Report

PAR Laboratory 4

Fall 2022-2023

Paula Barrachina Cáceres (*par4109*)
Marc Castro Chavez (*par4111*)

Index

1 Task decomposition analysis for MergeSort

In this section we investigated the potential task decomposition strategies and their implications in terms of parallelism and task interactions of multisort.

1.1 Divide and conquer

First we executed ***sbatch sumbit-seq.sh multisort-seq*** in order to understand how the code multisort.c implements divide and conquer and to obtain the times reported for the sequential execution to use them as a reference time to check the scalability of the parallel versions we will develop in this report.

```
SECUENCIAL
************************************************************************
**********************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
************************************************************************
**********************
Initialization time in seconds: 0.683291
Multisort execution time: 5.194329
Check sorted data execution time: 0.011761
Multisort program finished
************************************************************************
**********************
```

Figure 1. Times of the sequential execution of multisort.c.

1.2 Task decomposition analysis with Tareador

In this section we coded the two strategies, leaf and tree strategy and we studied the potential task decomposition and their implications in terms of parallelism and task interactions.

First we obtained both dependence graphs:
Observing the following dependency graphs we clearly see that the code is divided in two different sections, multisort and mergesort.

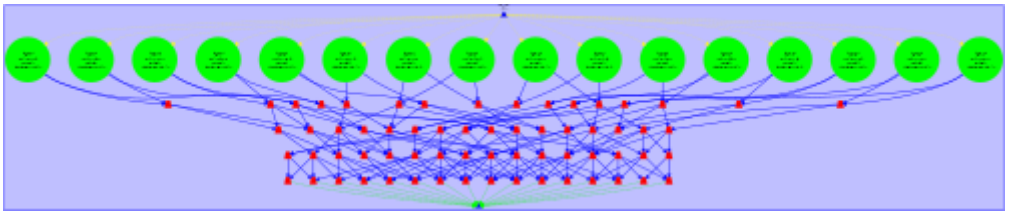

Figure 2. Dependence graph of leaf strategy.

In the leaf graph we can observe that there are 16 basicsort calls, the green ones, and 64 basicmerge calls, the red ones. The basicsort calls do not show any dependencies amongst them, so we can assume that the tasks of basicsort are perfectly parallelized. However, the basicmerge calls are dependent on each other and need some synchronization constraints. Moreover, we see a clear imbalance between the tasks that basicsort and basicmerge.

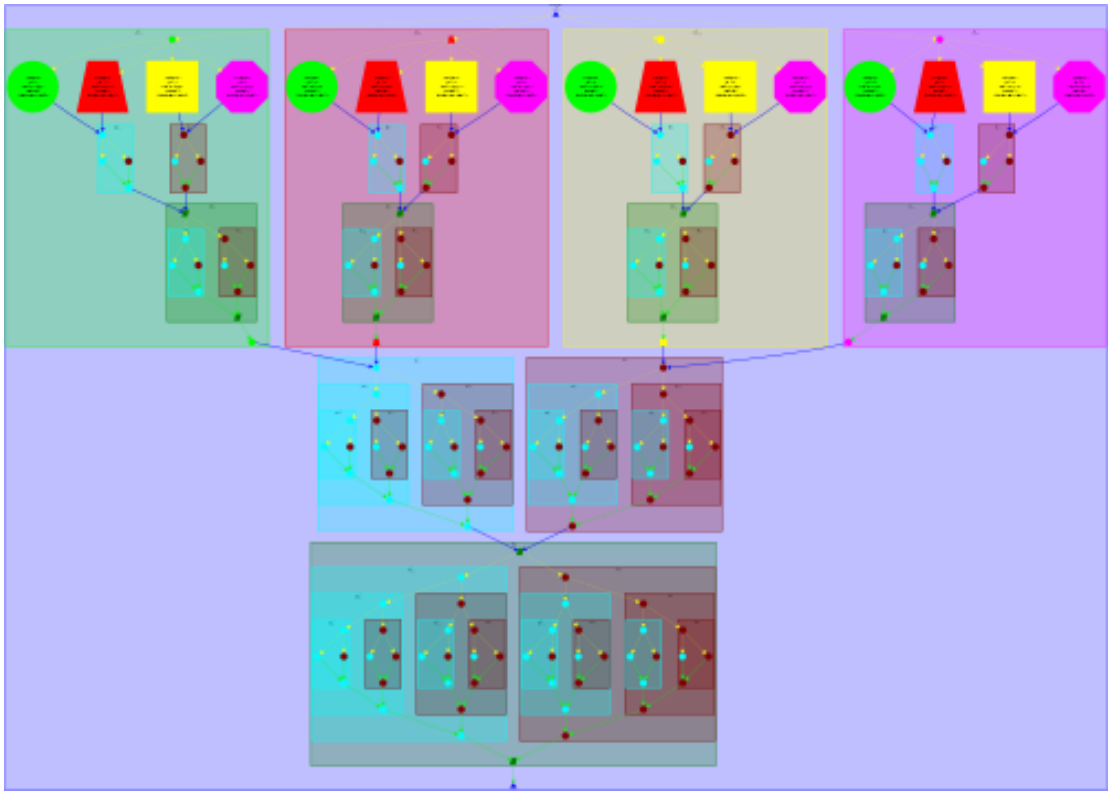

Figure 3. Dependence graph of tree strategy.

On the other hand the tree dependency graph shows how none of the childs are dependent between them, an improvement compared to the leaf strategy. This strategy has more tasks than the previous one because there are more invocations of merge and multisort functions. In terms of granularity we also see a big imbalance between the tasks granularities, specifically between the first multisort tasks.

To understand more about the synchronizations we used Paraver:

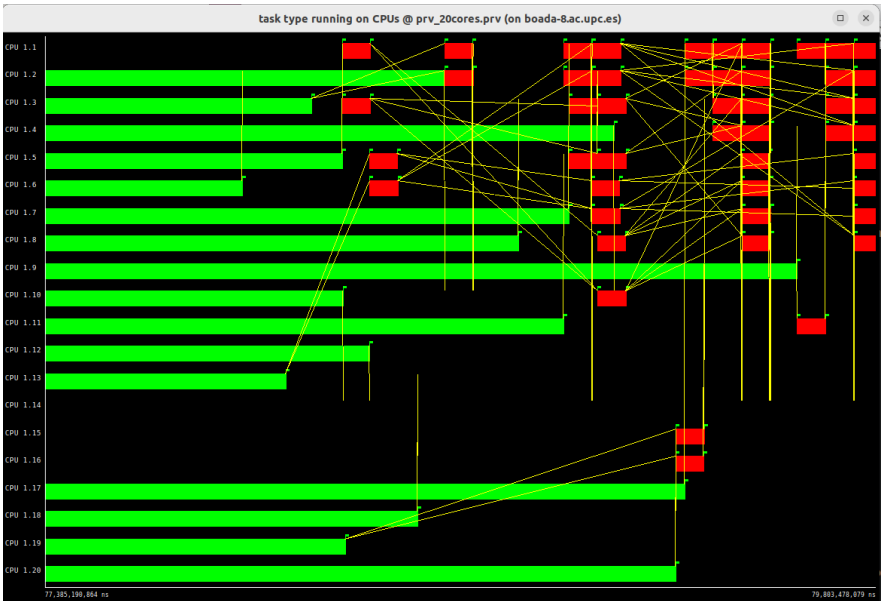

Figure 4. Timeline running of leaf strategy.

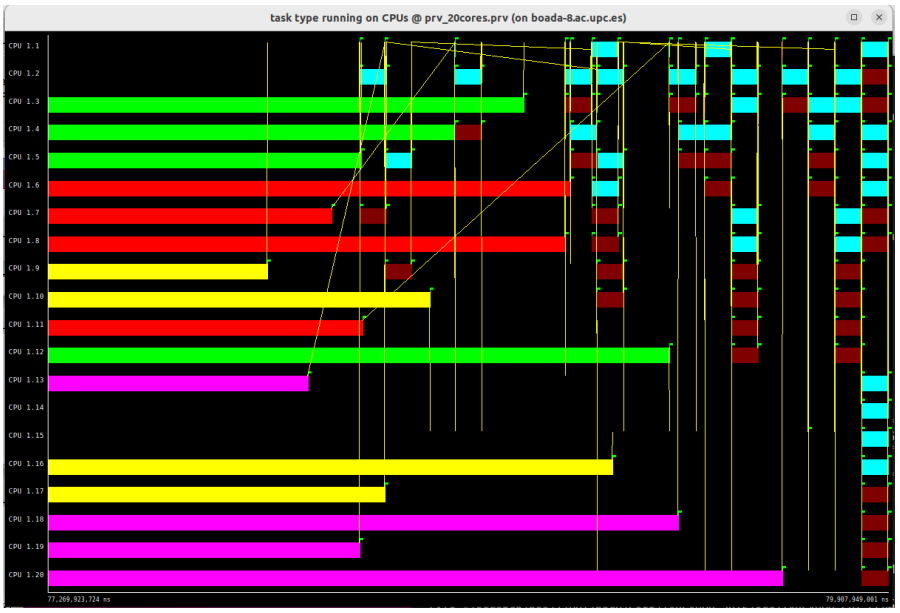

Figure 5. Timeline of tree strategy.

Looking at both the timelines we see that there is a huge difference between the two strategies in the second part of the program. Tree strategy shows a better distribution of the tasks and more parallel than leaf strategy. However, in both cases, merge is the function that creates dependencies and that the first part of the program is similarly executed.

2 Shared-memory parallelisation with OpenMP tasks

In this second session of the assignment we parallelise the original sequential code in multisor-omp.c using OpenMP.

2.1 Leaf strategy in OpenMP

First we inserted the necessary OpenMP tasks for the creation with leaf strategy and then we executed with 2 and 4 threads.

In order to do that we create a task each time we are in the base case of multisort and merge functions, using the ***#pragma omp taskwait*** to synchronize the different threads after making the calls recursive to the multisort function and to execute the two calls to the merge function.

```
************************************************************************************
*******
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************************
*******
Initialization time in seconds: 0.684910
Multisort execution time: 3.115087
Check sorted data execution time: 0.013391
Multisort program finished
************************************************************************************
*******
```

Figure 6. Times of leaf strategy in multisort.c.

In this first version we see an important decrease in the multisort execution time and how the initialization time maintains its value.

Next, after we checked the correctness of our program we analyzed the scalability.

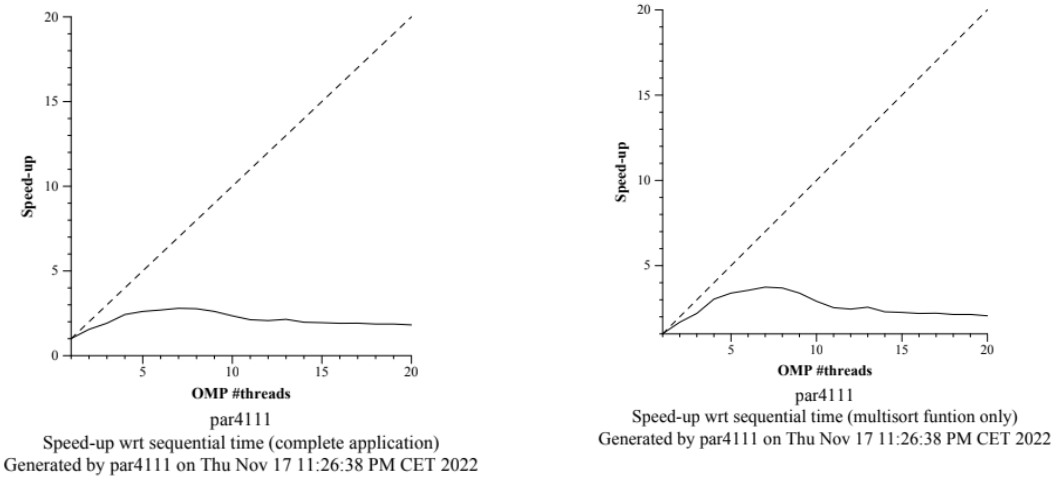

Figure 7. Scalability plots of leaf strategy.

As we can see in the plots the speedup is not as good as we would expect from a parallel code. This is due to the fact that the path to the leaves of the recursion tree is sequential so we spend a considerable amount of time executing sequential code until we reach the leaves.

In order to evaluate the overall performance of leaf strategy we executed 1,2,4,6,8,10,12. 14 and 16 threads with a smaller input. We obtained the following results:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.21	0.24	0.20	0.21	0.20	0.23	0.24	0.28	0.24
Speedup	1.00	0.90	1.07	1.01	1.04	0.91	0.87	0.76	0.89
Efficiency	1.00	0.45	0.27	0.17	0.13	0.09	0.07	0.05	0.06

Table 1: Analysis done on Tue Nov 22 04:53:30 PM CET 2022, par4111

Figure 8. Table 1 of leaf strategy.

Overview of the Efficiency metrics in parallel fraction, ϕ=89.28%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	92.04%	40.93%	24.82%	15.67%	12.12%	8.38%	6.63%	4.92%	5.08%
Parallelization strategy efficiency	92.04%	55.48%	38.43%	25.90%	21.38%	15.32%	12.65%	10.40%	10.29%
Load balancing	100.00%	99.41%	82.20%	64.12%	40.90%	29.21%	22.62%	16.41%	15.72%
In execution efficiency	92.04%	55.81%	46.75%	40.40%	52.27%	52.45%	55.91%	63.35%	65.50%
Scalability for computation tasks	100.00%	73.78%	64.58%	60.48%	56.68%	54.69%	52.42%	47.29%	49.39%
IPC scalability	100.00%	65.23%	57.78%	56.75%	54.73%	53.17%	50.97%	46.41%	49.62%
Instruction scalability	100.00%	112.39%	113.71%	113.37%	111.11%	111.60%	111.47%	110.59%	109.20%
Frequency scalability	100.00%	100.63%	98.29%	94.00%	93.21%	92.17%	92.26%	92.12%	91.15%

Table 2: Analysis done on Tue Nov 22 04:53:30 PM CET 2022, par4111

Figure 9. Table 2 of leaf strategy.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.67	0.74	0.77	0.8	0.76	0.78	0.75	0.8
LB (time executing explicit tasks)	1.0	0.79	0.82	0.79	0.85	0.79	0.8	0.79	0.87
Time per explicit task (average us)	2.74	3.47	3.84	4.05	4.07	3.98	4.0	3.95	4.06
Overhead per explicit task (synch %)	0.89	68.55	170.81	342.56	493.09	794.99	1042.72	1473.15	1393.91
Overhead per explicit task (sched %)	9.4	33.68	39.84	38.14	26.81	32.82	31.7	29.75	21.37
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Tue Nov 22 04:53:30 PM CET 2022, par4111

Figure 10. Table 3 of leaf strategy.

Observing the tables we can appreciate that the overall performance of the program is poor. In the first table observe that the time is nearly constant and does not improve as we add processors. In tables 2 and 3 we see that the global efficiency decreases considerably as whe add processors and the overhead increases.

Furthermore as we said before, the path to the leaves is sequential, so the code does not generate enough tasks and as we can appreciate in table 2 the load balance decreases as we add threads. The code generates 53248 tasks in total.

Figure 11. Instantaneous parallelism of leaf strategy.

As we can see in the figure 11 the parallelism level of the application decreases and increments from a constant value. And the strongest parallelism is found at the beginning and at the end of the program. We can extract from those values that the parallelism is not as efficient as we thought with the leave strategy in this case due to the problem we mentioned before.

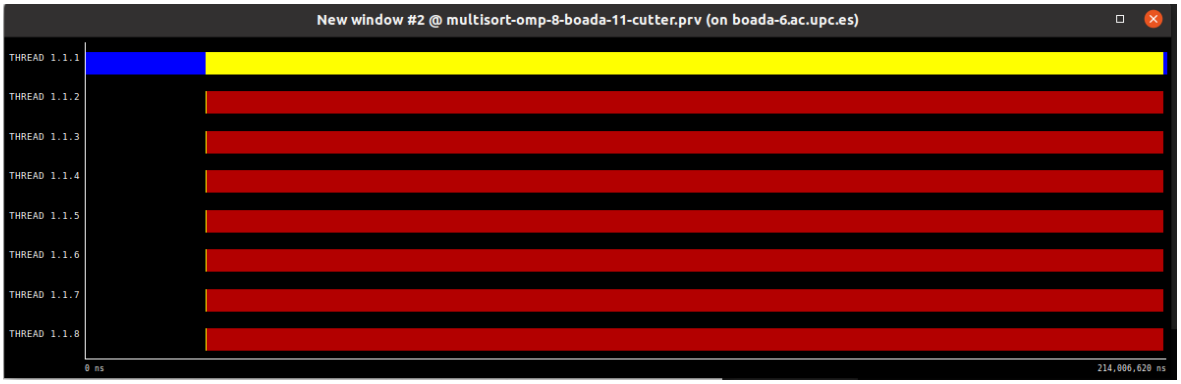

Figure 12. Timeline of leaf strategy.

Here we certify that the parallel execution does not take place until we reach he leaves, that is when the program starts creating tasks.

2.2 Tree strategy in OpenMP

Secondly we inserted the necessary OpenMP tasks for the creation with tree strategy and again, executed with 2 and 4 threads.

In order to correctly implement the the tree strategy we created tasks in the calls of merge and multisort functions using ***#pragma omp taskgroup***.

```
************************************************************************************
*******
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************************
*******
Initialization time in seconds: 0.687491
Multisort execution time: 2.958395
Check sorted data execution time: 0.014926
Multisort program finished
************************************************************************************
*******
```

Figure 13. Times of tree strategy in multisort.c.

In this case in terms of time we can not see a huge difference between leaf strategy and tree, because the multisort execution time varies vaguely.

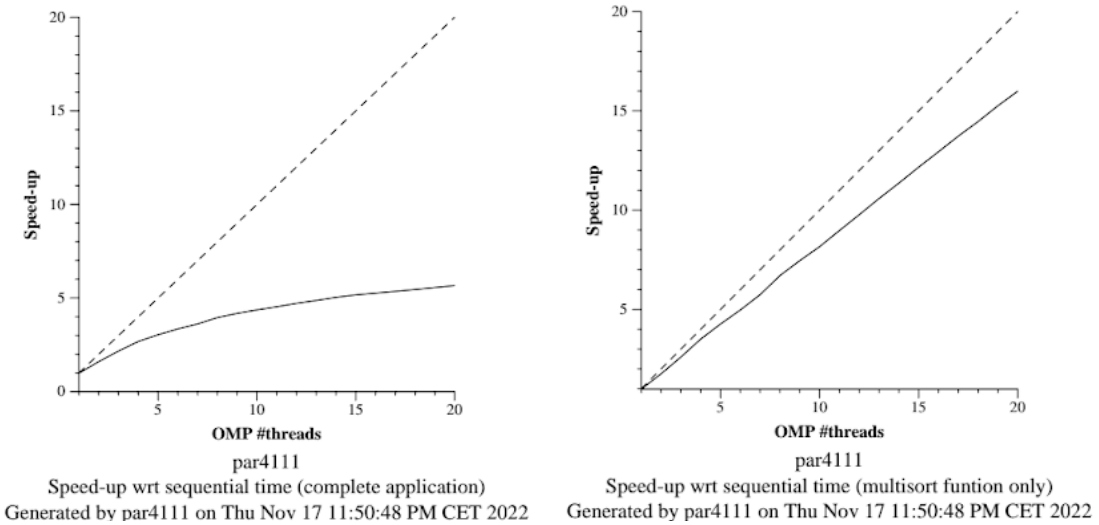

Figure 14. Scalability plots of tree strategy.

Tree strategy shows a slight improvement in Multisort execution time and a huge improvement in terms of speed-up in the parallel region of the program. In plot 14 we see that the speed-up is nearly ideal with 0 to 5 threads. Moreover as we add threads we get a better performance contrary to the leaf strategy case. That better performance taking advantage of the threads will result in more non-dependent tasks.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.26	0.30	0.24	0.23	0.22	0.22	0.22	0.22	0.22
Speedup	1.00	0.88	1.08	1.13	1.19	1.18	1.20	1.20	1.18
Efficiency	1.00	0.44	0.27	0.19	0.15	0.12	0.10	0.09	0.07

Table 1: Analysis done on Tue Nov 22 05:03:42 PM CET 2022, par4111

Figure 15 Table 1 of tree strategy.

Overview of the Efficiency metrics in parallel fraction, ϕ=91.30%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	89.06%	38.93%	24.21%	17.06%	13.57%	10.79%	9.19%	7.82%	6.73%
Parallelization strategy efficiency	89.06%	48.78%	35.48%	27.58%	20.59%	16.96%	14.50%	12.19%	10.47%
Load balancing	100.00%	94.06%	96.66%	96.88%	93.23%	93.00%	90.94%	92.05%	88.81%
In execution efficiency	89.06%	51.86%	36.70%	28.47%	22.09%	18.24%	15.94%	13.24%	11.79%
Scalability for computation tasks	100.00%	79.80%	68.26%	61.84%	65.90%	63.63%	63.40%	64.18%	64.27%
IPC scalability	100.00%	65.65%	56.72%	54.97%	57.83%	57.77%	58.09%	59.05%	58.99%
Instruction scalability	100.00%	121.24%	121.37%	121.44%	121.48%	121.47%	121.38%	121.39%	121.50%
Frequency scalability	100.00%	100.25%	99.16%	92.64%	93.82%	90.68%	89.92%	89.54%	89.67%

Table 2: Analysis done on Tue Nov 22 05:03:42 PM CET 2022, par4111

Figure 16. Table 2 of tree strategy.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.97	0.95	0.97	0.98	0.98	0.97	0.96	0.97
LB (time executing explicit tasks)	1.0	0.98	1.0	0.99	0.99	0.99	0.99	0.99	0.98
Time per explicit task (average us)	1.85	3.87	5.61	7.55	8.95	11.04	12.75	14.78	16.96
Overhead per explicit task (synch %)	1.02	41.02	56.41	64.99	74.93	78.38	81.42	83.79	86.28
Overhead per explicit task (sched %)	13.29	32.16	46.0	56.18	65.87	71.78	75.77	79.8	82.76
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Tue Nov 22 05:03:42 PM CET 2022, par4111

Figure 17. Table 3 of tree strategy.

Tree strategy shows a huge improvement in terms of speed-up. However the time it does not improve in a significant way and it stays constant from 8 processors. In terms of efficiency we observe a slight improvement but not as significant as we expected after seeing the speed-up results. One significant change we can observe is the improvement in load balancing that is far better than the other strategy, which indicates the increase of granularity, also we can appreciate less time added due to overheads.
The code generates 99669 tasks in total, much more than the leaf strategy.

Figure 18. Instantaneous parallelism of tree strategy.

In image 18 we observe that the parallelism of the application is nearly constant since the beginning of the paralyzation and it increases more than decreases and contrary to the leaf strategy it maintains its value since the beginning.

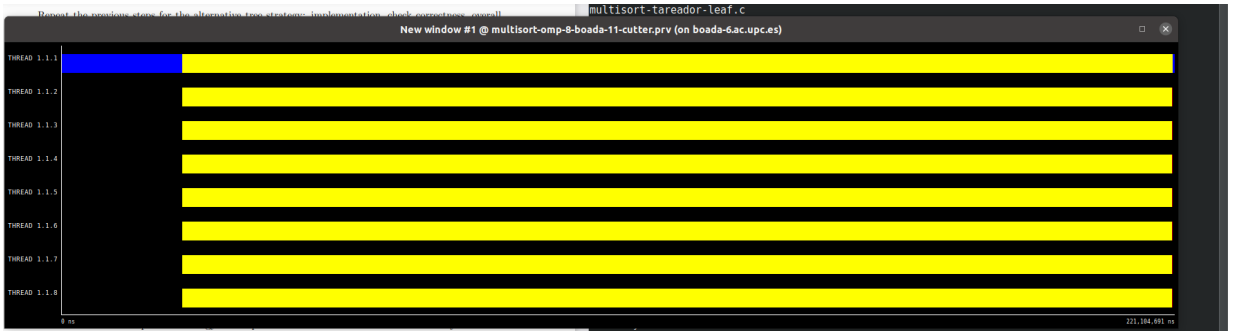

Figure 19. Timeline of tree strategy.

Tree strategy has more tasks, that is because we have one different call for each recursive call in the program. That allows the program to take advantage of more threads at the same time and explote parallelism and have more granularity. That is why the speed-up in this version is far better than in leaf strategy. Furthermore, contrary to leaf strategy, in this version there is not any specific thread that is exclusively dedicated to generating tasks, all of them generate tasks. However, the biggest problem with this version is the complexity of distributing the work between the threads, but as we see before in figure 16 the workload is distributed equally.

2.3 Optimization: Cut-off mechanism

Lastly we improve the tree strategy by adding a cut-off in order to control the maximum recursion level for task generation and to improve the granularity.

To do this we add the parameter ***'d'*** and ***#pragma omp task final()*** to the multisort and merge functions and we increment it by one for each recursive call in order to control tasks generation.

CUT-OFF 0:

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=0
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************
******************>
Initialization time in seconds: 0.681552
Multisort execution time: 2.738405
Check sorted data execution time: 0.011302
Multisort program finished
********************************************************************
******************
```

CUT-OFF 1:

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=1
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************
******************>
Initialization time in seconds: 0.681056
Multisort execution time: 2.638648
Check sorted data execution time: 0.011853
Multisort program finished
********************************************************************
******************
```

CUT-OFF 2:

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=2
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************
******************>
Initialization time in seconds: 0.680207
Multisort execution time: 2.637996
Check sorted data execution time: 0.011917
Multisort program finished
********************************************************************
```

CUT-OFF 4:

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=4
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************
******************>
Initialization time in seconds: 0.681654
Multisort execution time: 2.643796
Check sorted data execution time: 0.012008
Multisort program finished
********************************************************************
******************
```

CUT-OFF 8:

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=4
Number of threads in OpenMP:          OMP_NUM_THREADS=2
********************************************************************
******************>
Initialization time in seconds: 0.681654
Multisort execution time: 2.643796
Check sorted data execution time: 0.012008
Multisort program finished
********************************************************************
******************
```

Figure 20. Times of tree strategy with cut-off.

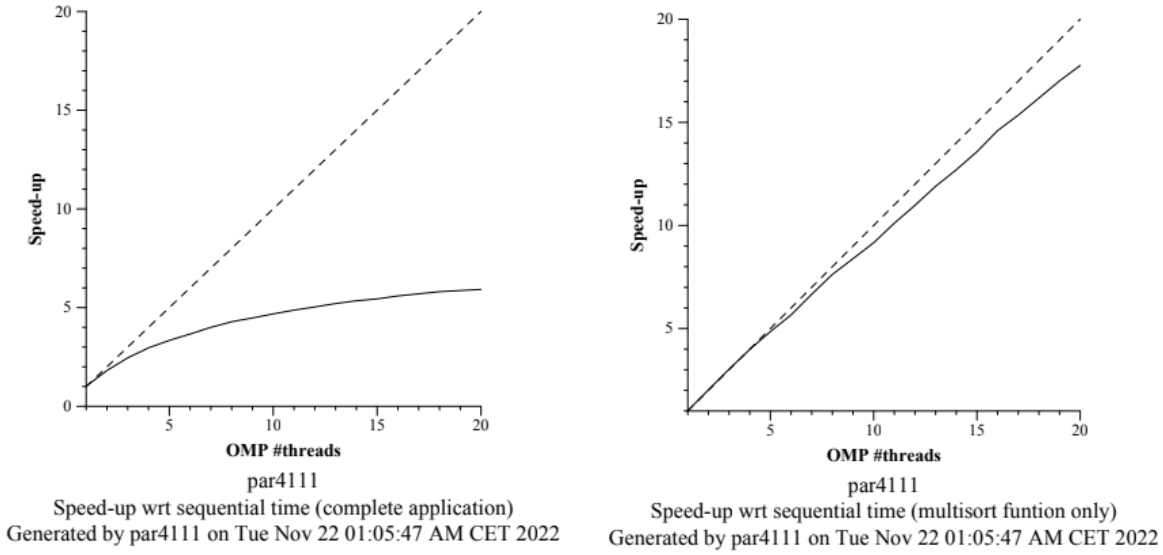

Figure 21. Scalability plots of tree strategy with cut-off.

Adding the cut-off we can observe an improvement in the speed-up on the multisort function. Speed-up is ideal until using 5 threads and nearly ideal as we increase threads.
This is due to the fact that the task creation is as before but at a certain depth in the recursion the tasks stop being created and the program continues sequentially.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.19	0.13	0.10	0.09	0.08	0.08	0.08	0.08	0.08
Speedup	1.00	1.40	1.91	2.20	2.39	2.49	2.48	2.36	2.31
Efficiency	1.00	0.70	0.48	0.37	0.30	0.25	0.21	0.17	0.14

Table 1: Analysis done on Tue Nov 22 05:06:57 PM CET 2022, par4111

Figure 22. Table 1 using cut-off.

Seeing the results and comparing to the tree strategy without the cut-off. We can observe an overall improvement and a huge time improvement. In addition we see a huge improvement in terms of speedup.

Overview of the Efficiency metrics in parallel fraction, ϕ=88.14%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	94.98%	70.30%	53.06%	42.48%	35.74%	30.50%	25.48%	20.35%	17.36%
Parallelization strategy efficiency	94.98%	74.24%	63.14%	55.10%	45.76%	39.54%	32.85%	26.53%	22.76%
Load balancing	100.00%	98.22%	97.59%	98.31%	97.74%	92.99%	91.99%	86.74%	91.40%
In execution efficiency	94.98%	75.59%	64.70%	56.05%	46.82%	42.52%	35.71%	30.59%	24.91%
Scalability for computation tasks	100.00%	94.69%	84.05%	77.09%	78.10%	77.15%	77.58%	76.71%	76.28%
IPC scalability	100.00%	87.46%	78.44%	76.05%	76.80%	77.52%	78.39%	78.10%	77.46%
Instruction scalability	100.00%	107.98%	107.94%	107.91%	107.91%	107.94%	107.88%	107.89%	107.96%
Frequency scalability	100.00%	100.26%	99.26%	93.93%	94.24%	92.20%	91.73%	91.03%	91.21%

Table 2: Analysis done on Tue Nov 22 05:06:57 PM CET 2022, par4111

Figure 23. Table 2 using cut-off.

In terms of efficiency we observe an improvement in global efficiency and parallelization efficiency, that is caused because the sequential part reduces the overhead of the task creation-. We also observe an improvement in the load balance.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0
LB (number of explicit tasks executed)	1.0	1.0	0.93	0.94	0.93	0.95	0.96	0.97	0.96
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.97	0.95	0.97	0.96	0.95
Time per explicit task (average us)	5.8	7.15	8.72	10.24	11.37	12.74	14.54	17.29	19.6
Overhead per explicit task (synch %)	1.32	19.82	29.15	36.58	46.07	52.53	59.65	68.19	73.58
Overhead per explicit task (sched %)	4.31	11.87	20.05	27.19	36.45	43.64	52.41	60.84	66.69
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Tue Nov 22 05:06:57 PM CET 2022, par4111

Figure 24. Table 3 using cut-off.

In the last table we see that the number of explicit tasks executed has been reduced to 25557 instead of 99669. We also observe that the time added by overheads has slightly decreased.

To study the instantaneous parallelism we used a cut-off of 1 in order to see how the different threads instantiate tasks.

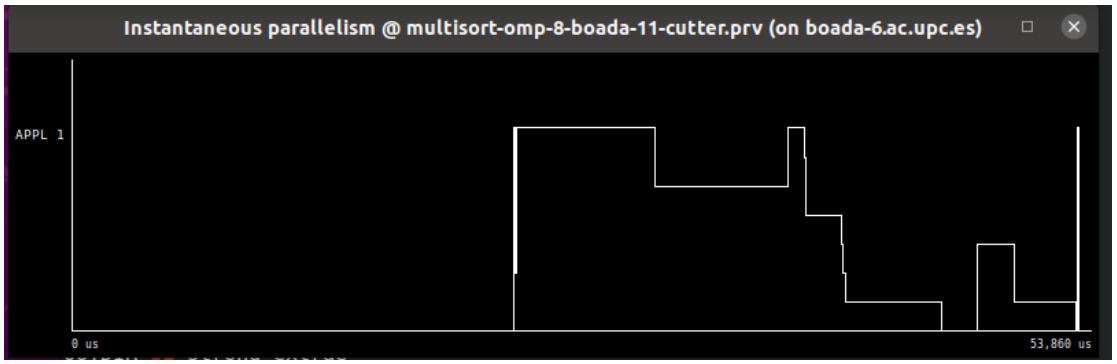

Figure 25. Instantaneous parallelism using 1 of cut-off.

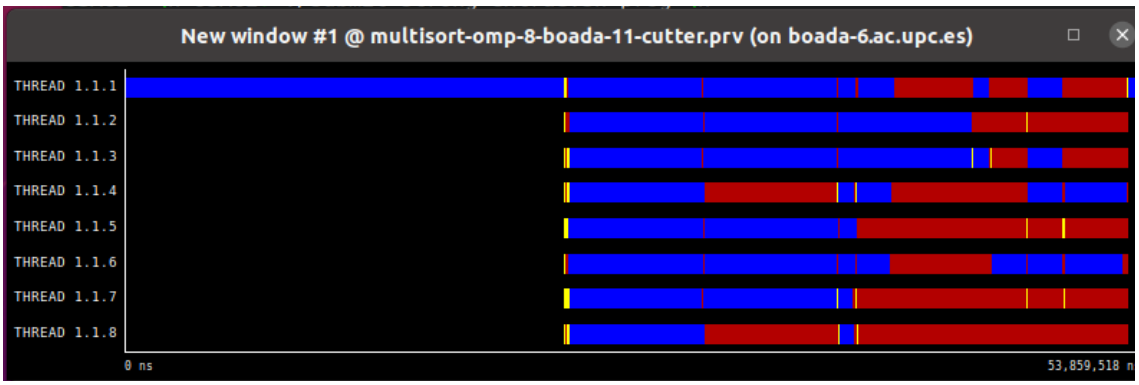

Figure 26. Timeline using 1 of cut-off.

In figures 25 and 26 we can observe that the second level of recursion tasks are also created by various threads. Compared to the initial version we see a big change in terms of granularity. We have bigger tasks which cause synchronization overheads but reduce the time invested in scheduling.

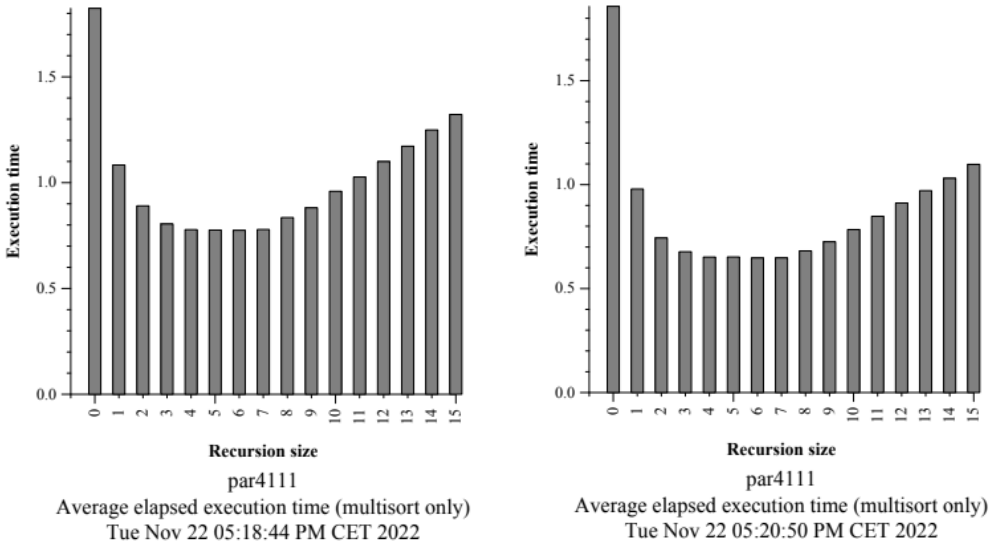

Figure 27. Plots with 6 of cut-off and 8 and 10 threads respectively.

Observing the results we see that the best value for cut-off is 6. Moreover if we change the number of threads the overall performance changes, with the data we conclude that the better performance is achieved between 8 and 10 threads as we can see in the images.

3 Shared-memory parallelisation with OpenMp task using dependencies

In this funeral session we modify the tree parallelisation in order to express dependencies among tasks and avoid some of the ***taskwait/taskgroup*** synchronization.
We avoided the taskwait and taskgroup clauses that we used before and we add ***#pragma omp depen(in:)*** and ***#pragma omp depen(out:).***

Next we compiled the new version using 8 threads and using a cut-off of 6 considering the previous results.

```
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                  CUTOFF=6
Number of threads in OpenMP:        OMP_NUM_THREADS=8
*****************************************************************************************
Initialization time in seconds: 0.683843
Multisort execution time: 0.700728
Check sorted data execution time: 0.012482
Multisort program finished
*****************************************************************************************
```

Figure 28. Time using dependencies.

Looking at the multisort execution time we see that it has decreased in a significant way compared to the last strategy used and we see a huge improvement.

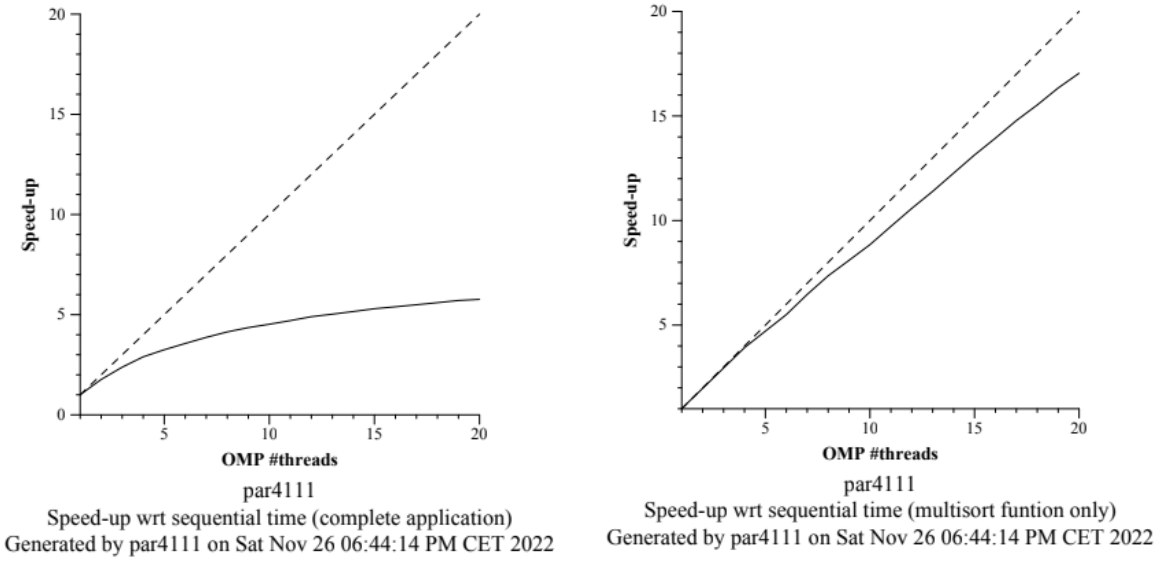

Figure 29. Scalability plots of tree strategy with dependence.

If we compare the plots with the previous version we see that they are very similar in performance and there is no evolution. In fact the previous version seems better.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.19	0.14	0.10	0.09	0.09	0.09	0.09	0.08	0.09
Speedup	1.00	1.37	1.87	2.17	2.19	2.23	2.19	2.29	2.22
Efficiency	1.00	0.69	0.47	0.36	0.27	0.22	0.18	0.16	0.14

Table 1: Analysis done on Sat Nov 26 06:36:04 PM CET 2022, par4111

Figure 30. Table 1 using dependencies.

Overview of the Efficiency metrics in parallel fraction, ϕ=87.86%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	93.68%	67.77%	49.85%	40.78%	31.36%	25.69%	20.92%	18.98%	15.99%
Parallelization strategy efficiency	93.68%	73.38%	60.18%	53.05%	41.14%	34.44%	28.29%	25.42%	21.56%
Load balancing	100.00%	98.44%	98.33%	94.20%	97.73%	94.19%	93.51%	95.45%	87.10%
In execution efficiency	93.68%	74.54%	61.20%	56.32%	42.10%	36.57%	30.26%	26.63%	24.76%
Scalability for computation tasks	100.00%	92.36%	82.83%	76.87%	76.22%	74.59%	73.95%	74.67%	74.16%
IPC scalability	100.00%	86.94%	79.34%	77.19%	77.11%	77.18%	77.17%	78.15%	77.68%
Instruction scalability	100.00%	106.93%	106.81%	106.81%	106.82%	106.78%	106.72%	106.70%	106.71%
Frequency scalability	100.00%	99.35%	97.75%	93.24%	92.53%	90.51%	89.80%	89.54%	89.47%

Table 2: Analysis done on Sat Nov 26 06:36:04 PM CET 2022, par4111

Figure 31. Table 2 using dependencies.

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0
LB (number of explicit tasks executed)	1.0	1.0	1.0	0.96	0.97	0.96	0.95	0.97	0.95
LB (time executing explicit tasks)	1.0	0.99	0.99	0.99	0.99	0.99	0.98	0.98	0.96
Time per explicit task (average us)	5.63	7.6	9.74	11.61	14.7	17.64	21.47	23.4	27.51
Overhead per explicit task (synch %)	2.81	20.76	30.59	35.67	43.65	48.53	52.1	54.62	57.47
Overhead per explicit task (sched %)	4.54	10.99	19.77	25.21	34.75	40.3	45.89	48.46	51.99
Number of taskwait/taskgroup (total)	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0

Table 3: Analysis done on Sat Nov 26 06:36:04 PM CET 2022, par4111

Figure 32. Table 3 using dependencies.

Observing the tables we concluded that the overall program has maintained his efficiency and speedup although it is slightly lower than before.
However, we can observe an improvement in the parallelization strategy efficiency and a reduction of the overhead time as we increased the number of processors.

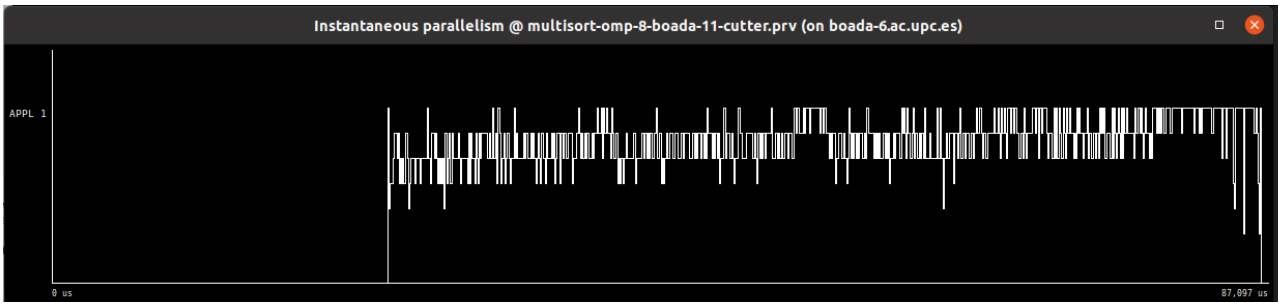

Figure 33. Instantaneous parallelism using dependencies.

We see that the parallelism in the code decreases at the beginning and at the end of the parallelization showing a worse performance than the previous strategy.

To conclude this section we could affirm that parallelism of multisort function has better results with the tree strategy using cut-off. Looking at the results we obtained we cleary see that it is the best option in this case.