PARALLELISM

Laboratory 4

Divide and Conquer parallelism with OpenMP: Sorting

Daniel Muñoz Arroyo, par3110
8/12/2021
3rd, Spring semester

Index

1. Introduction . . . . . page 1

2. Parallelisation strategies
    2.1 Tareador Tree Strategy . . . . page 1
    2.2 Tareador Leaf Strategy . . . . page 4
    2.3 Strategy comparative analysis . . . page 5

3. Performance evaluation
    3.1 Tree parallelization without cut-off . . . page 6
    3.2 Leaf parallelization without cut-off . . . page 8
    3.3 Strategy comparative performance . . . page 10
    3.4 Tree strategy with Cut-off . . . . page 10
    3.5 Tree strategy with Cut-off and depend clauses . page 12

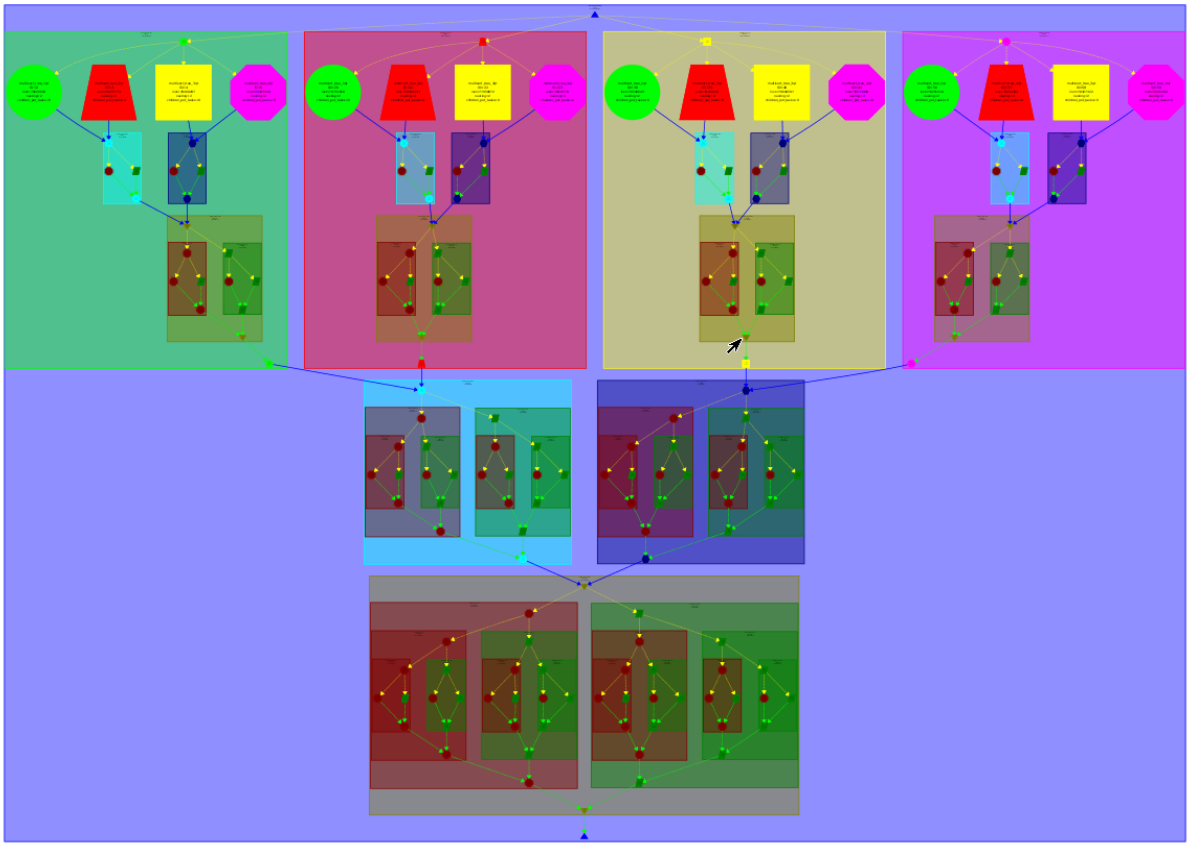
4. Conclusions . . . . . page 15

1. Introduction

In this laboratory, I have to parallelize the multisort code in OpenMP. To do that, First of all we need to know which dependencies are to respect it. Then we can parallelize the code. To do that, I will use some different strategies and types of parallelization. The strategies that I will use are Leaf strategy and Tree strategy. And the type of parallelization is with barriers for both strategies and with a cut-off by levels of recursion and with depends for Tree strategy.

2. Parallelisation strategies

2.1 Tareador Tree Strategy



TDG of the multisort-tareador.c with tree strategy

Multisort function sorts the vector that is passed as parameter.
Merge function merges 2 sortert vectors to merge them in a bigger sorted vector.

The first level of the big tasks are the calls to multisort of each 1/8 of the initial vector. The next level of the big tasks are the merge of every haf (the first tasks, the merge is between the first and second quarter and the second tasks, the merge is between the third and fourth quarter).

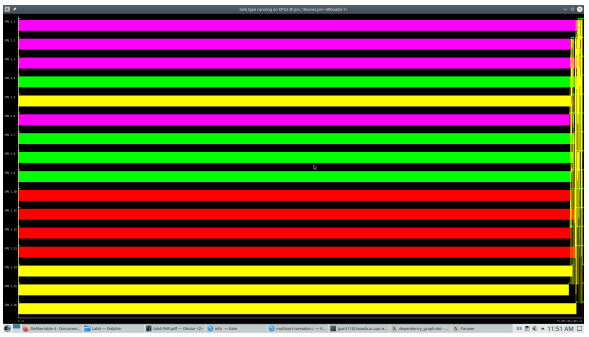
It has dependencies between the first and second quarter in multisort to the first call to merge them, and another between the third and fourth quarter to merge them because sequentially, the multisort of the quarters are done before merging the resultant vector.
The last level is a big task is a call to merge with the 2 vectors that are the result to merge the 1 with the 2 quarter (it is the first half of the initial vector) and the 3 with the 4 quarter (it is the second half of the initial vector).

It has another dependency because sequentially, it computed the merge with the quarters of the vector and then another merge with the halfs resultant vectors of the first merges, so that will need two dependencies to guarantee that.

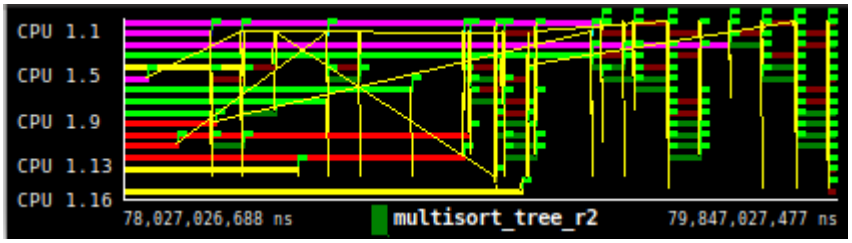
The dependencies in each multisort call have the same structure of dependencies as the graph because it will sort each quarter of the initial vector and then the merge of the two first quarters with a dependency and another merge of the two lasts quarters with a dependency.

The dependencies in each call to merge function are because each can't be computed before his antecedent. It is to respect the order like in the sequential execution.

To respect the dependences, we can use taskgroup before the tasks that need to be done before the other tasks.



Timeline execution of the program with tree strategy in paraver with 16 processors

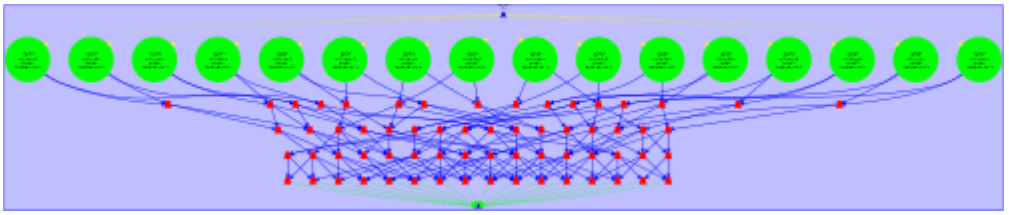


Timeline execution of the final of program with tree strategy in paraver with 16 processors

As we can see in the pictures above the main tasks are the multisort function that is the first level of the TDG. Each processor is executing one task of the multisort function and the flags that are on the right indicate when the multisort task ends and executes the respective recursive merges. The main execution is the multisort task because it takes a lot of time than the execution of merge, as we can see in the second picture above, the multisort_tree_r2 and r1 are the recursive calls of merge.

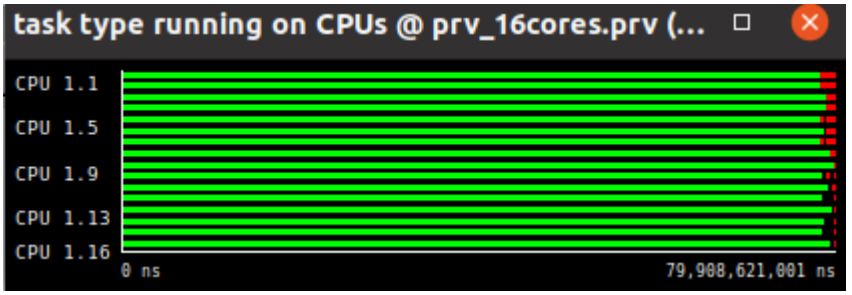
The tasks that the tree strategy generated are the nexts:
The execution was with 32000 elements of the vector and 2048 elements to stop the recursion. Firstly will be 4 task creation and then each tasks will create 4 tasks more with 2048 elements by vector, so 4 + 4*4 = 20 multisort tasks.
Then, the merges will need to merge 3 times the result of the multisort function. One to sort a half, 1 more to sort the other half and the last sort to sort all the vector. It will do in every quarter of the multisort function 3 times more by multisor function = 3 * (16 + 1) = 51 merge in multisort tasks.
in every merge in multisort task, will need 2 tasks more (multisort_tree_r1 and multisort_tree_r2) 16 *2 = 32. In the merge to sort a half of the original vector will need 1 more level because the size are the double (before was a quarter and after a half), so 16 *2 *2 = 64 and in the last merge of all the vector will need 4 times more tasks than in the merge of a quarter of the vector, so 16*2*2*2 = 128 tasks
The recursive merge to merge a vector will need merges tasks = merge in multisort function + merge in merge function = 51 + 32 + 64 +128 = 275 tasks
The total taks needed are 20 +275 = 295 tasks

2.2 Tareador Leaf Strategy

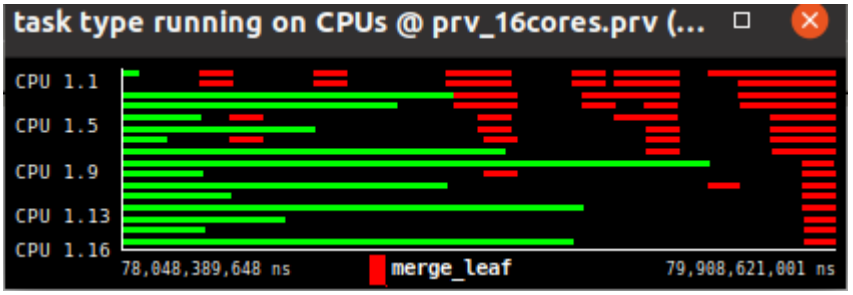


TDG of the multisort-tareador.c with leaf strategy

The first level is the task of the calls to multisort of each ⅛ size of the initial vector. That is because the recursive calls stops when achieve ⅛ size of the initial vector, so they will need to make 32000/2048 = 16 tasks of multisort
In the case of the merge function, the minimum size is 32000/2048 = 16 tasks.
it will make 16 tasks in all the merge levels because the initial vector size is the same. The first level will order every 1/16, the nexts levels will order every 1/8, ¼, ½ and finally, all the vetor. In total there are 16 + 16 *4 = 80 tasks.
The dependencies are the same in both strategies, and in the tree strategy.



Timeline execution of the program with leaf strategy in paraver with 16 processors



Timeline showing the final execution of the program with leaf strategy in paraver with 16 processors

As we can see in the pictures above (the red things are the execution of merge tasks and the green the execution of the multisort tasks), The execution is basically the time to compute the multisort task, because the time to compute the merge tasks is so small.

2.3 Strategy comparative analysis

The differences between leaf and tree strategy are that In the tree strategy, every task is made in every call to recursive functions. But in leaf strategy, every task is a base case.

I calculated the tasks created for both strategies and we can see a summary of them in the net table:

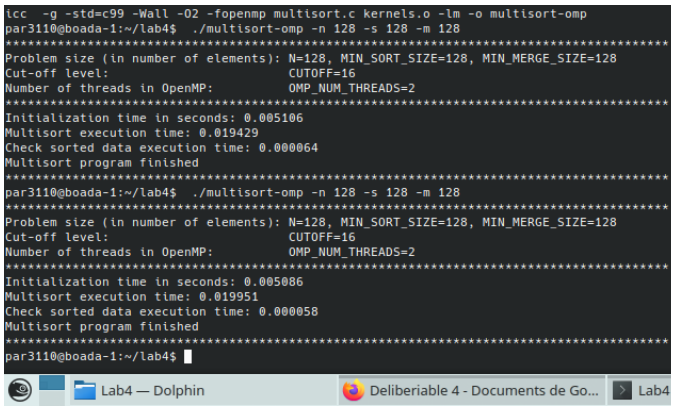
Strategy	multisort tasks	merge tasks	total tasks
Leaf	16	64	80
Tree	20	275	295

As we can see in the table, the tasks in leaf strategy are less than in the tree strategy. It could be because the tree strategy will be creating tasks if the base case is not done, but leaf strategy creates only the tasks for each base case. To respect the dependencies, in the tree strategy I think is better a taskgroup, because they create tasks in tasks and the taskwait for the leaf strategy because the there aren’t tasks in tasks.

3. Parallelisation and performance analysis with tasks

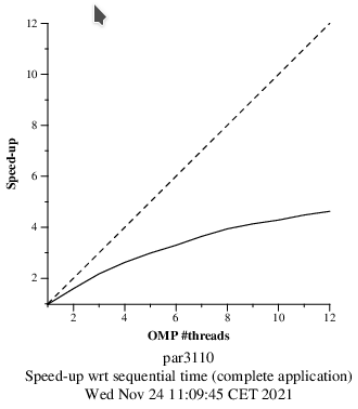
3.1 Tree parallelization without cut-off

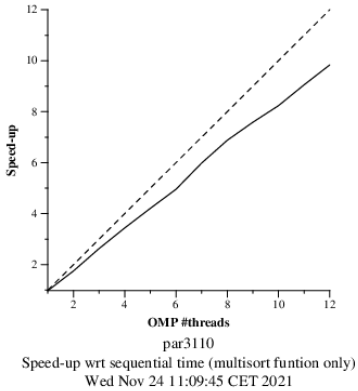
I implemented the tree parallelisation with taskgroups there are tasks created inside other tasks as we can see in the TDG. To respect the dependencies, I put a taskgroup for the 4 recursive calls to mergesort, another one for the 2 first merges, another for the last call to merge and the last one in the merge function, for the 2 recursive calls. We can see it in the file named omp_Tree.c.



To see if it works, I executed it with the same entrance (-n 128 -s 128 -m 128), and the execution time was more or less the same. The next picture shows 2 executions of the tree strategy that I implemented and, we can see, as I just said, there are more or less the same value.

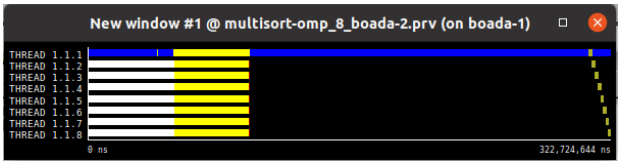
Now, we know it works and with the command “sbatch submit-strong-omp.sh” and “gs file.ps” we can create the next plot and visualize it respectively.



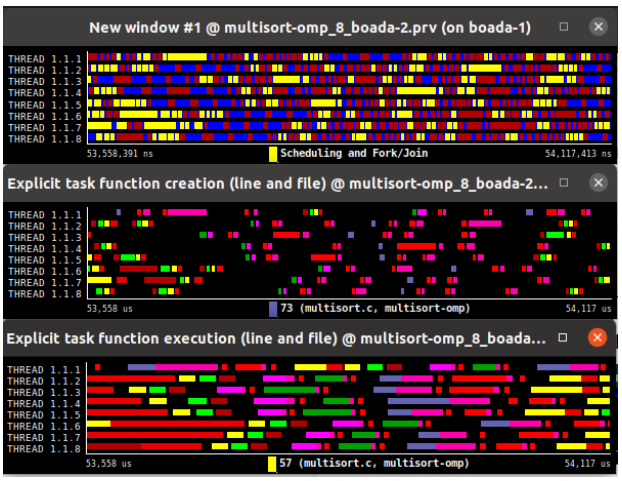


The plot on the left is the execution of the program with the same entry for every execution. In the Y axis we can see the Speed-up and in the X axis the number of threads that were executing the program.
The upper plot is the Speed-up of the whole program and the down plot is Speed-up of multisort function only. As we can see, I reduced the execution time of the multisort function with a good scale, but it is not the whole program, it implies that the speed up of the whole program increases. But it is not the same scalability because the program has sequential execution that is not parallelized and by the amdahl law will not
be the same scale.

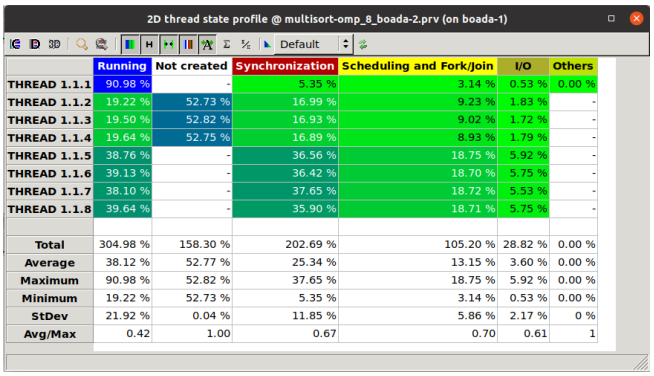
To see if this program can have a better scalability, with “sbatch submit-extrae.sh multisort-omp 8” we can create the traces to see it in paraver and with the command “wxparaver multisort-omp_8_boada-2.prv” we can open it.



The left picture is the timeline of the execution of the program.
The As we can see in the picture on the left have a lot of task creation time (is represented with the yellow color)



To see the synchronization better, I made a zoom on it (upper timeline of the left picture) and with (Hints -> OMP tasking -> explicit tasks function created and executed) I created the explicit tasks and function created (medium timeline on the left picture) and the explicit tasks and function executed (lower timeline on the left picture). The red color for the upper timeline represents the synchronization, the yellow the tasks creation and the blue the task execution.
The red colour for the medium and lower timeline represents the multisort execution or creation and the purple colour the merge execution or creation.
We can not conclude any with the last timelines. To see in detail the percentages of time executon, synchronization time and task creation time, I use a hint of paraver with Hints -> OMP -> Thread state profile (efficiency)



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	90.98 %	-	5.35 %	3.14 %	0.53 %	0.00 %
THREAD 1.1.2	19.22 %	52.73 %	16.99 %	9.23 %	1.83 %	-
THREAD 1.1.3	19.50 %	52.82 %	16.93 %	9.02 %	1.72 %	-
THREAD 1.1.4	19.64 %	52.75 %	16.89 %	8.93 %	1.79 %	-
THREAD 1.1.5	38.76 %	-	36.56 %	18.75 %	5.92 %	-
THREAD 1.1.6	39.13 %	-	36.42 %	18.70 %	5.75 %	-
THREAD 1.1.7	38.10 %	-	37.65 %	18.72 %	5.53 %	-
THREAD 1.1.8	39.64 %	-	35.90 %	18.71 %	5.75 %	-
Total	304.98 %	158.30 %	202.69 %	105.20 %	28.82 %	0.00 %
Average	38.12 %	52.77 %	25.34 %	13.15 %	3.60 %	0.00 %
Maximum	90.98 %	52.82 %	37.65 %	18.75 %	5.92 %	0.00 %
Minimum	19.22 %	52.73 %	5.35 %	3.14 %	0.53 %	0.00 %
StDev	21.92 %	0.04 %	11.85 %	5.86 %	2.17 %	0 %
Avg/Max	0.42	1.00	0.67	0.70	0.61	1

The left picture shows the percentage of execution time, the percentage of synchronization time and the percentage of creation time.
As we can see, the average time of running is 38,53% , 25,34 % of synchronization, 13,35% of task creation time.

The percentage of time running is more or less the same as synchronization and task creation time (38,53% = more or less 38,69%).
The speed-up plot has a good scale because it is running in the parallel region more code than waiting for synchronization or task creation. It can be better performance if we change the barriers for explicit dependencies on the tasks because now there are tasks waiting for synchronization when they could be executing more tasks than if they don’t have the barrier and respecting the dependencies too.

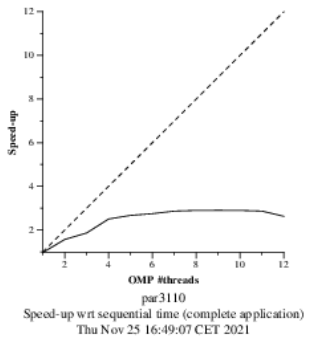
3.2 Leaf parallelization without cut-off

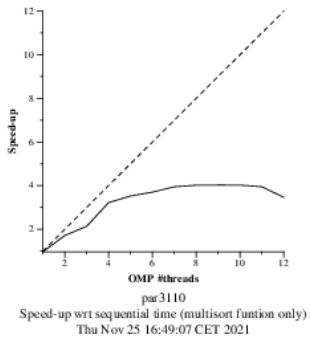
I implemented the leaf parallelisation with taskwait because there aren't tasks created inside other tasks as we can see in the TDG. To respect the dependencies, I put a taswait under the 4 recursive calls to mergesort, another one under the 2 first merges, another under the last call to merge and the last one in the merge function, under the 2 recursive calls. We can see it in the file named omp_Tree.c.

```
Multisort execution time: 0.020831
Check sorted data execution time: 0.000043
Multisort program finished
********************************************************************************
par3110@boada-1:~/lab4$ make multisort-omp
icc  -g -std=c99 -Wall -O2 -fopenmp multisort.c kernels.o -lm -o multisort-omp
par3110@boada-1:~/lab4$  ./multisort-omp -n 128 -s 128 -m 128
********************************************************************************
Problem size (in number of elements): N=128, MIN_SORT_SIZE=128, MIN_MERGE_SIZE=128
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************************
Initialization time in seconds: 0.005358
Multisort execution time: 0.021740
Check sorted data execution time: 0.000060
Multisort program finished
********************************************************************************
par3110@boada-1:~/lab4$  ./multisort-omp -n 128 -s 128 -m 128
********************************************************************************
Problem size (in number of elements): N=128, MIN_SORT_SIZE=128, MIN_MERGE_SIZE=128
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************************
Initialization time in seconds: 0.004434
Multisort execution time: 0.022030
Check sorted data execution time: 0.000058
Multisort program finished
********************************************************************************
par3110@boada-1:~/lab4$  ./multisort-omp -n 128 -s 128 -m 128
********************************************************************************
Problem size (in number of elements): N=128, MIN_SORT_SIZE=128, MIN_MERGE_SIZE=128
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************************
```

To see if it works, I executed it with the same entrance (-n 128 -s 128 -m 128), and the execution time was more or less the same. The next picture shows 3 executions of the leaf strategy that I implemented. We can see, as I just said, there are more or less the same value (0,).

Now, we know it works and with the command "sbatch submit-strong-omp.sh" and "gs file.ps" we can create the next plot and visualize it respectively.

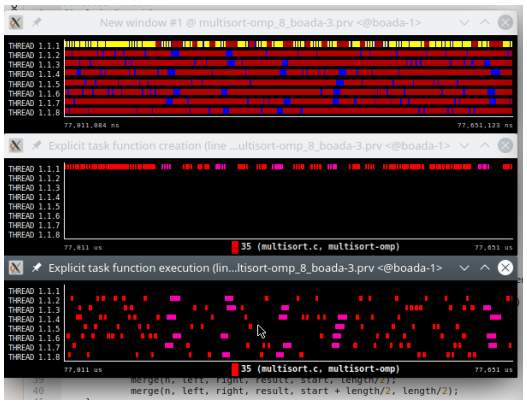




The plot on the left is the execution of the program with the same entry for every execution. In the Y axis we can see the Speed-up and in the X axis the number of threads that were executing the program.
The upper plot is the Speed-up of the whole program and the down plot is Speed-up of multisort function only. As we can see, I reduced the execution time of the multisort function with a good scale, but it is not the whole program, it implies that the speed-up of the whole program increases, but not with the same scalability because the program has sequential execution that is not parallelized and by the amdahl law, will not be the same scale.
When it executes the program with 11 threads, it has less Speed up than before. It could be because it has more synchronization time and task creation than the execution time when executing the parallel code.

To see if this program can have a better scalability, with “sbatch submit-extrae.sh multisort-omp 8” we can create the traces to see it in paraver and with the command “wxparaver multisort-omp_8_boada-3.prv” we can open it.

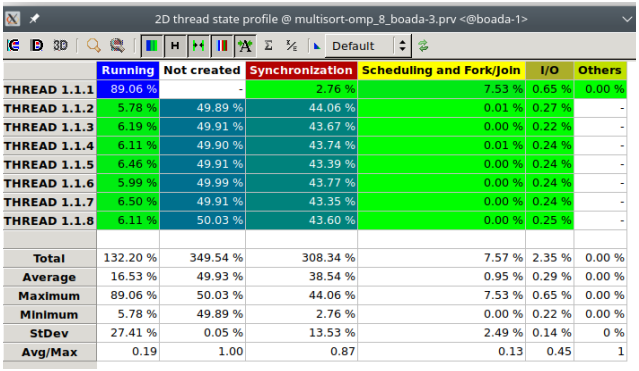
The first timeline is the execution time of the program. As we can see in the picture on the left, it have a lot of synchronization(is represented with the red colour)



To see the synchronization better, I made a zoom on it (upper timeline of the left picture) and with (Hints -> OMP tasking -> explicit tasks function created and executed) I created the explicit tasks and function created (medium timeline on the left picture) and the explicit tasks and function executed (lower timeline on the left picture). The red color for the upper timeline represents the synchronization, the yellow the tasks creation and the blue the task execution.

The red colour for the medium and lower timeline represents the multisort execution or creation and the purple colour the merge execution or creation.

As we can see, the tasks execution spends more time with synchronization than execution. We can see it in the timelines, but to quantify it , with Hints -> OMP -> Thread state profile (efficiency) It generates the left picture that shows the percentage of execution time, the percentage of synchronization time and the percentage of creation time.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	89.06 %	-	2.76 %	7.53 %	0.65 %	0.00 %
THREAD 1.1.2	5.78 %	49.89 %	44.06 %	0.01 %	0.27 %	-
THREAD 1.1.3	6.19 %	49.91 %	43.67 %	0.00 %	0.22 %	-
THREAD 1.1.4	6.11 %	49.90 %	43.74 %	0.01 %	0.24 %	-
THREAD 1.1.5	6.46 %	49.91 %	43.39 %	0.00 %	0.24 %	-
THREAD 1.1.6	5.99 %	49.99 %	43.77 %	0.00 %	0.24 %	-
THREAD 1.1.7	6.50 %	49.91 %	43.35 %	0.00 %	0.24 %	-
THREAD 1.1.8	6.11 %	50.03 %	43.60 %	0.00 %	0.25 %	-
Total	132.20 %	349.54 %	308.34 %	7.57 %	2.35 %	0.00 %
Average	16.53 %	49.93 %	38.54 %	0.95 %	0.29 %	0.00 %
Maximum	89.06 %	50.03 %	44.06 %	7.53 %	0.65 %	0.00 %
Minimum	5.78 %	49.89 %	2.76 %	0.00 %	0.22 %	0.00 %
StDev	27.41 %	0.05 %	13.53 %	2.49 %	0.14 %	0 %
Avg/Max	0.19	1.00	0.87	0.13	0.45	1

As we can see, the average time of running is 16,53% , 38,54 % of synchronization, 0,95% of task creation time and the rest are the sequential execution.

The synchronization time is double the execution time, so it shouldn't be because it wastes so much time waiting for synchronization. For that reason it has a bat scale, as we saw before in the plot. It can be better performance if we change the barriers for explicit dependencies on the tasks because now there are tasks waiting for synchronization when they could be executing more tasks than if they don’t have the barrier and respecting the dependencies too.

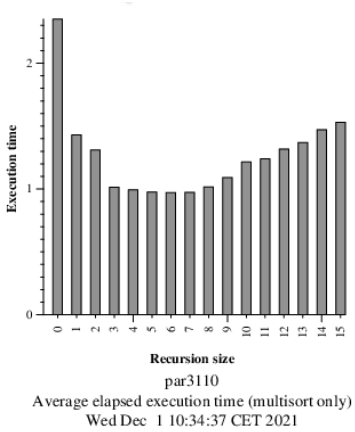
3.3 Strategy comparative performance

The leaf strategy has a bad performance because the synchronization time is double of the execution time. In the tree strategy, there is less synchronization time but more task creation time, but globally, is more or less the same as the execution time. For this reason, tree strategy is better to execute the program than the leaf strategy. The details are explained separately. Each strategy has his respective performance analysis in the 2 sections above.

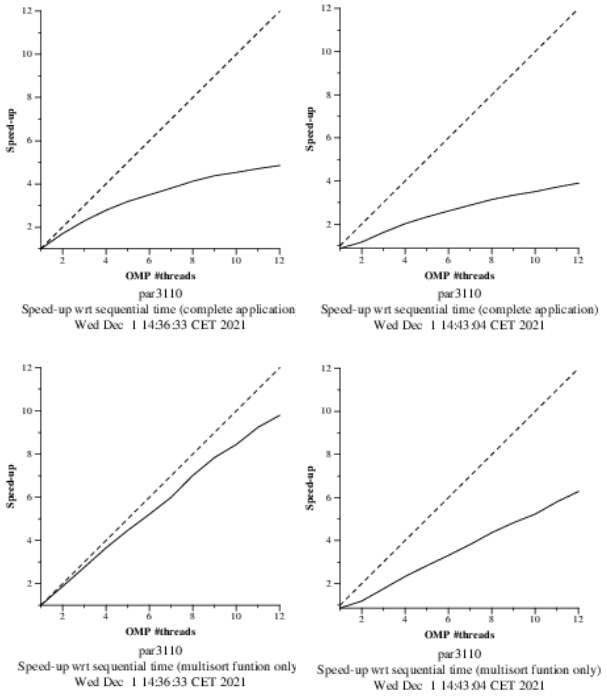
3.4 Tree strategy with Cut-off

The cutt-of is a mechanism to stop generating tasks if the task size is the same or lower than cutt-of value. In mutisort, the cut-off value is readed by the terminal with the -c option and the number of cut-off. In the code, the variable that stores this value is CUTOFF. With that value, we not want to create more tasks if the size of the task is the same or less than the cut-off value, so if it happens, It will call the function recursively, but without creating more tasks. It will finish when the tasks arrive to the min size, like before. The changes that I add in the function multisort and merge if(!omp_in_final()) { and inside the recursive calls with the tasks and in each tasks I add the clause final (n > CUTOFF) to check if the value of the size of the vector is the same or less than the cutoff and after the if, an else with the recursive calls without create more tasks. The omp_in_final() will be false at the beginning and it will be true if the final condition of the tasks are true, so if the size is bigger than cutoff, they will do the if, creating tasks calling the recursive functions, otherwise, it won't create more tasks and it will be calling the recursive function and will finalize when it arrives to the base case like before.
The code is attached with the name omp_tree_cut-off.c

Once implemented, following the reasoning I just explained, we can try to find the optimal number of cut-off and number of processors. To do that, I executed the command "sbatch submit-cutoff-omp.sh numProc" where numProc is the number of processors executed. This scrip, generates a plot whit the execution time and changing the cut-off value from 0 to 15. I did it with 8 processors that are the next:



As we can see, the best execution time for 8 processors is when it executes with a number of cut-off = 6. So 6 is the optimal value. The maximum number that executes the script is 12 (we will see the maximum number on the X axis).

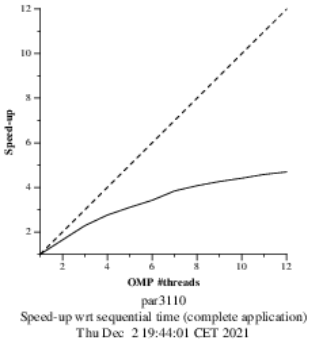


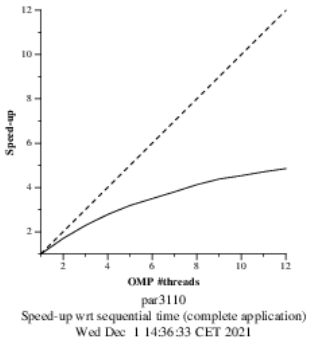
Before I executed it, I changed the values from 1024 to 128 of the multisort_size and merge_size that will determine the MIN_SORT_SIZE and MIN_MERGE_SIZE values respectively of the multisort program.
On the left-left we have the optimal value that I had with the execution (previously, I changed the cut-off value in the script doing cut-off = 6). On the left-right there is the plot with the maximum number of cut-off of the plot (16).

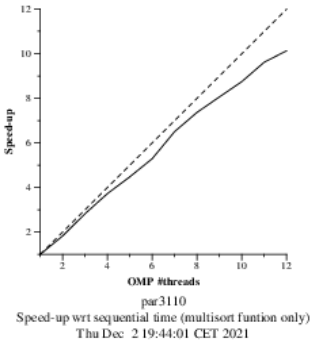
As we can see, the execution with the optimal value of the cut-off has better scalability than the other with the maximum value of the cut-off. It could be because if we decrease the cut-off (from 6 to 1), the granularity will be bigger and it couldn't achieve the best parallelization time. If we increment the cut-off, the synchronization will increase and the execution time will decrease. So the better balance of execution time and synchronization is with the cut-off = 6 and with other numbers of cut-off will have less scalability.

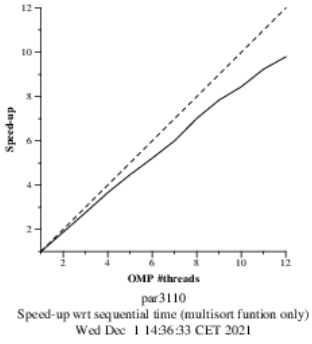
3.5 Tree strategy with Cut-off and depend clauses

To do that, I make a dependence in and out as we can see in the code that are causing dependences (data[0], data[n/4L], data[n/2L], data[3n/4L], tmp[0], tmp[n/2L]) because the data[0] is used to order the first quarter of the vector, data[n/4L] the second one, data[n/2L] the third one and data[3n/4L] the last one. tmp[0] is used to store the first half sorted vector and tmp[n/2L] the last half of the sorted vector. These variables will create depends (data between multisort call and the merges of the quarters and tmp between the merge of the quarters and merge of the halfs).
If I create these depend, I can get out some taskgroups (those are one of multisort in multisort function and one of the merge in merge function). It is possible, because with the depend clause, we respect the multisports call dependencies and the recursive merge doesn't need the taskgroup because it has one implicit in the first merge call and the dependencies are respected with the depend clause.







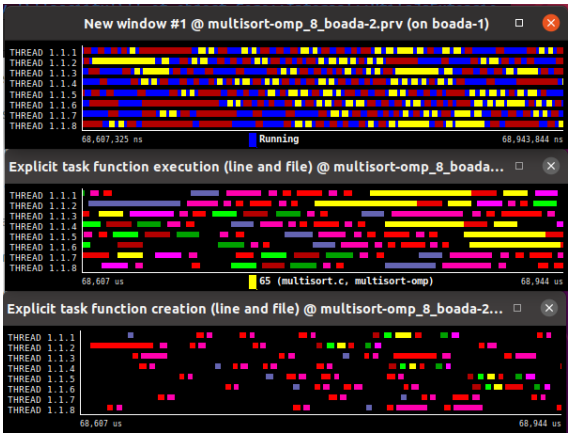


Both plots are with tree stragtegy and cut-off by level of recursion, but the left left plot is the Cut-off with depends and the left right is the plot with only barriers (The plot is the same as in the last section). We can see that they have more or less the same scalability.

If we have a look on paraver, we can see the nexts pictures:

This is the execution with 8 processors. The yellow colour represents the task creation, the red color of the first timetable represents the synchronization time and the blue colour the running tasks.

The next executions are the exection of 8 processors of the code omp_tree_cut-off_depend.c the first timetable is the upper picture but with zoom, so the colour corresponds like I explained before and the 2 last timetables are the result of hints -> OpenMP tasking -> explicit task function created and executed.



The other two timelines are the task function execution and the task function creation respectively and the colours that are not yellow represent the execution or creation task (each different colour represents a different task).

The next picture is the result of do the next path in paraver Hints-> OpenMp -> Thread State Profile (Efficiency)

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

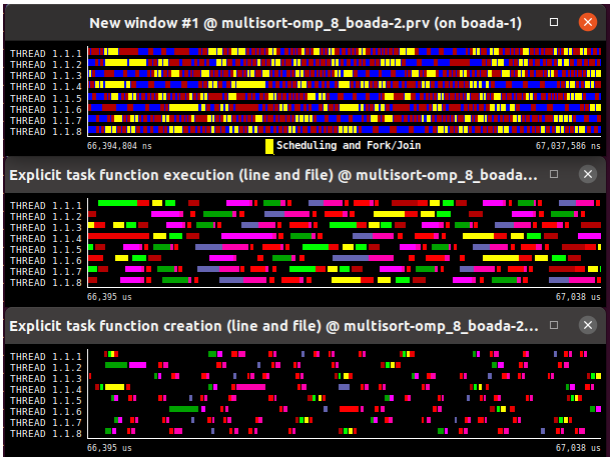
Default

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	95.70 %	-	2.22 %	1.91 %	0.17 %	0.00 %
THREAD 1.1.2	11.38 %	74.05 %	8.43 %	5.53 %	0.60 %	-
THREAD 1.1.3	11.19 %	74.11 %	8.26 %	5.90 %	0.53 %	-
THREAD 1.1.4	11.23 %	74.06 %	8.27 %	5.86 %	0.59 %	-
THREAD 1.1.5	11.48 %	74.03 %	8.47 %	5.49 %	0.53 %	-
THREAD 1.1.6	11.24 %	74.04 %	8.27 %	5.85 %	0.60 %	-
THREAD 1.1.7	10.59 %	74.58 %	8.50 %	5.84 %	0.49 %	-
THREAD 1.1.8	11.39 %	74.15 %	8.07 %	5.86 %	0.53 %	-
Total	174.19 %	519.02 %	60.50 %	42.24 %	4.05 %	0.00 %
Average	21.77 %	74.15 %	7.56 %	5.28 %	0.51 %	0.00 %
Maximum	95.70 %	74.58 %	8.50 %	5.90 %	0.60 %	0.00 %
Minimum	10.59 %	74.03 %	2.22 %	1.91 %	0.17 %	0.00 %
StDev	27.94 %	0.18 %	2.02 %	1.28 %	0.13 %	0 %
Avg/Max	0.23	0.99	0.89	0.89	0.84	1

As we can see, 21,77% is the execution of multisort code, 7,56% of synchronization time and 5,28% is the task creation. The rest is the inactivity but the other threads and I think that is when only a thread executes the sequential part.

If we have a look on the last version (cut-off with barriers with treee strategy), we will see the next:

This is the execution with 8 processors. The yellow colour represents the task creation, the red color of the first timetable represents the synchronization time and the blue colour the running tasks.



The next executions are the exection of 8 processors of the code omp_tree_cut-off_depend.c the first timetable is the upper picture but with zoom, so the colour corresponds like I explained before and the 2 last timetables are the result of hints -> OpenMP tasking -> explicit task function created and executed.
The other two timelines are the task function execution and the task function creation respectively and the colours that are not yellow represent the execution or creation
task (each different colour represents a different task).

The next picture is the result of do the next path in paraver Hints-> OpenMp -> Thread State Profile (Efficiency)

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	95.65 %	-	2.61 %	1.51 %	0.23 %	0.00 %
THREAD 1.1.2	13.24 %	70.30 %	10.08 %	5.44 %	0.94 %	-
THREAD 1.1.3	13.48 %	70.32 %	9.70 %	5.61 %	0.90 %	-
THREAD 1.1.4	13.20 %	70.27 %	10.16 %	5.40 %	0.98 %	-
THREAD 1.1.5	13.53 %	70.28 %	10.21 %	5.03 %	0.95 %	-
THREAD 1.1.6	13.18 %	70.36 %	10.05 %	5.45 %	0.96 %	-
THREAD 1.1.7	13.09 %	70.36 %	10.21 %	5.40 %	0.93 %	-
THREAD 1.1.8	13.28 %	70.41 %	9.97 %	5.38 %	0.96 %	-
Total	188.65 %	492.30 %	72.98 %	39.22 %	6.85 %	0.00 %
Average	23.58 %	70.33 %	9.12 %	4.90 %	0.86 %	0.00 %
Maximum	95.65 %	70.41 %	10.21 %	5.61 %	0.98 %	0.00 %
Minimum	13.09 %	70.27 %	2.61 %	1.51 %	0.23 %	0.00 %
StDev	27.24 %	0.05 %	2.47 %	1.29 %	0.24 %	0 %
Avg/Max	0.25	1.00	0.89	0.87	0.87	1

As we can see, 23,65% is the execution of multisort code, 9,12% of synchronization time and 4,9 % is the task creation. The rest is the inactivity but the other threads and I think that is when only a thread executes the sequential part.

As we saw in the plots, both are more or less the same, but if we compare the paraver captures, with the Hints in paraver, we can see that the synchronization time is less with depends than with barriers. I think it can be because the depends are less restrictive because if a task ends, the next task that was depending on it can be executed. But with the barriers, all the tasks in the task group will wait to finish the task with a higher execution time. The execution time and task creation are more or less the same.

4. Conclusions

The Leaf and Tree strategy needs some depends if we want to parallelize the code. To do that, the simplest way is to use barriers between the levels like are in the code omp_Leaf.c and omp_Tree.c. This strategy will take more synchronization time or will be unbalanced. We can solve it by adding a cut-off by recursion level. For that, It needs to study which level of recursion is the best. In my case it is with 6 because if I put less or more values, the execution time will increase because it takes more synchronization time or it is unbalanced. Use barriers have a problem because all the tasks in the same barrier will wait for the task that needs more time and if a task in the taskwait can be executed before the taskwait ends, with the clause depends, we can solve it in some parts of the code. So with 8 processors and cut-off = 6 the differences are not so big, but if we can decide between them, the best is with depends instead only with barriers to don't waste so much synchronization time as I explained before. The other strategies will take more time, but this strategy will need more thinking (when to put the barriers and which depends we need).