what happens when we execute such a program with more than those 24: since the system is physically impossible to assume more threads, there's a downfall in the obtained speed-up.

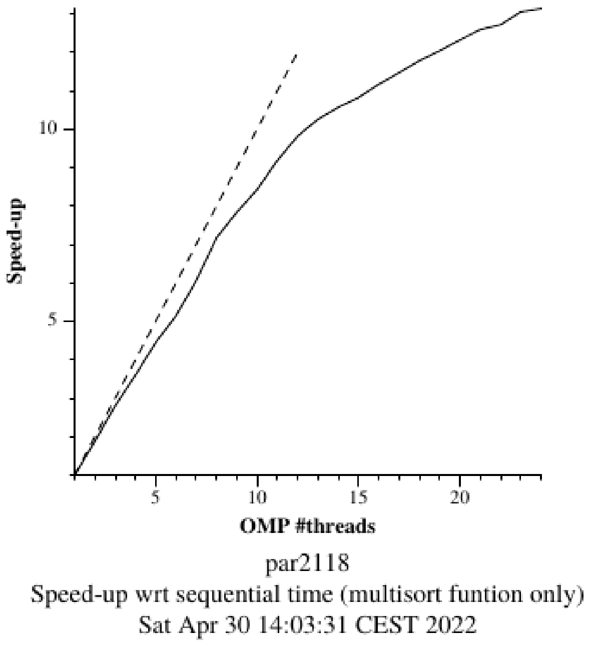


Figure 17: strong scalability of the tree strategy for 24 threads

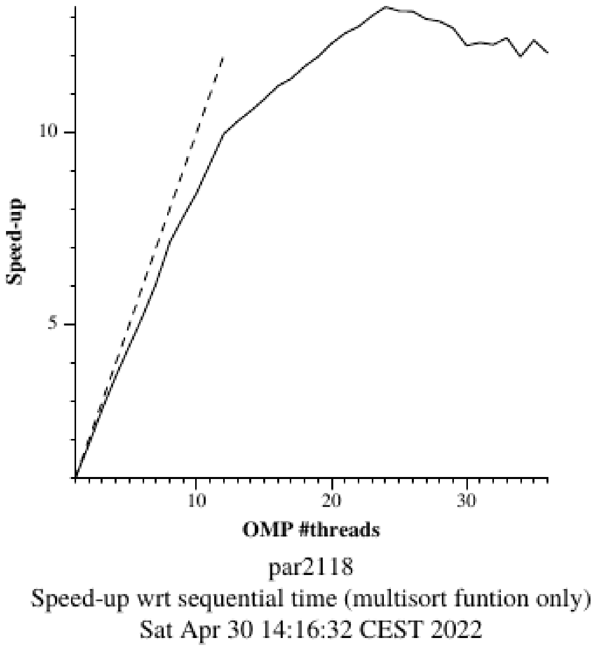


Figure 18: strong scalability of the tree strategy for 36 threads

4. Using OpenMP task dependencies

Analyse its scalability by looking at the two strong scalability plots and compare the results with the ones obtained in the previous chapter. Are they better or worse in terms of performance? In terms of programmability, was this new version simpler to code?

To compare the present implementation with the previous one, we've done the same two tests with the *submit-strong-omp.sh* (just as in the previous section), with *CUTOFF* = 16 for the first case and *CUTOFF* = 7 for the second one. Now, in both cases we detect a very slight improvement in the obtained speed-up which, it must be noted, we'd already expected since the *depend* clause can only introduce a slight change in the course of the execution, for instead of waiting for a determined group of tasks to end (as in the *taskwait-taskgroup* case), it waits for a given value (or set of values).

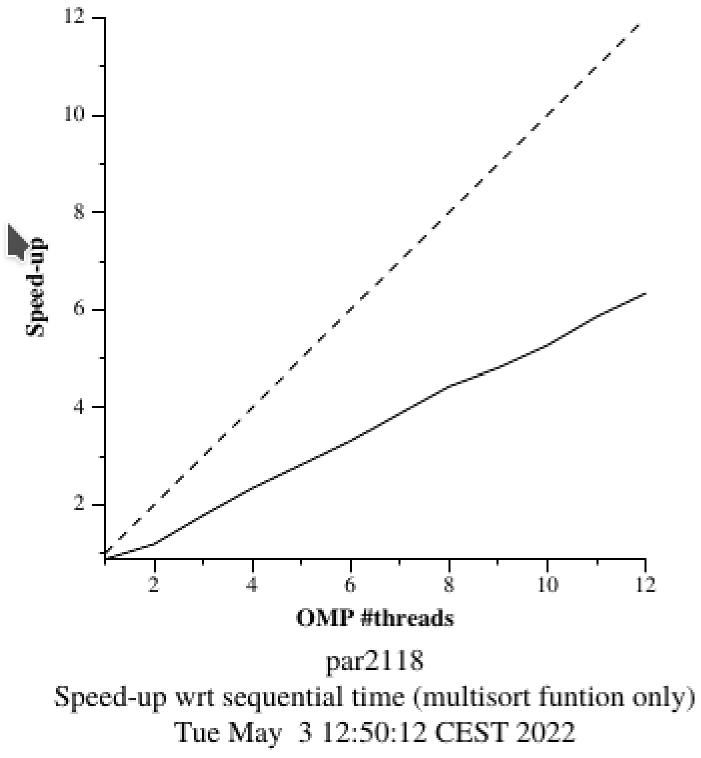


Figure 19: speed-up of the tree strategy with depend clauses and CUTOFF = 16

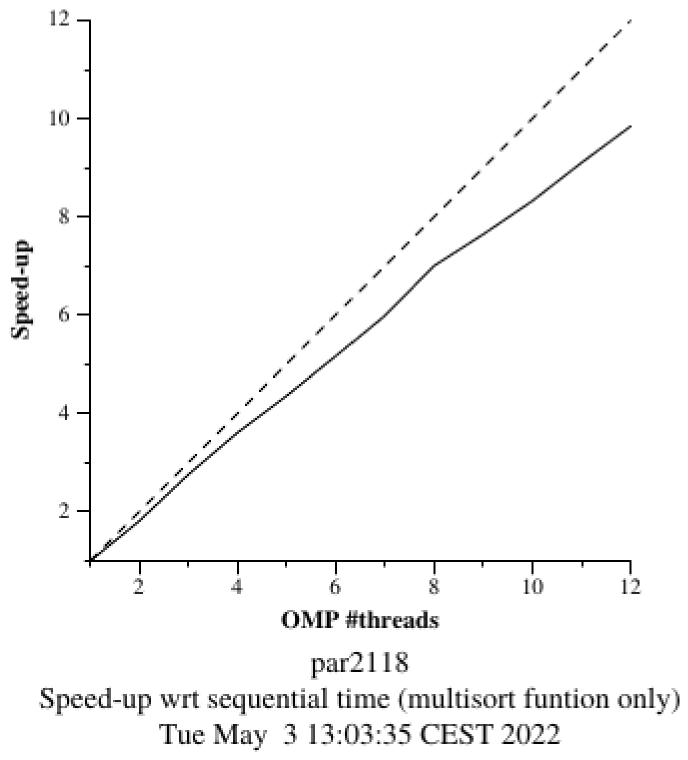


Figure 20: speed-up of the tree strategy with depend clauses and CUTOFF = 7

In regards to the execution of the program for a set of given values for the *CUTOFF*, in contrast to the previous version, there is not enough difference between the plots (figures 13, 14 and 21) to claim that the present implementation is better or more efficient. Thus, with all the graphs presented until now, we're not capable of determining if such a change has an impact on the execution of the program, thus having to recur to the *paraver* data.

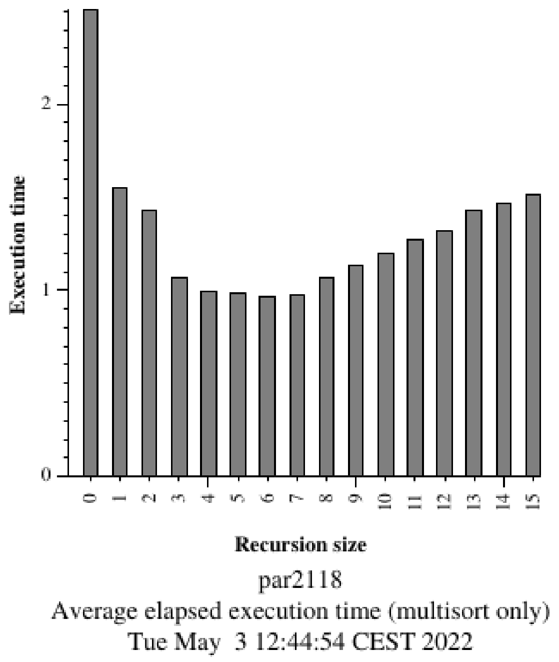


Figure 21: average time for the tree strategy with depend clauses and 8 threads

Now, in regards to the execution of the program in the *Paraver*, the most important thing that one might take into consideration is the difference between the start time (and duration) of each executed task: in the previous section, it can be seen that the execution time of each task is more compact due to the presence of *taskwaits* (which force a kind of synchronization of execution), whereas in the present section, with the use of the *depend* directive, the execution of the program presents a more flexible flow of execution of tasks (since the depends allows for an approach that allows threads to execute certain tasks once *some* of them, the indispensable ones, have already finished).

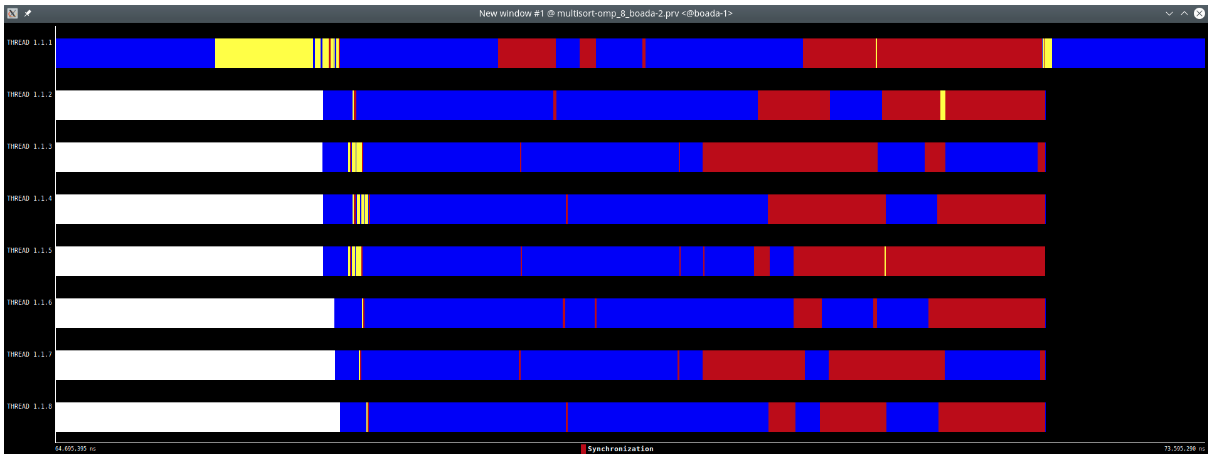


Figure 22: Paraver window of the tree strategy with depend clauses and CUTOFF = 1

Optional 2

If, for all previous implementations, the obtain speed-up of the complete execution of the program didn't scale adequately with the present number of threads (for this matter, figure 5 may be checked since such plot can be extended to all subsequent studied cases), by parallelizing the initializations of the vectors we obtain an even bigger strong scaling: with the addition of the directive *#pragma omp parallel for* (a directive that parallelizes a *for* loop according to the number of threads present during the execution of the program) we can improve the performance of the associated workload to both *initialize* and *clear* functions in a way that evades the stallment of the overall execution time. Such an effect can be seen in figure 23: now that we're using *all* present threads to do a great part of the secondary computations (such as the initializations), we're considerably optimizing the execution time since we're economizing resources in relation to the present workload of the program. It is worth noting that the observable tendency leads to an eventual stallment of the obtained speed-up, even though it is quite clear that such a state is relatively far away (we'd still have room to improve the overall performance of the execution).

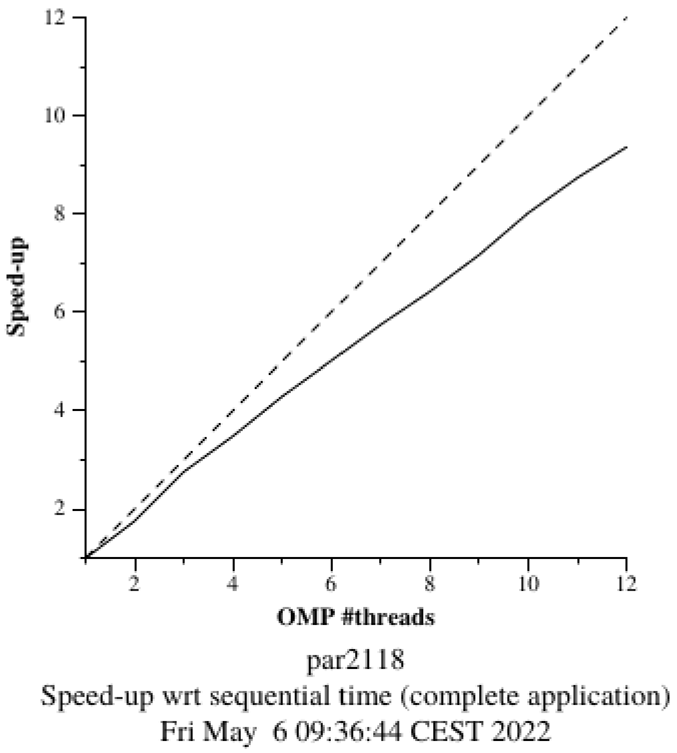


Figure 23: execution time of the whole program with parallel initialization of data and tmp with 8 threads and CUTOFF = 7

Now, in regards to the Paraver execution, the manifestation of some new tasks executing at the beginning of the program (before the *multisort* computation) can be detected in figure 24. Such portions of the figure correspond to the parallelization of both the *initialize* and *clear* functions, which are adequately divided between all available threads, thus leading to a fast initialization of the needed vectors

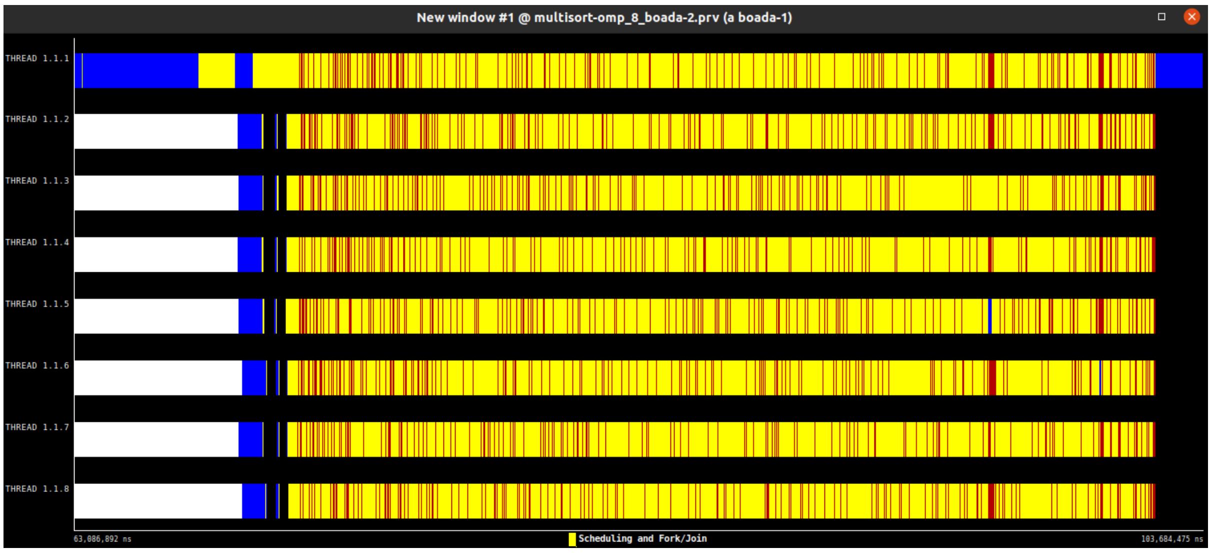


Figure 24: Paraver traces of the execution of the program with initializations parallelized