PAR Laboratory Assignment

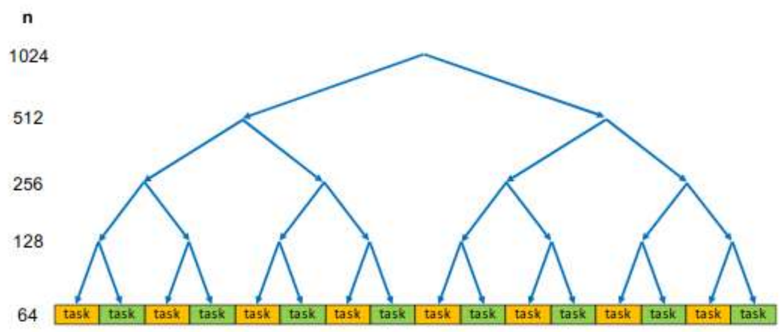
Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Albert Bernal (2205) and Sergi Bertran (2206)

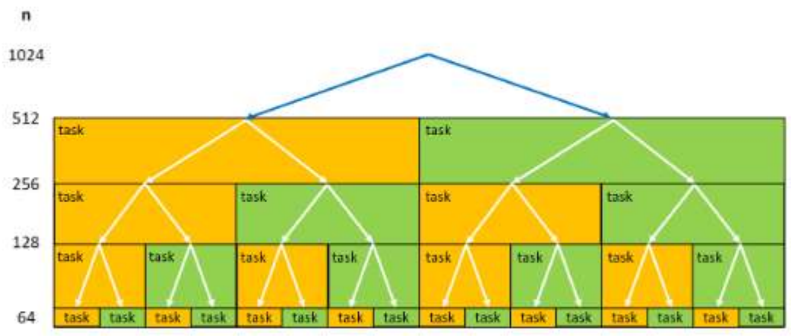
03/11/2020

1.- Introduction

In this laboratory session we will be analysing, implementing and studying the results of parallelization within the divide and conquer strategy used to solve big problems by splitting them into smaller ones that can be solved in parallel.
Our case study will be a classic algorithmic problem: sorting. There are multiple effective strategies that can be implemented while sorting a vector, such as quicksort, bubble sort, etc; and also other quite unconventional algorithms like bogosort. A great option is called mergesort, a divide and conquer algorithm that was invented by John von Neumann in 1945. It works by dividing the vector into small chunks, then sorting them and finally merging them together to obtain a perfectly sorted vector.
We will be approaching the problem with two strategies: leaf and tree. The first one divides the problem recursively into tasks that represent the leaves of the binary tree generated by the recursion. This is a great representation of how the distribution looks, taken from the assignment:



In the tree strategy the task creation is done differently, since we generate a task each time the recursive function is called, therefore where the division of the problem happens. A graphical representation is this one, also taken from the assignment:



Before starting the session both seem great strategies to approach our problem in hand, so our task now is to find out which one is better in which situations by taking a deeper look into the topic.

2.- Task decomposition analysis for Mergesort

2.1 "Divide and conquer"

As always, we copied the current session's tar and extracted it. The first thing we are asked to do is take a look at the **multisort.c** file. Its main function first acts consequently to the parameters that were added when we run the command to execute the program, such as -n (size of the list), -s (size of the vectors that the main one is divided into, and -m (number of merge phases). Then, it initializes the vector, calls the multisort function and checks that the result is correct.

Multisort is a recursive function that first divides the main vector into N/MIN_SORT_SIZE chunks, applies the quicksort algorithm to each one and then merges them into one main sorted vector, obviously the same size as the initial one, N.

We then compiled the code and ran it with the default values. The output (summarized) was the following:

```
N = 32768, MIN_SORT_SIZE = 1024, MIN_MERGE_SIZE = 1024
Initialization time in seconds: 0.872654
Multisort execution time: 6.917492
Check sorted data execution time: 0.021103
Multisort program finished
```

These time values will be used later on to compare the scalability with future parallel versions of the program.

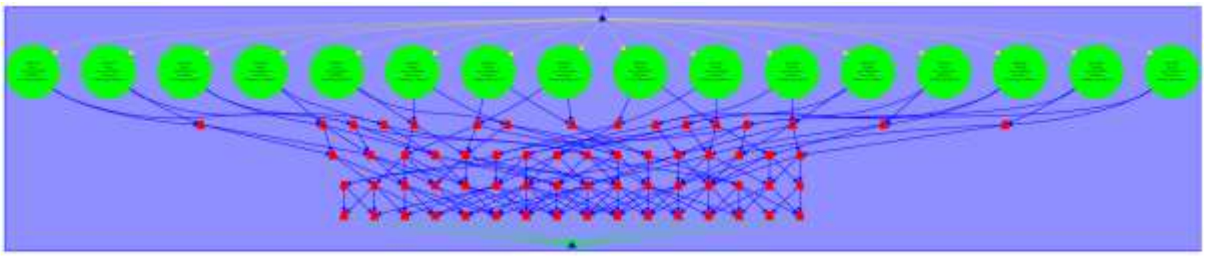
2.2 Task decomposition analysis with Tareador

In this segment we will be taking a deeper look at the code and starting to think about parallelization strategies thanks to Tareador.

The first parallelization method we are going to use is the leaf strategy (see file **multisort-tareador-leaf.c**), in which the problem is divided into small tasks and then every small problem, every leaf is executed in parallel or in parallel groups of leaves. To do it, we inserted "tareador_start_task" in the following positions:

- Twice, before the basicmerge call in the merge function and before the basicsort call in the multisort function. These define the leaves.

This was the result:

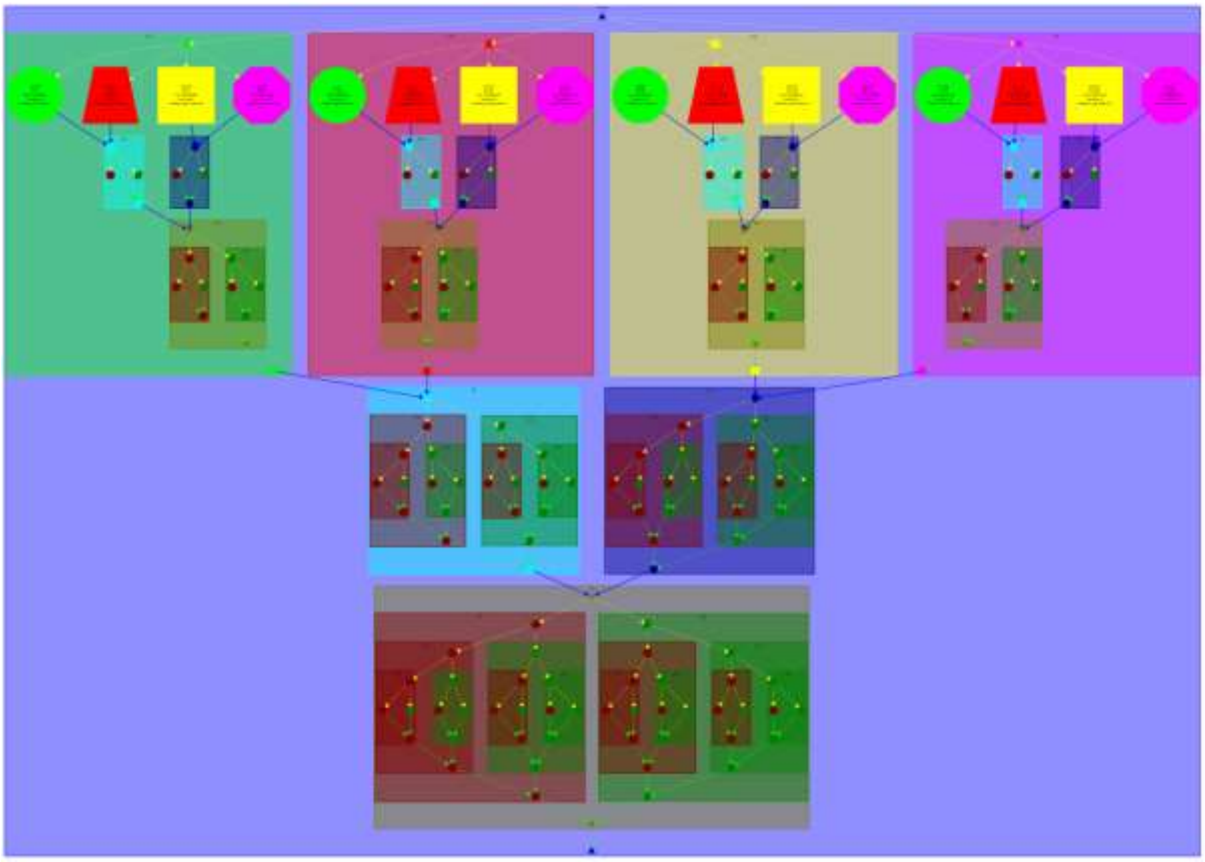


In the graph we can see the 16 basicsort tasks that are generated to compute the result in chunks and the 64 basicmerge tasks that form the vector, divided in 4 stages that are each dependent on the previous stage (1<-2<-3<-4).
We can also observe that the distribution is load balanced and its parallelism is limited by the chunks in which we divide the vector. If we had more threads to run the program, a good idea would be to do another stage of division before the quicksort algorithm is applied, therefore reducing the size of each basicsort task by half. Also reducing the number of merge stages by increasing the number of tasks that are executed in each stage would be beneficial, also needing a big thread count.
We then simulated the execution with a variable number of processors to see the potential scalability of the program:

#threads	time (ns)
1	1.263.265.436.001
2	631.667.499.001
4	315.827.690.001
8	158.459.203.001
16	79.908.621.001
32	79.908.621.001
64	79.908.621.001

As we can see, the speedup stops once we pass the 16 mark, due to the aforementioned fact that the task division is in 16 chunks that can be executed in parallel. Before that, the speedup is really close to 2, hitting the ideal half of the execution time by having twice the threads.

Now let's do the same for the second method, called the tree strategy. In this one, the task corresponds to the invocation of the recursive function itself. The modifications are too many to list here, see **multisort-tareador-tree.c**), but the gist of it is to create a Tareador task before every call to the functions merge and multisort. The Tareador output was the following:



This graph now shows a very apparently different and more visually pleasing image. Upon further inspection, though, we can see that this task distribution is not that far off the previous leaf strategy one. We still have the first layer of 16 tasks that also share the same weight as their leaf counterparts, around 78 million instructions. This is due to the fact that the tasks basically do the same in both decompositions, compute the sorting algorithm. If we take a look at the merging part, at every stage the size of the merge increases due to recursivity, as we can see by the size of the squares that contain each layer, something that didn't happen with the leaf strategy.
The improvements that come to mind are similar to the previous, being decreasing the chunk size the obvious one to reduce the size of the sorting part. In this strategy we should also take into account the bigger number of tasks created, inevitably increasing overhead.

Now let's simulate again the execution with a variable number of processors to see the potential scalability of the program:

#threads	time (ns)
1	1.263.265.436.001
2	631.671.202.001
4	315.837.600.001
8	158.901.550.001
16	79.907.967.001
32	79.907.939.001
64	79.907.939.001

As we can see, the results are very similar, also approaching that ideal speedup of 2 when doubling the threads. Very similar doesn't mean equal, though. The leaf strategy obtained slightly better times, and we think the reason is the task creation overhead.
All in all, both strategies work well and offer great scalability.

3.- Shared-memory parallelisation with OpenMP tasks

Once we have analysed the task decomposition for both strategies, in this section we are going to be implementing them. As we did before, we are going to start with the leaf strategy, but instead of doing everything related with it and then everything related to the tree strategy, in this session we are going to compare them side by side.

In order to parallelise our program using OpenMP we inserted the following directives to the **multisort.c** file (see **multisort-leaf.c** in the deliverable):

- #pragma omp task: Twice, before the basicmerge call in the merge function and before the basicsort call in the multisort function. These define the leaves.
- #pragma omp taskwait: Twice, in the multisort function. These are so that the task definition starts once the recursive divide–and–conquer decomposition stops.

To implement the tree strategy (see **multisort-tree.c)** we modified **multisort.c** by adding:

- #pragma omp task: 9 times, before every call to merge and multisort. Just as we did with the Tareador version.
- #pragma omp taskgroup: Twice, in the multisort function, first grouping the multisort recursive calls, and secondly grouping the merge calls. This is to prevent the second taskgroup from executing before any of the multisort calls, since, the way the algorithm is defined, the merge stage that depends on the result from the multisort stage.

For easier troubleshooting, we added two new entries to the Makefile:

```
multisort-leaf: multisort-leaf.c  kernels.o
	$(OMPC) $(CFLAGS) $(OPT2) $(OPENMP) $+ $(LFLAGS) -o $@

multisort-tree: multisort-tree.c  kernels.o
	$(OMPC) $(CFLAGS) $(OPT2) $(OPENMP) $+ $(LFLAGS) -o $@
```

These are the same as the multisort-omp target, just renamed. Once the codes compiled, we submitted both executions with sbatch.

The output for the leaf strategy was the following (summarized):

```
N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024 CUTOFF=16
OMP_NUM_THREADS=4
Initialization time in seconds: 0.853955
Multisort execution time: 1.983024
Check sorted data execution time: 0.016477
```

These values seem correct, since the time is reduced greatly compared to the sequential execution, which took 6.91 seconds.

And for the tree strategy:

```
N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024 CUTOFF=16
OMP_NUM_THREADS=4
Initialization time in seconds: 0.852604
Multisort execution time: 6.866286
Check sorted data execution time: 0.015309
```

These values are obviously wrong, so we checked the code to see what happened. After too many minutes we found the problem: turns out we had forgotten to add the #pragma omp parallel and #pragma omp single directives before the first multisort call in the main function. The dumbest of errors are the hardest to find.

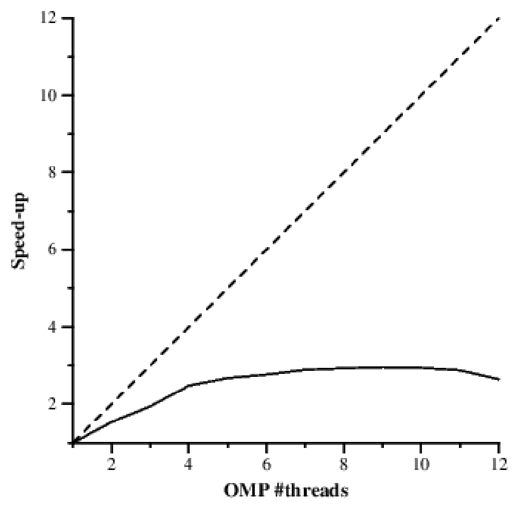
The new values for the tree strategy are the following:

```
N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024 CUTOFF=16
OMP_NUM_THREADS=4
Initialization time in seconds: 0.853156
Multisort execution time: 1.865641
Check sorted data execution time: 0.017906
```

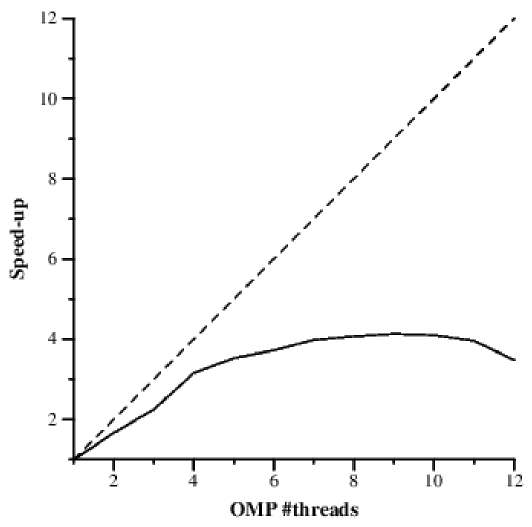
These are what we expected having analysed the simulations in the previous session.

We then submitted the executables with the **submit-strong-omp.sh** script to obtain the speedup plots.

Leaf strategy:

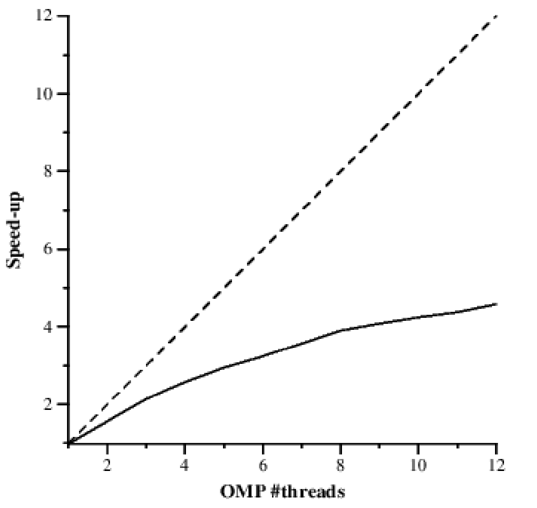


par2205
Speed-up wrt sequential time (complete application)
Tue Dec 1 11:23:56 CET 2020

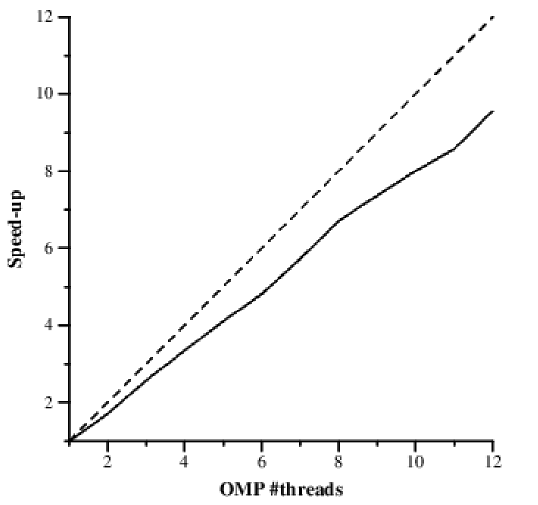


par2205
Speed-up wrt sequential time (multisort funtion only)
Tue Dec 1 11:23:56 CET 2020

Tree strategy:

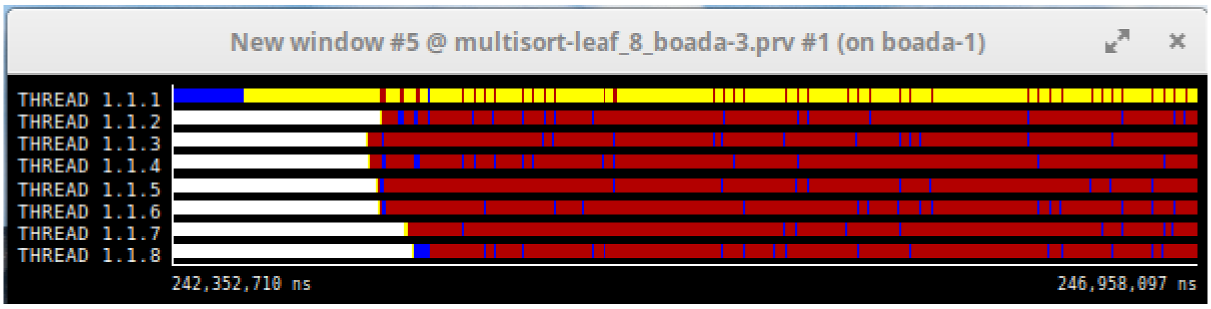


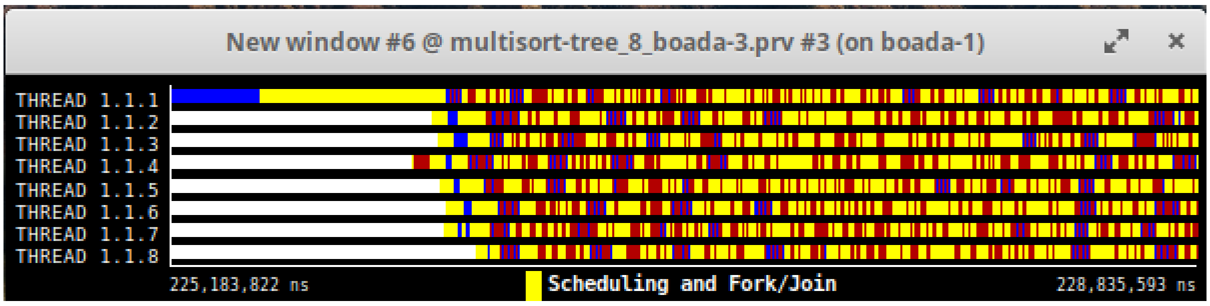
par2205
Speed-up wrt sequential time (complete application)
Tue Dec 1 11:28:32 CET 2020



par2205
Speed-up wrt sequential time (multisort funtion only)
Tue Dec 1 11:28:32 CET 2020

As can be seen in these plots, the tree strategy is clearly superior to its leaf counterpart, specifically when looking at the execution of the multisort function only, which is the part that we parallelized and therefore where the big improvements should be. We made sure that the results were correct checking the output of each execution, and they were. At this point, our only guess is that in the leaf strategy we lose performance because of threads not doing anything while waiting for dependencies. To understand these charts and make a more educated guess about what the problem is, we are going to submit the binaries with the **submit-extrae.sh** script and open the traces generated with Paraver. These are the two execution plots, first leaf, second tree:

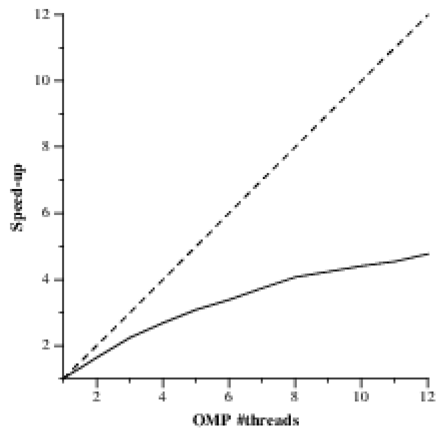




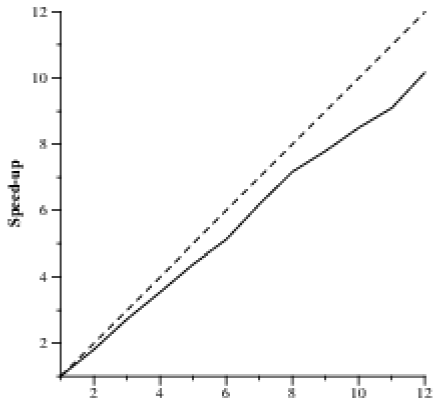
As can be seen in the plots above, even though it seems like the leaf strategy one spends less time in single-threaded execution in the beginning and therefore getting a headstart, this is because of my ability to zoom in, they should be about the same since the sequential code thread 1 executes is the same in both cases. When the parallel section is reached, there's a lot of time spent waiting for other threads, colored in red. On the other hand, the tree strategy spends less time doing things that are not work, such as scheduling and synchronization. This explains the huge difference in the speedup plots that referred to the multisort function only, where the leaf strategy did not escalate well.
Taking into account the previous data, our conclusion is that we should choose the tree strategy over the leaf one, especially when sorting bigger vectors, since we will be taking advantage of the better parallelism in the multisort function.

3.1 Task granularity control: the cut–off mechanism

Finally we will use the cut-off mechanism based on the tree parallelization version code in order to increase task granularity. With this mechanism, we are able to control the maximum recursion level for task generation.



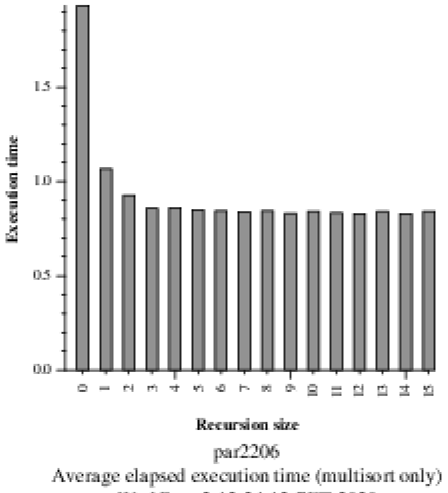
par2206
Speed-up wrt sequential time (complete application)
Wed Dec 2 21:02:55 CET 2020



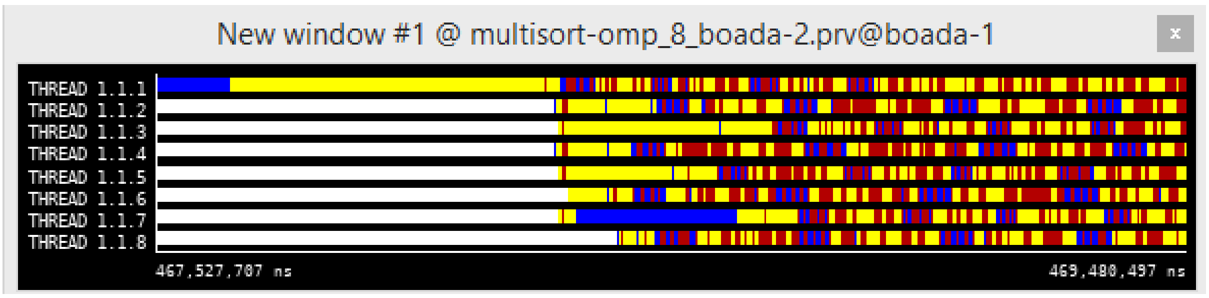
par2206
Speed-up wrt sequential time (multisort funtion only)
Wed Dec 2 21:02:55 CET 2020

In the plot it can be seen the good scalability that this version presents. The speed-up is close to the ideal, so we consider it having considerably good performance.
We can control the cut–off level from the execution command line with the optional flag -c. The default value is 16.
With this option we have tested the execution time depending on the recursion size. We can see in the plot that when the recursion size is higher than 1 the execution time improves considerably. The optimum value is a recursion size of 9.

This version was a little more complex to code. It is necessary to add one more argument in the function and also to replace some #pragma omp task for #pragma omp task final (c>= CUTOFF) in order to the cut-off version works properly. To see every change we implemented, see code on file **multisort-omp-cut-off.c**.



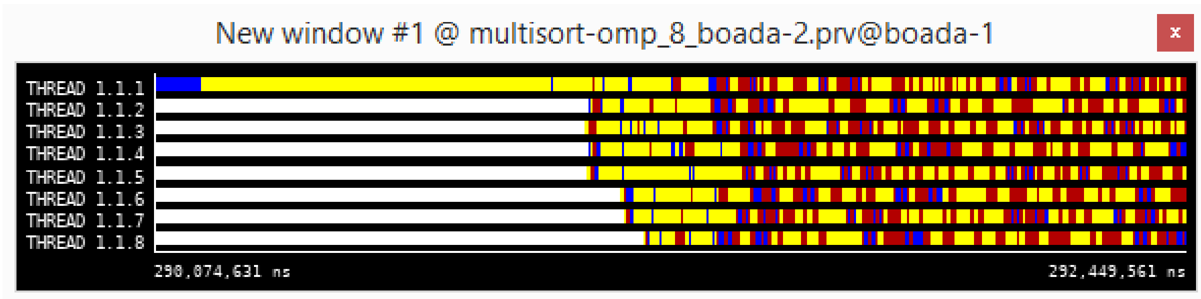
par2206
Average elapsed execution time (multisort only)
Wed Dec 2 13:24:12 CET 2020

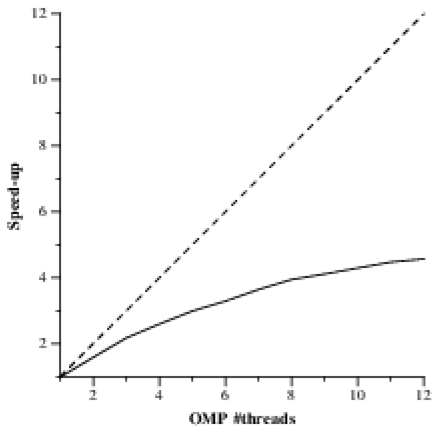


To be honest, we expected a better result. It's still a great performer, but the improvement is not notable.

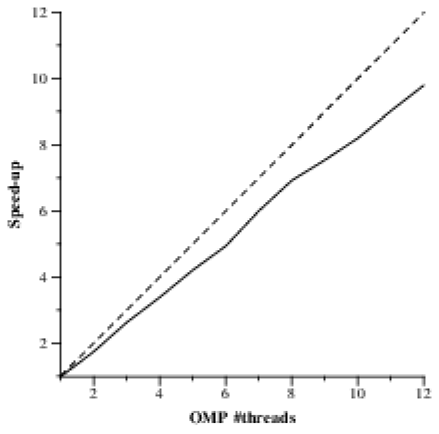
4.- Using OpenMP task dependencies

The aim of this session is to change the tree parallelisation code so as to express dependencies among tasks and avoid some of the taskwait/taskgroup synchronisations. To do so, we have replaced some of the taskwait/taskgroup to depend. The changes are too many to list on this document, see **multisort-omp-dependencies.c**. Not all task synchronisations are needed to be removed, so we have just removed those that are redundant after the specification of dependencies among tasks. We can use the #pragma omp task depend directive in order to set the execution sequence of tasks.
If we compare and analyse the scalability results with the ones obtained in the previous chapter, we see that with 8 processors the scalability is quite similar. This version has more overheads and because of that its performance decreases.





par2206
Speed-up wrt sequential time (complete application)
Wed Dec 2 20:49:13 CET 2020



par2206
Speed-up wrt sequential time (multisort funtion only)
Wed Dec 2 20:49:13 CET 2020

As can be seen in these plots, this is a quite efficient version of the parallel multisort program and it could be the one we would choose to implement, due to its great scalability in the recursive function itself.