PAR Laboratory Assignment
Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

David Cañadas López and Igor Duran Gallart, group 42,
19 October 2023, par4204 and par4206 (respectively)

November 30th, 2023

1 Laboratory 4 notebook

1.1 Task decomposition analysis with Tareador

1.1.1 Leaf and Tree Analysis with Tareador

This part refers to subsection 2.2 of the practice document.

Include the TDG of both leaf and tree strategies Tareador analysis.
Comments/Observations

Identify the points of the code where you should include synchronizations to guarantee the dependences for each task decomposition strategy. Which directives/annotations/clauses will you use to guarantee those dependencies in the OpenMP implementations?

The resulting TDG of the **leaf strategy**:

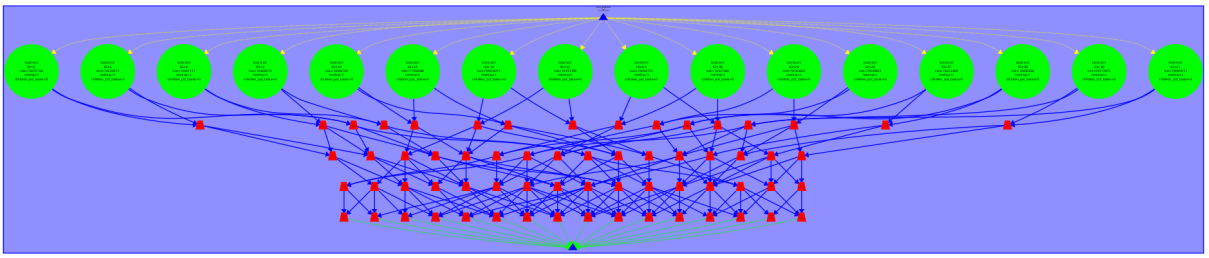


Figure 1. - Leaf strategy TDG using Tareador.

To ensure the correct synchronization of the tasks of this strategy, we can make use of the *taskwait* directive after the four recursive calls to multisort to wait for any sorting task created recursively by the same thread to finish before proceeding with the merging into the tmp vector to avoid data races. We must also add another *taskwait* just before the last call to merge, since the tasks created by the merge function calls must have finished merging the results into the tmp vector before the final merge into the data vector.

The resulting TDG of the **tree strategy**:

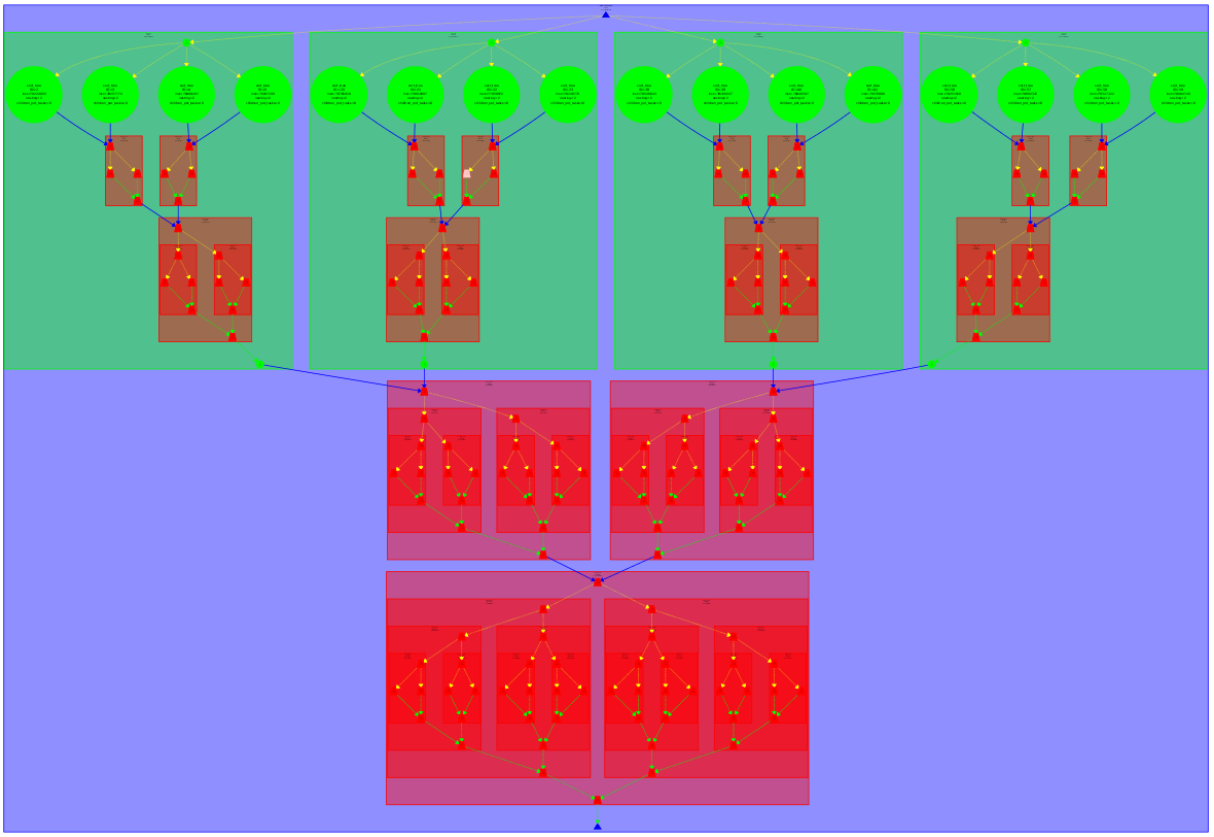


Figure 2. - Tree strategy TDG using Tareador.

To ensure the correct synchronization of the tasks of this strategy, we could also use the *taskwait* directive after creating the tasks on the recursive case of each function. This is to make sure that the vector's elements that each function call operates on have been correctly sorted before proceeding with the program, similarly to the leaf strategy. Two additional *taskwait*s are required: one just before creating the last merge task of the recursive case on multisort, so that each half of the tmp vector is properly sorted before the final merge into the data vector, and another right after that same call, to ensure the task doesn't end before their child tasks have finished, which would cause data races.

1.2 Leaf and Tree strategies in OpenMP

1.2.1 Analysis with modelfactors

This part refers to point 4 of subsection 3.1 and subsection 3.2 of the practice document.

Include the three tables generated for the leaf and tree strategies. Neither leaf nor tree strategies are showing a good parallel performance.

Comments/Observations

Which of the factors do you think is making the parallelisation efficiency so low in the case of the leaf ? Which of the following parameters do you think is making the tree parallelisation better than the leaf version? Several options to think about: parallel fraction, in-execution efficiency (related with the overheads of sync and sched reported in the third table), number of tasks and their execution time, load balancing, ... Is the granularity of both (leaf and tree) strategies in using the parallel performance (see overheads in modelfactors tables for both strategies)? What is the number of tasks created (see modelfactors tables for both strategies)?

The resulting tables for the **leaf strategy**:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.28	0.43	0.44	0.46	0.45
Speedup	1.00	0.64	0.62	0.60	0.62
Efficiency	1.00	0.16	0.08	0.05	0.04

Table 1: Analysis done on Thu Nov 23 03:43:52 PM CET 2023, par4204

Figure 3. - Execution metrics of the OpenMP leaf strategy.

Overview of the Efficiency metrics in parallel fraction, ϕ=92.47%					
Number of processors	1	4	8	12	16
Global efficiency	73.32%	11.39%	5.56%	3.55%	2.75%
Parallelization strategy efficiency	73.32%	16.34%	8.31%	5.65%	4.49%
Load balancing	100.00%	58.25%	27.92%	18.87%	12.87%
In execution efficiency	73.32%	28.05%	29.78%	29.94%	34.87%
Scalability for computation tasks	100.00%	69.72%	66.85%	62.74%	61.36%
IPC scalability	100.00%	62.20%	63.37%	62.69%	63.86%
Instruction scalability	100.00%	111.99%	112.17%	112.30%	112.03%
Frequency scalability	100.00%	100.10%	94.06%	89.13%	85.76%

Table 2: Analysis done on Thu Nov 23 03:43:52 PM CET 2023, par4204

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.94	0.44	0.29	0.16
LB (time executing explicit tasks)	1.0	0.93	0.47	0.31	0.18
Time per explicit task (average us)	2.61	3.64	3.85	3.99	3.99
Overhead per explicit task (synch %)	16.63	696.14	1496.28	2335.35	3046.38
Overhead per explicit task (sched %)	9.88	15.82	14.81	14.09	13.71
Number of taskwait/taskgroup (total)	53248.0	53248.0	53248.0	53248.0	53248.0

Table 3: Analysis done on Thu Nov 23 03:43:52 PM CET 2023, par4204

Figure 4. - Efficiency and task metrics of the OpenMP leaf strategy.

The resulting tables for the **tree strategy**:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.31	0.25	0.23	0.23	0.23
Speedup	1.00	1.23	1.39	1.39	1.34
Efficiency	1.00	0.31	0.17	0.12	0.08

Table 1: Analysis done on Thu Nov 16 03:53:15 PM CET 2023, par4204

Figure 5. - Execution metrics of the OpenMP tree strategy.

Overview of the Efficiency metrics in parallel fraction, ϕ=93.35%					
Number of processors	1	4	8	12	16
Global efficiency	74.76%	23.50%	13.34%	8.92%	6.46%
Parallelization strategy efficiency	74.76%	37.62%	23.87%	16.99%	12.28%
Load balancing	100.00%	95.32%	92.85%	91.35%	91.51%
In execution efficiency	74.76%	39.47%	25.71%	18.59%	13.42%
Scalability for computation tasks	100.00%	62.46%	55.89%	52.52%	52.63%
IPC scalability	100.00%	53.55%	50.16%	50.21%	52.08%
Instruction scalability	100.00%	119.10%	118.99%	119.04%	119.02%
Frequency scalability	100.00%	97.94%	93.62%	87.87%	84.89%

Table 2: Analysis done on Thu Nov 16 03:53:15 PM CET 2023, par4204

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.96	0.97	0.96	0.98
LB (time executing explicit tasks)	1.0	0.99	0.98	0.98	0.98
Time per explicit task (average us)	1.73	6.55	11.55	17.31	23.85
Overhead per explicit task (synch %)	11.19	51.7	59.56	62.29	64.6
Overhead per explicit task (sched %)	13.82	37.35	48.75	55.4	59.67
Number of taskwait/taskgroup (total)	49152.0	49152.0	49152.0	49152.0	49152.0

Table 3: Analysis done on Thu Nov 16 03:53:15 PM CET 2023, par4204

Figure 6. - Efficiency and task metrics of the OpenMP tree strategy.

In terms of efficiency the leaf strategy is performing much worse than the tree strategy, as seen in Figures 3 and 5. Analyzing Figures 4 and 6, we can conclude that this poor performance does not have to do with the parallel fraction (since both have about the same), nor the in-execution efficiency (the tree strategy does also perform badly in this metric), but rather the huge amount of tasks created an the fact that in the leaf strategy they are being created sequentially by the main thread (and they have dependencies with one another), while in the tree strategy the tasks are created in parallel by all threads. This, in turn, introduces a load balance problem in the leaf strategy which makes it perform worse.
We can see that in the leaf strategy 53248 tasks are generated on any case (Figure 4 Table 3), while in the tree strategy 99669 tasks are created, also on any of the cases (Figure 6: Table 3).

1.2.2 Analysis with Paraver

This part refers to point 5 of subsection 3.1 and subsection 3.2 of the practice document.

Include parts (zoom in) of the Paraver visualizations that help you explain the lack of scalability. In particular, and for both strategies, we think it would be good to show those parts that show examples of

1) the amount of tasks executed in parallel,
2) which threads are executing tasks and
3) which thread/s is/are creating tasks?.

Comments/Observations

Is the program generating enough tasks to simultaneously feed all threads? How many? How many threads are creating tasks?

The resulting timeline for the **leaf strategy**:

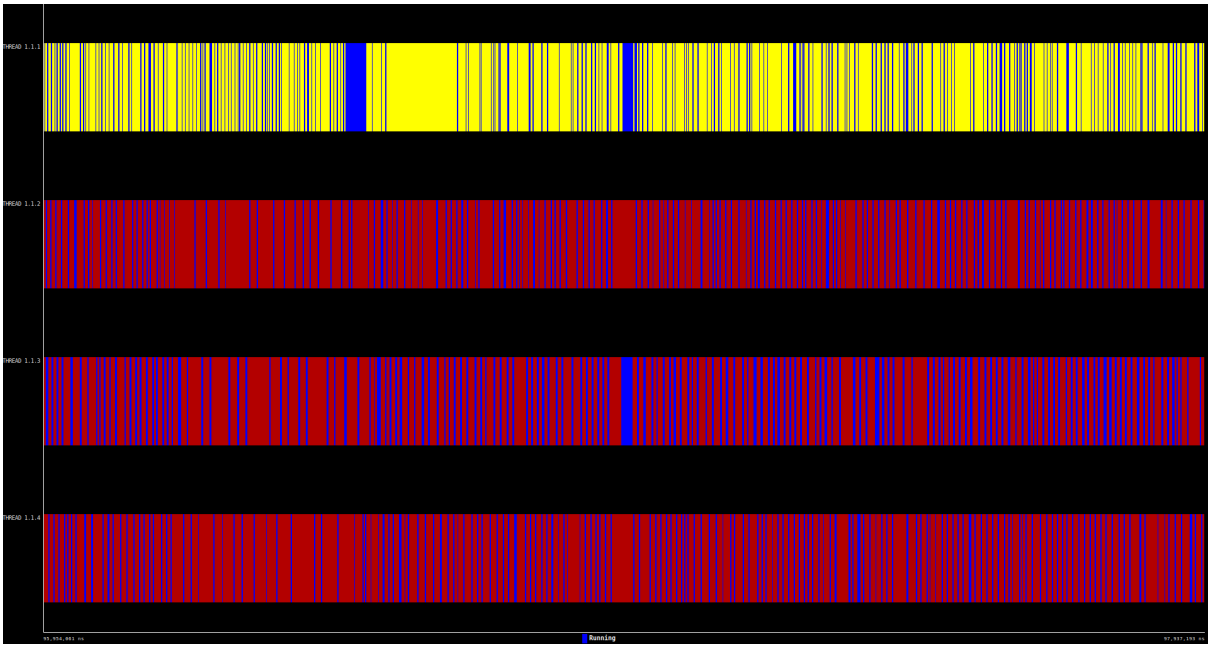


Figure 7. - Extract of the Paraver timeline for 4 threads on leaf strategy (red: synchronization, blue: running, yellow: scheduling and fork/join)

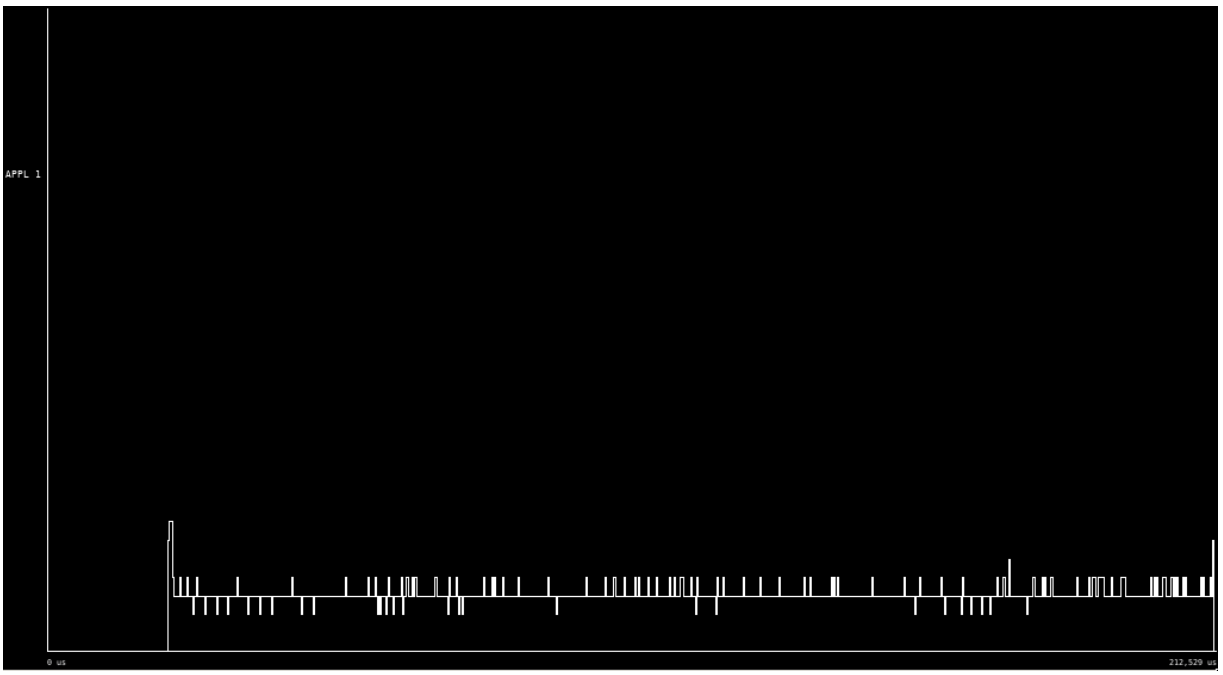


Figure 8. - Tasks executed in parallel for 8 threads on leaf strategy

As we can see in Figure 7, in the leaf strategy only one thread creates the tasks sequentially, and all of the threads execute them. Figure 8 shows how, on average, 4 tasks are executed in parallel when using 8 threads.

The resulting timeline for the **tree strategy**:

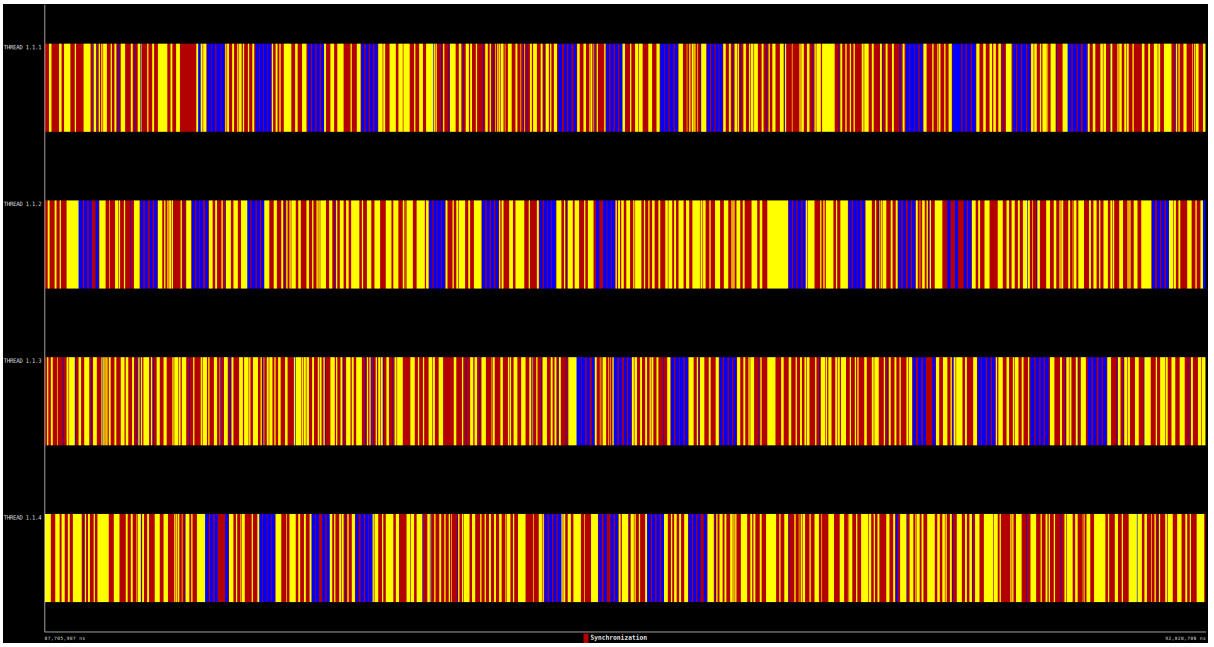


Figure 9. - Extract of the Paraver timeline for 4 threads on tree strategy (red: synchronization, blue: running, yellow: scheduling and fork/join)

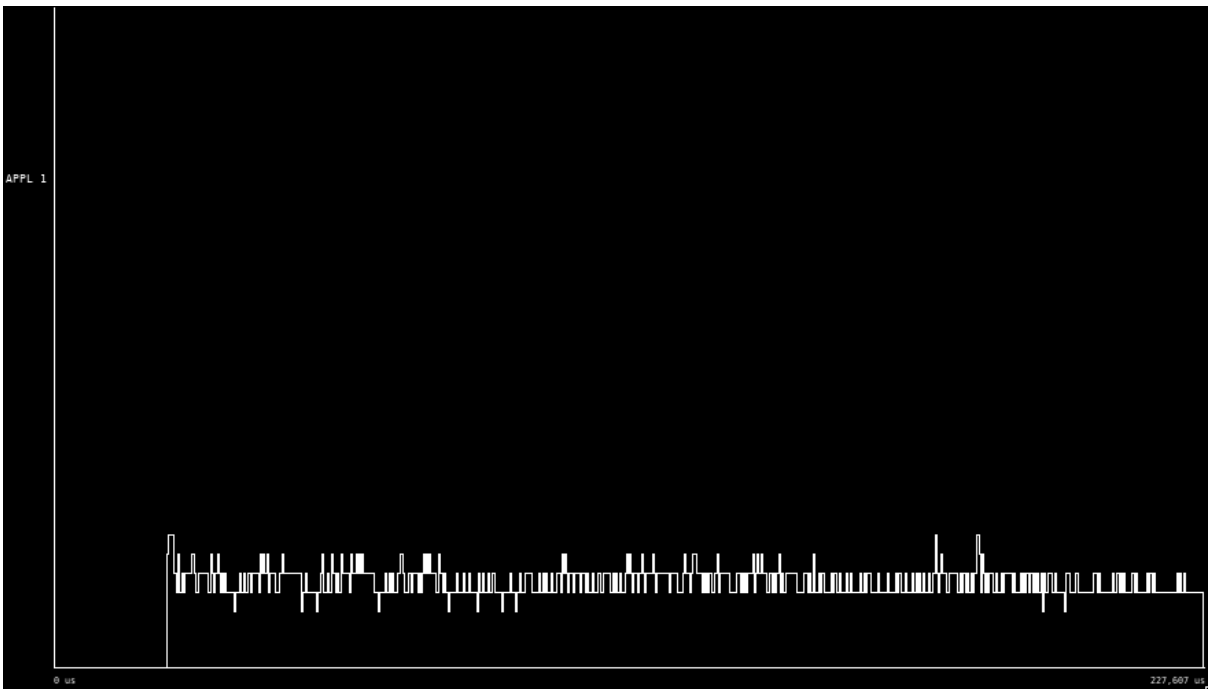


Figure 10. - Tasks executed in parallel for 8 threads on tree strategy

As we can see in Figure 9, in the tree strategy all threads create the tasks in parallel, and also all of the threads execute them. Figure 10 shows how, on average, 6 tasks are executed in parallel when using 8 threads, which is better than the leaf strategy, as we have already analyzed.

1.3 Task granularity control: the cutoff mechanism

1.3.1 Analysis with modelfactors

This part refers to points 4 and 5 of subsection 3.3 of the practice document.

Include the tables of modelfactors for cut-off level equal to 4 and the information of the number of tasks generated for each of the cut-off levels used.

Comments/Observations

Which is the best value for cut-off ? Does it significantly change with the number of threads used?

The modelfactors tables generated for a cutoff value of 4 are the following:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.15	0.06	0.04	0.04	0.04
Speedup	1.00	2.55	3.41	3.89	3.97
Efficiency	1.00	0.64	0.43	0.32	0.25

Table 1: Analysis done on Tue Nov 28 08:55:44 PM CET 2023, par4204

Figure 11. - Execution metrics of the OpenMP tree strategy with cutoff=4.

Overview of the Efficiency metrics in parallel fraction, ϕ=85.96%					
Number of processors	1	4	8	12	16
Global efficiency	98.79%	85.16%	68.94%	60.59%	49.01%
Parallelization strategy efficiency	98.79%	91.97%	82.82%	77.45%	68.91%
Load balancing	100.00%	98.88%	96.50%	96.68%	92.29%
In execution efficiency	98.79%	93.01%	85.82%	80.11%	74.67%
Scalability for computation tasks	100.00%	92.60%	83.24%	78.23%	71.13%
IPC scalability	100.00%	94.55%	92.23%	91.40%	88.61%
Instruction scalability	100.00%	101.19%	101.17%	101.16%	101.13%
Frequency scalability	100.00%	96.79%	89.21%	84.62%	79.38%

Table 2: Analysis done on Tue Nov 28 08:55:44 PM CET 2023, par4204

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3317.0	3317.0	3317.0	3317.0	3317.0
LB (number of explicit tasks executed)	1.0	1.0	0.96	0.97	0.88
LB (time executing explicit tasks)	1.0	0.98	0.97	0.93	0.94
Time per explicit task (average us)	38.53	42.21	46.49	48.95	53.61
Overhead per explicit task (synch %)	0.53	5.12	13.39	15.44	24.17
Overhead per explicit task (sched %)	0.71	1.94	3.39	4.93	8.42
Number of taskwait/taskgroup (total)	1488.0	1488.0	1488.0	1488.0	1488.0

Table 3: Analysis done on Tue Nov 28 08:55:44 PM CET 2023, par4204

Figure 12. - Efficiency and task metrics of the OpenMP tree strategy with cutoff=4.

The amount of tasks generated for each cutoff level is the following:

CUTOFF	Number of tasks generated
0	1
1	41
2	189
4	3317
8	57173

As expected, a cutoff value of 0 implies a sequential execution in which only the main task created by the *single* directive executes the whole algorithm. Then, the higher the cutoff value, the more tasks are created recursively from the main one.

1.3.2 Analysis with Paraver

This part refers to point 5 of subsection 3.3 of the practice document.

Include the Paraver trace visualizations you have analyzed for the execution with 8 threads.

Comments/Observations

Is the instantaneous parallelism achieved larger than one along all the execution traces?

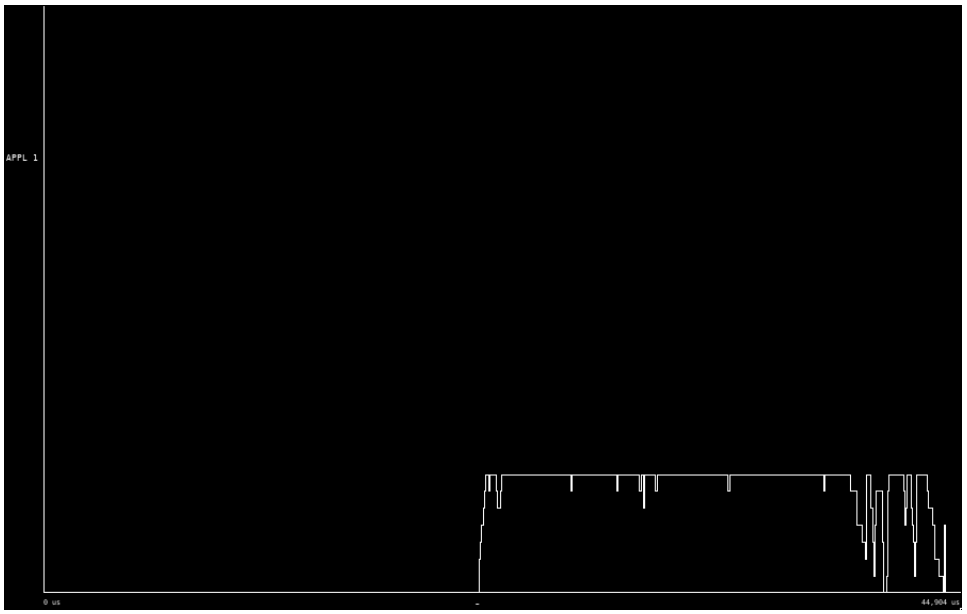


Figure 13. - Instantaneous parallelism across the execution with 8 threads and cutoff=4.

Figure 13 shows that during the most part of the execution, 8 tasks were able to be executed in parallel. Now, we will compare this result with the traces of the executions with other cutoff values:

Figure 14. - Instantaneous parallelism across the execution with 8 threads and cutoff=0.

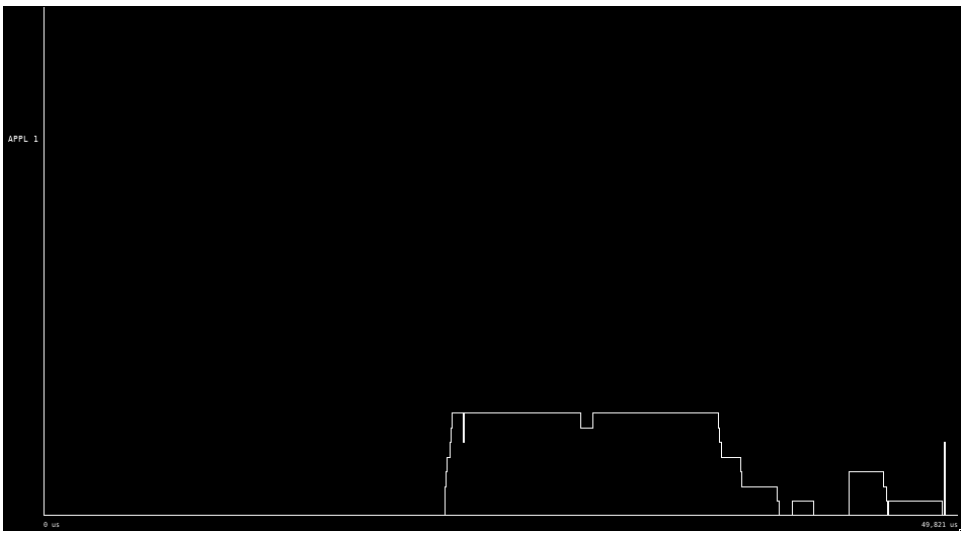


Figure 15. - Instantaneous parallelism across the execution with 8 threads and cutoff=1.

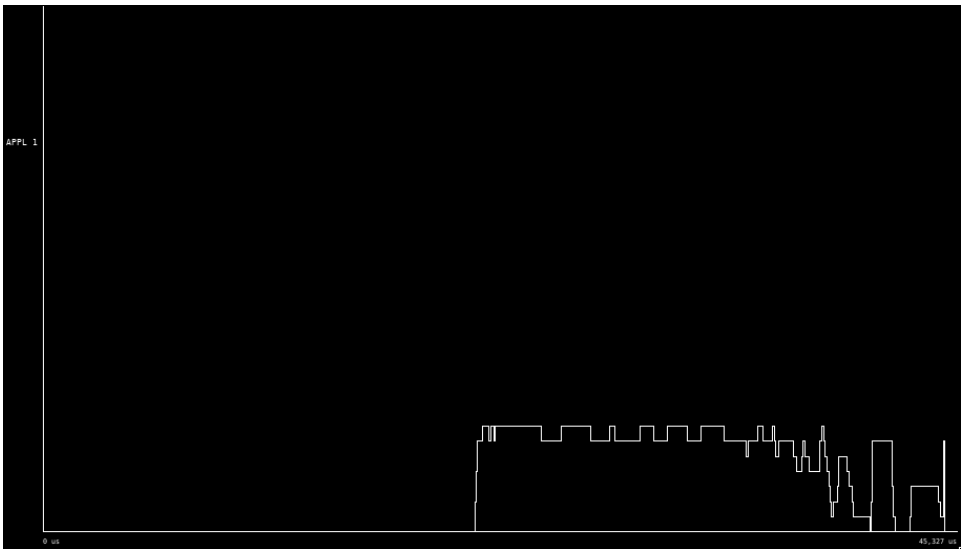


Figure 16. - Instantaneous parallelism across the execution with 8 threads and cutoff=2.

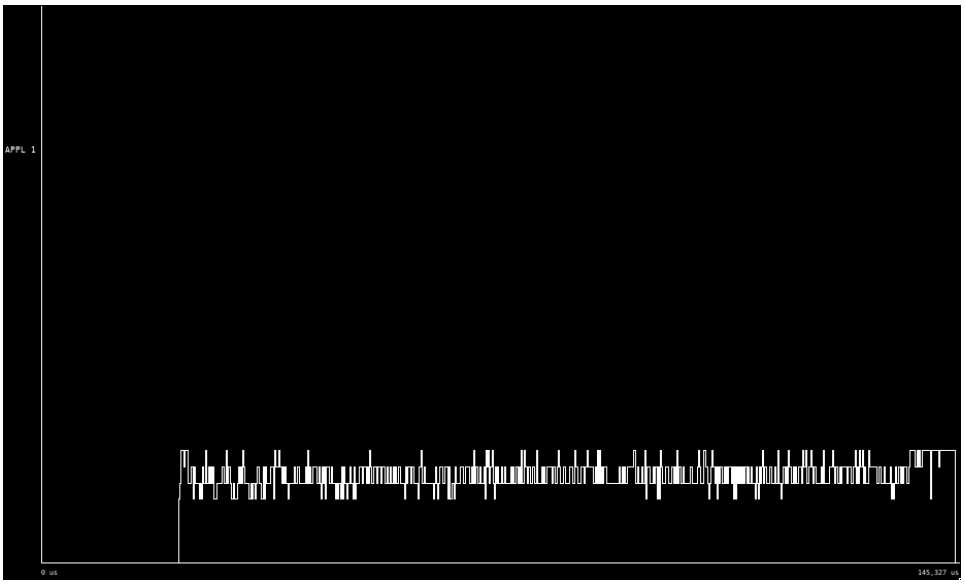


Figure 17. - Instantaneous parallelism across the execution with 8 threads and cutoff=8.

As can be observed in Figures 14, 15, 16 and 17, all the other cases for values of cutoff different from 8 have worse instant parallelism than the trace for cutoff=4 (Figure 13).

1.3.3 Granularity Analysis

This part refers to point 3 of subsection 3.2 and 7 of subsection 3.3 of the practice document.

Include the postscript levels generated by submit-cutoff-omp.sh to explore different cut-off levels for 8 threads.

Comments/Observations

Which is the best value for cut-off for this problem size (look at the script to gure out the size of the problem) and 8 threads?

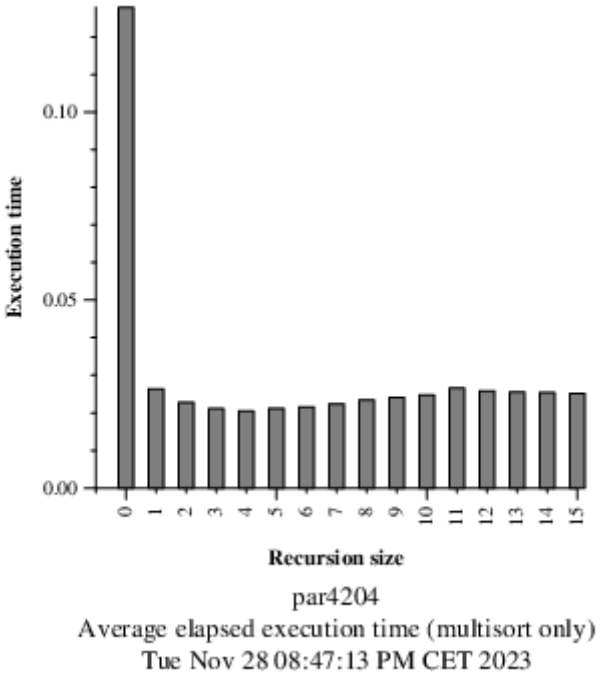


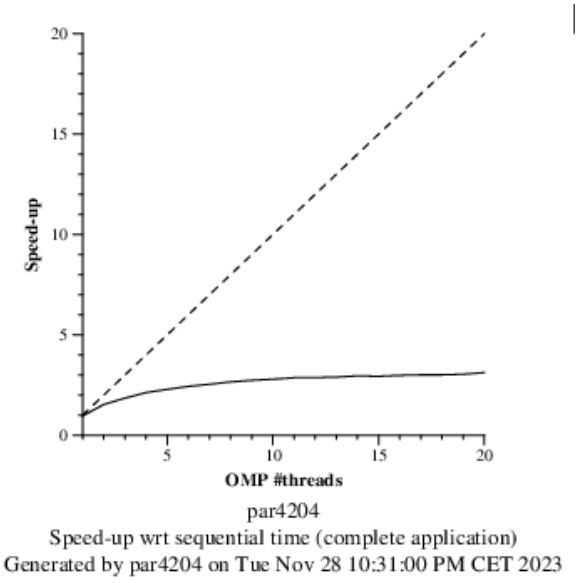
Figure 18. - Postscript graph for 8 threads on tree strategy with cutoff=4.

Figure 18 reveals that a cutoff value of around 4 is the optimum with 8 threads for the problem size the script uses to test by default (which are also the one used on the strong-extrae tests). It also shows that the execution time varies only slightly when more threads are introduced (and also fewer threads, excluding 0 is the fully sequential case) to execute the program in parallel.

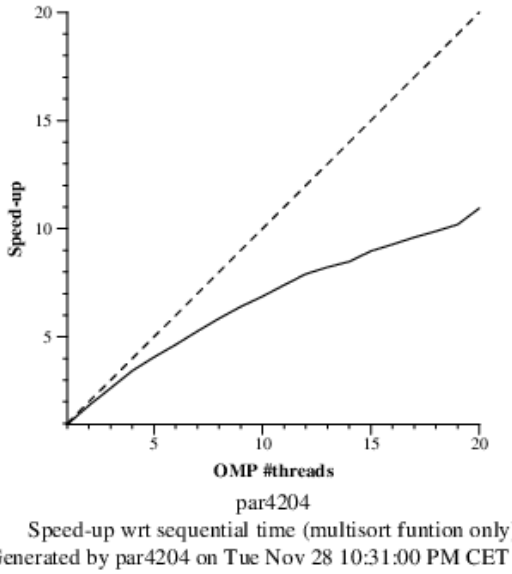
Include the strong scalability plot generated for the best cutoff level.

Comments/Observations

Are the speedup for the complete and multisort function close to the speed-up ideal? Reason the difference in speedup based on Paraver trace you have analyzed and the parallelization of your multisort-omp.c code.



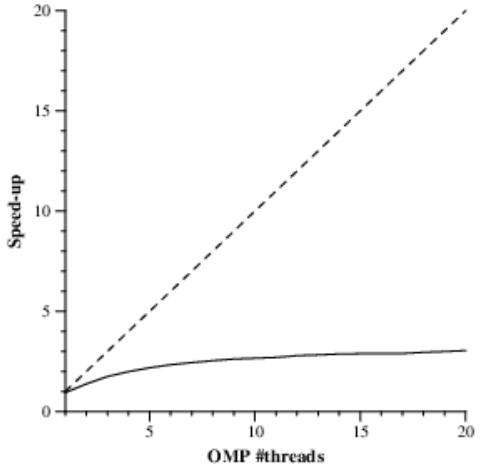
par4204
Speed-up wrt sequential time (complete application)
Generated by par4204 on Tue Nov 28 10:31:00 PM CET 2023



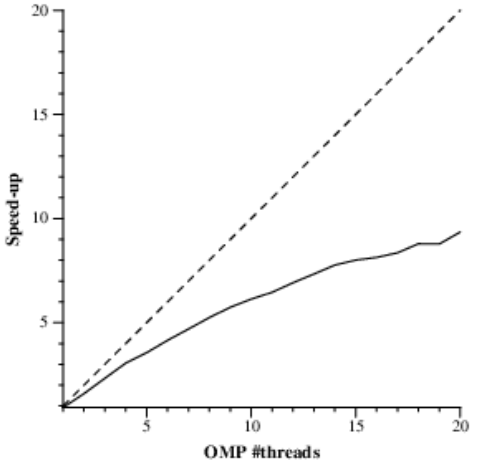
par4204
Speed-up wrt sequential time (multisort funtion only)
Generated by par4204 on Tue Nov 28 10:31:00 PM CET 2023

Figure 19. - Postscript graph for tree strategy with cutoff=4.

The results are promising. Now, we compare this graphs with those generated by the script with another cutoff value, such as 8:



par4204
Speed-up wrt sequential time (complete application)
Generated by par4204 on Tue Nov 28 11:31:44 PM CET 2023



par4204
Speed-up wrt sequential time (multisort funtion only)
Generated by par4204 on Tue Nov 28 11:31:44 PM CET 2023

Figure 19. - Postscript graph for tree strategy with cutoff=8.

As expected with Figures 18 and 17, the speedup for the strong scalability is slightly worse than the one in Figure 19. Even though it does not produce the ideal speedup, a cutoff value of 4 seems to be the best for this scenario.

2 Shared-memory parallelisation with OpenMP task using dependencies

This part refers to section 4 of the practice document.

Include: Tables of modelfactors. Strong Scalability plot. The Paraver traces analyzed. Comments/Observations

Is this parallel implementation better or worse compared to OpenMP versions of the previous chapter in terms of performance? In terms of programmability, was this new version simpler to code?

The modelfactors tables generated by the new version using de *depends* clause are the following:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.14	0.06	0.04	0.04	0.04
Speedup	1.00	2.56	3.32	3.53	3.76
Efficiency	1.00	0.64	0.41	0.29	0.24

Table 1: Analysis done on Wed Nov 29 12:23:48 AM CET 2023, par4204

Figure 20. - Execution metrics of the OpenMP tree strategy using dependencies with cutoff=4.

Overview of the Efficiency metrics in parallel fraction, ϕ=85.71%					
Number of processors	1	4	8	12	16
Global efficiency	98.92%	86.90%	68.57%	53.69%	45.06%
Parallelization strategy efficiency	98.92%	92.19%	83.19%	73.40%	65.56%
Load balancing	100.00%	99.62%	98.31%	95.11%	92.83%
In execution efficiency	98.92%	92.54%	84.62%	77.17%	70.62%
Scalability for computation tasks	100.00%	94.26%	82.43%	73.15%	68.74%
IPC scalability	100.00%	95.54%	91.39%	89.86%	88.32%
Instruction scalability	100.00%	100.89%	100.85%	100.81%	100.74%
Frequency scalability	100.00%	97.79%	89.44%	80.76%	77.25%

Table 2: Analysis done on Wed Nov 29 12:23:48 AM CET 2023, par4204

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3317.0	3317.0	3317.0	3317.0	3317.0
LB (number of explicit tasks executed)	1.0	1.0	0.94	0.9	0.91
LB (time executing explicit tasks)	1.0	1.0	0.96	0.91	0.94
Time per explicit task (average us)	36.51	39.71	45.61	51.14	54.98
Overhead per explicit task (synch %)	0.33	5.37	10.63	19.1	24.37
Overhead per explicit task (sched %)	0.78	1.79	4.5	6.93	11.37
Number of taskwait/taskgroup (total)	806.0	806.0	806.0	806.0	806.0

Table 3: Analysis done on Wed Nov 29 12:23:48 AM CET 2023, par4204

Figure 21. - Efficiency and task metrics of the OpenMP tree strategy using dependencies with cutoff=4.

When comparing the resulting metrics (Figures 20 and 21) with the metrics from the cutoff version (Figures 11 and 12) we are able to see that there has been improvement in the number of taskwaits/taskgroups, but tasks require more synchronization than before,

causing the overall efficiency and speedup to lower the more threads are used. Regarding programmability, this version is more difficult to code than the previous, since the clauses in the program are lengthy and error-prone, even if they are slightly more intuitive than using *taskwait*. All in all, this is a negative first impression.

We continue our analysis generating the strong scalability plot:

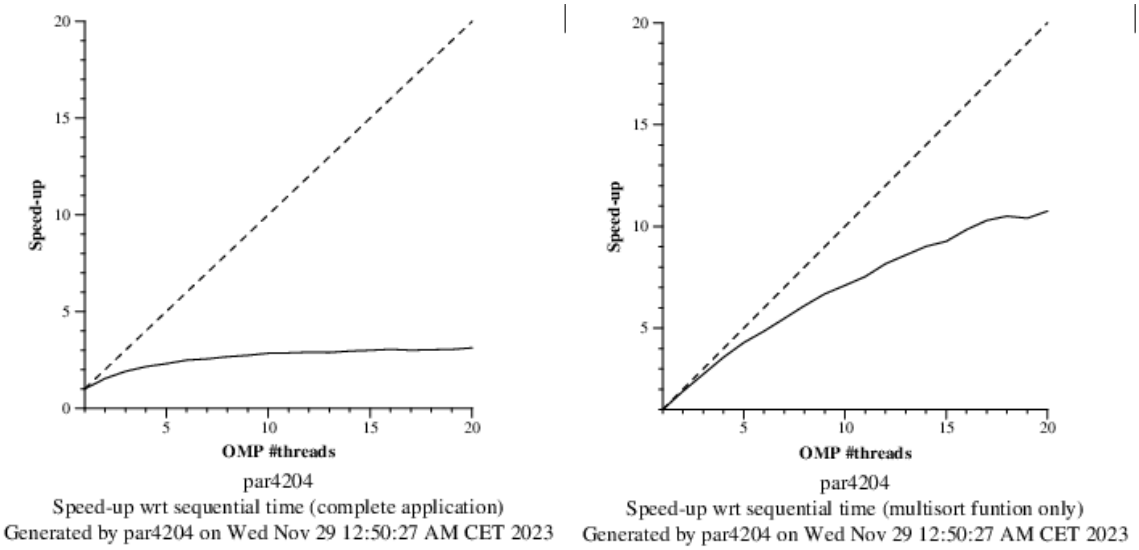


Figure 22. - Postscript graph for tree strategy using dependencies with cutoff=4.

We then again compare Figure 22 with Figure 19, seeing that the scalability of this strategy is not improved. To conclude the analysis, we present a relevant Paraver trace:

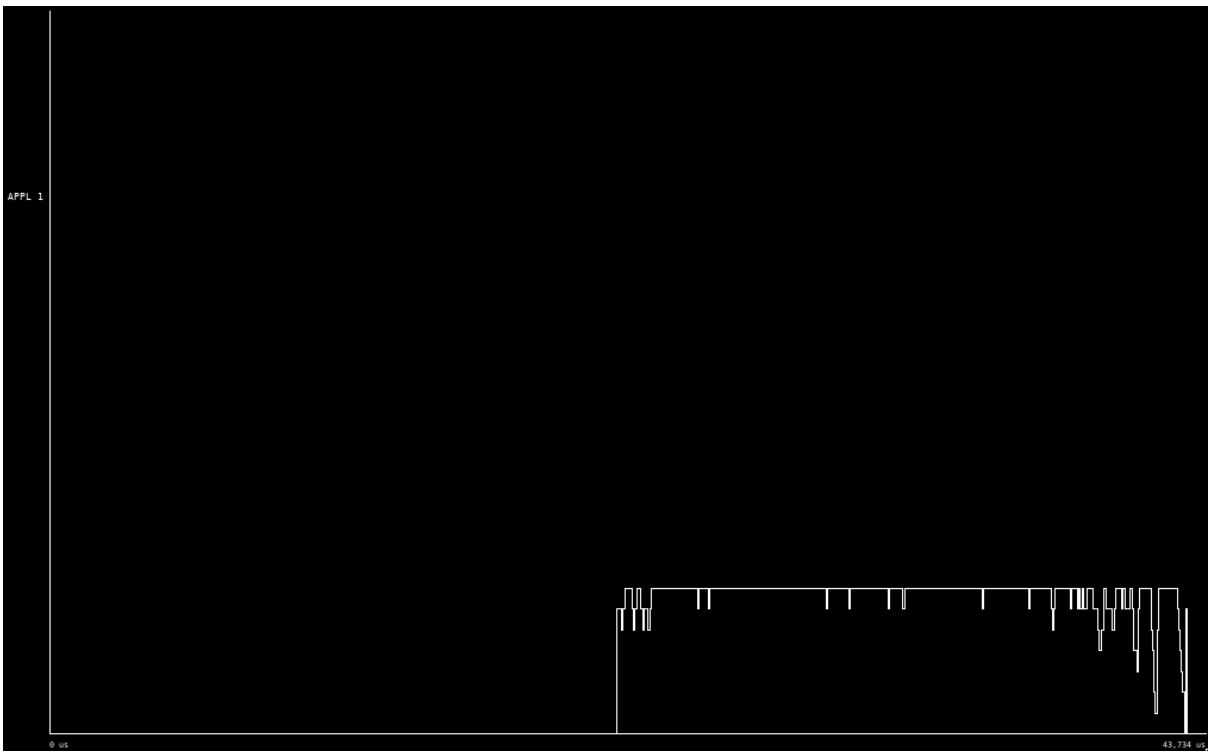


Figure 23. - Instantaneous parallelism across the execution with dependencies, 8 threads and cutoff=4.

Figure 23 and Figure 13 are almost identical, and as such any possible gain would be minimal in comparison with the reduction in efficiency and speedup. Therefore, we deem this strategy as a worse option than the one without dependencies.