LABORATORY 4

Group components: Albert Bausili Fernández, Noa-Yu Ventura Vila

Group identifier: par2301

01/12/2022

Index

1. Task decomposition analysis for Mergesort

1.1 "Divide and conquer"

We submitted multisort-seq.c and got the following output. It's important because we will use them as reference times to check the scalability of the parallel versions in OpenMP we will develop in another section of the project.

```
Initialization time in seconds: 0.683093

Multisort execution time: 5.185487

Check sorted data execution time: 0.011297
```

Figure 1. Results we obtained executing multisort-seq.c.

1.2 Task decomposition analysis with Tareador

In figure 2 we can see how we implemented the leaf decomposition strategy using tareador. We create a task for each base case, so creation of tasks is sequential.

```
…
// Base case
      tareador_start_task("basic_merge");
      basicmerge(n, left, right, result, start, length);
      tareador_end_task("basic_merge");

…
// Base case
      tareador_start_task("basic_sort");
      basicsort(n, data);
      tareador_end_task("basic_sort");
…
```

Figure 2. Excerpts of the code multisort-tareador.c using the leaf strategy.

```
…
// Recursive decomposition
       tareador_start_task("multisort1");
       multisort(n/4L, &data[0], &tmp[0]);
       tareador_end_task("multisort1");
       tareador_start_task("multisort2");
       multisort(n/4L, &data[n/4L], &tmp[n/4L]);
       tareador_end_task("multisort2");
…
…
// Recursive decomposition
       tareador_start_task("merge4");
       merge(n, left, right, result, start, length/2);
       tareador_end_task("merge4");
       tareador_start_task("merge5");
       merge(n, left, right, result, start + length/2, length/2);
       tareador_end_task("merge5");
…
```

Figure 3. Excerpts of the code multisort-tareador.c using the tree strategy.

In contrast with leaf strategy decomposition, we have tree strategy decomposition which *tareador* implementation is shown in figure 3. In this case, for each recursive call to merge and multisort we create a task, therefore the creation of tasks is made in parallel.

The following figure (figure 4) shows the task dependence graph of the leaf strategy, and figure 5 for the tree strategy. For the leaf strategy we can see that tasks are created sequentially and all of them have a similar granularity, unlike tree strategy where tasks are created in parallel and a task may have a granularity way bigger than another one. It should also be noted that there's a higher number of tasks for the tree strategy than the leaf strategy, because in the first one we create a task for each recursive call which is made way more times than the number of calls to the base case functions basicmerge and basicsort.

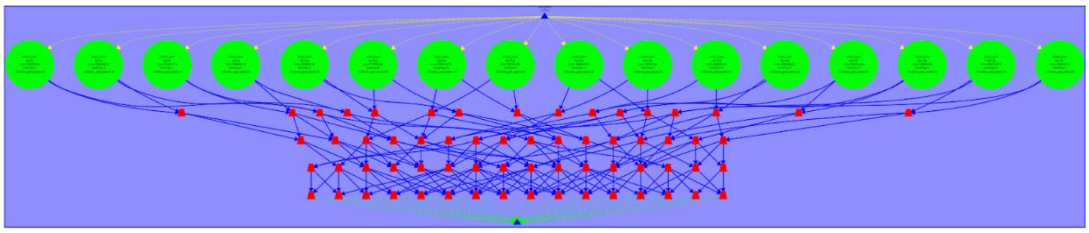


Figure 4. Task Dependence Graph of multisort-tareador.c using the leaf strategy.

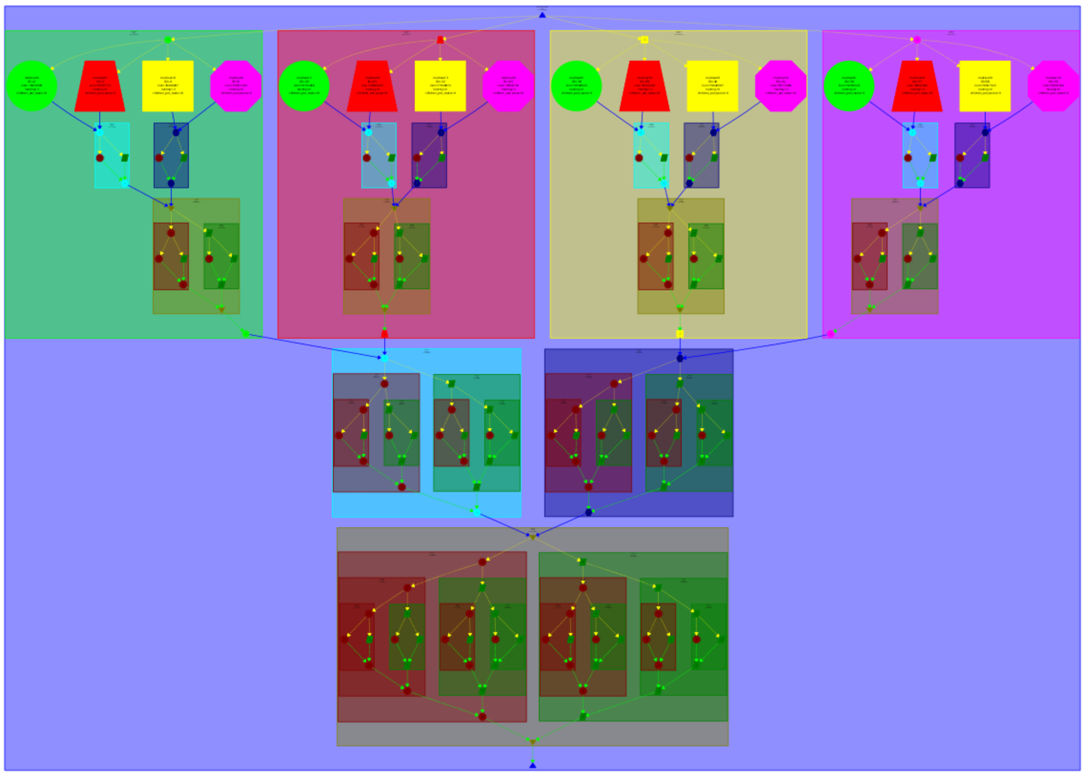


Figure 5. Task Dependence Graph of multisort-tareador.c using the tree strategy.

Creating tasks and parallelizing code may influence the result we expected from the function, which is caused by the dependencies between tasks. In this case, we have a strong dependency between the last call to multisort and the first call to merge (see figure 6 dependency 1) and between the second call to merge and the last call to merge (see figure 6 dependency 2). This is due to the nature of the sorting algorithm, which needs each of its blocks sorted in order to merge them into a sorted bigger block. This is, we need the four blocks obtained from multisort to be sorted, and then merge two of them together with a function merge getting two blocks, and finally call to the last function merge to sort the last two blocks together.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        multisort(n/4L, &data[0], &tmp[0]);
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

       //dependency 1

        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

      //dependency 2

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 6. Extract of the function multisort from multisort-tareador.c.

1.3 Paraver analysis

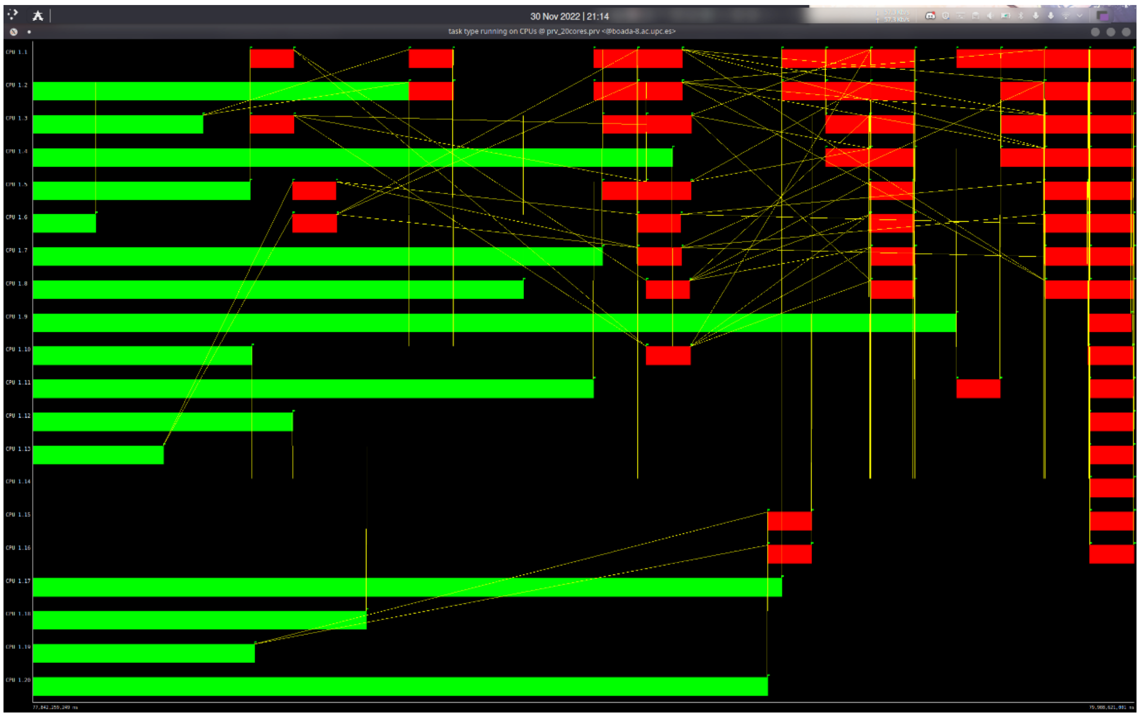
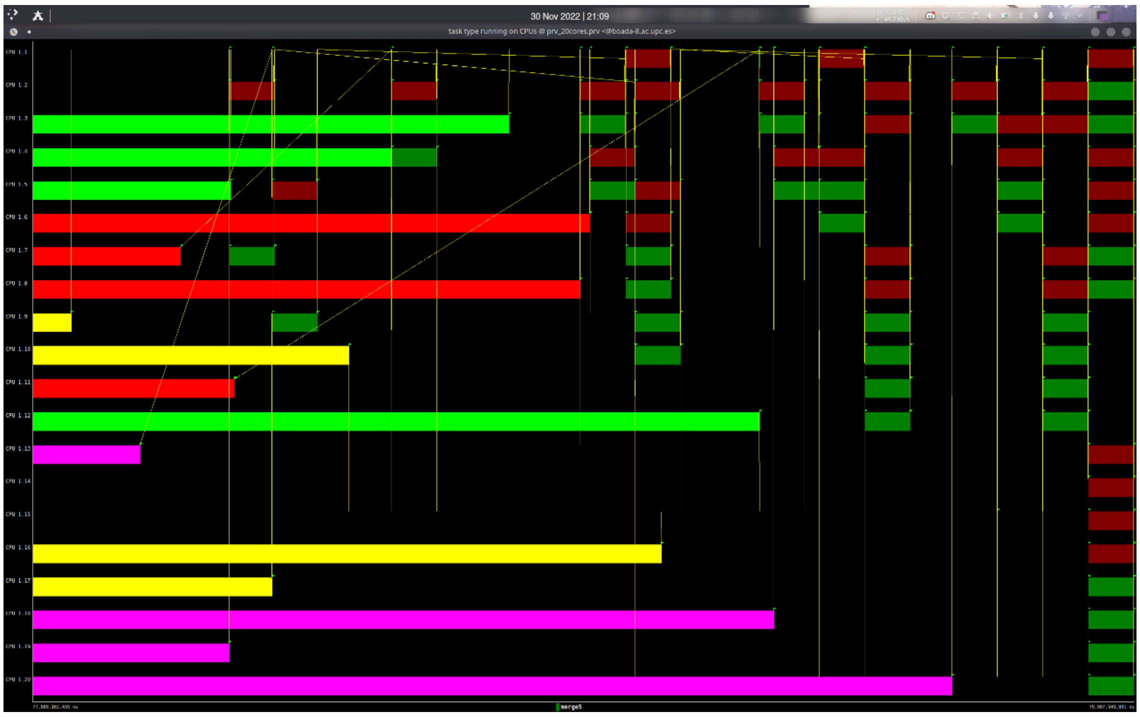
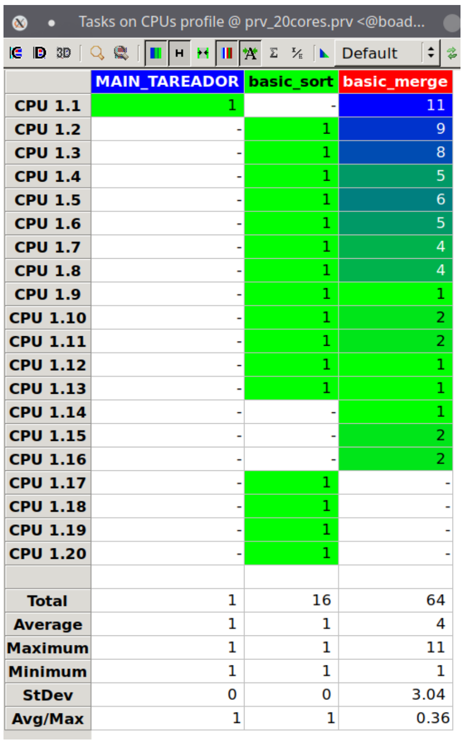


Figure 7. Paraver trace for tareador 20 cores leaf strategy simulation


30 Nov 2022 | 21:09
task type running on CPUs @ prv_20cores.prv <@boada-8.ac.upc.es>
merge5
Figure 8. Paraver trace for tareador 20 cores tree strategy simulation

The main difference between the two implementations is the number of dependencies that a single execution block can have, in the tree strategy a single execution block will have at maximum 2 in dependencies and 2 out dependencies (as we can see in figure 5), this happens because in the code the merge function is the only one that creates dependencies for its intrinsic nature. On the other hand in the leaf code each basicmerge can have a higher number of dependencies as there isn't this merge nature (as we can see in figure 4).

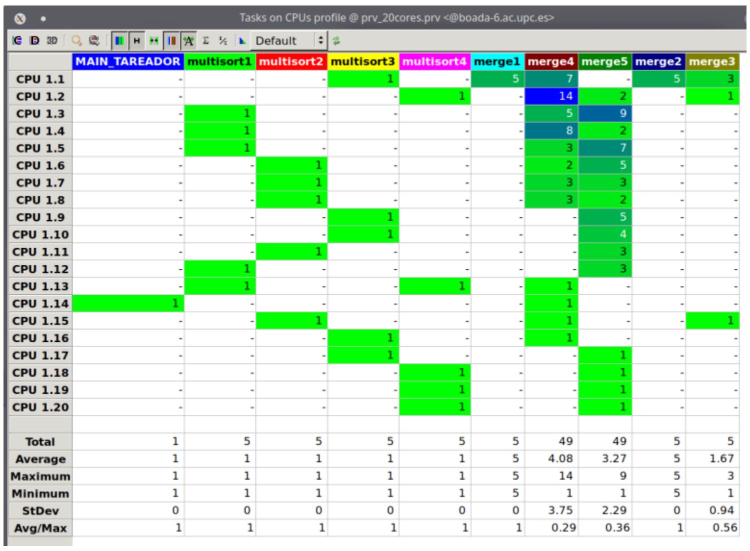
We can corroborate this analyzing the figure 7 and 8, where we can see the yellow lines which represent the dependencies between execution blocs.


Tasks on CPUs profile @ prv_20cores.prv <@boad...

Default

	MAIN_TAREADOR	basic_sort	basic_merge
CPU 1.1	1	-	11
CPU 1.2	-	1	9
CPU 1.3	-	1	8
CPU 1.4	-	1	5
CPU 1.5	-	1	6
CPU 1.6	-	1	5
CPU 1.7	-	1	4
CPU 1.8	-	1	4
CPU 1.9	-	1	1
CPU 1.10	-	1	2
CPU 1.11	-	1	2
CPU 1.12	-	1	1
CPU 1.13	-	1	1
CPU 1.14	-	-	1
CPU 1.15	-	-	2
CPU 1.16	-	-	2
CPU 1.17	-	1	-
CPU 1.18	-	1	-
CPU 1.19	-	1	-
CPU 1.20	-	1	-
Total	1	16	64
Average	1	1	4
Maximum	1	1	11
Minimum	1	1	1
StDev	0	0	3.04
Avg/Max	1	1	0.36

Figure 9. Paraver histogram trace for tareador 20 cores leaf strategy simulation


Tasks on CPUs profile @ prv_20cores.prv <@boada-6.ac.upc.es>

Default

	MAIN_TAREADOR	multisort1	multisort2	multisort3	multisort4	merge1	merge4	merge5	merge2	merge3
CPU 1.1	-	-	-	1	-	5	7	-	5	3
CPU 1.2	-	-	-	-	1	-	14	2	-	1
CPU 1.3	-	1	-	-	-	-	5	9	-	-
CPU 1.4	-	1	-	-	-	-	8	2	-	-
CPU 1.5	-	1	-	-	-	-	3	7	-	-
CPU 1.6	-	-	1	-	-	-	2	5	-	-
CPU 1.7	-	-	1	-	-	-	3	3	-	-
CPU 1.8	-	-	1	-	-	-	3	2	-	-
CPU 1.9	-	-	-	1	-	-	-	5	-	-
CPU 1.10	-	-	-	1	-	-	-	4	-	-
CPU 1.11	-	-	1	-	-	-	-	3	-	-
CPU 1.12	-	1	-	-	-	-	-	3	-	-
CPU 1.13	-	1	-	-	1	-	1	-	-	-
CPU 1.14	1	-	-	-	-	-	1	-	-	-
CPU 1.15	-	-	1	-	-	-	1	-	-	1
CPU 1.16	-	-	-	1	-	-	1	-	-	-
CPU 1.17	-	-	-	1	-	-	-	1	-	-
CPU 1.18	-	-	-	-	1	-	-	1	-	-
CPU 1.19	-	-	-	-	1	-	-	1	-	-
CPU 1.20	-	-	-	-	1	-	-	1	-	-
Total	1	5	5	5	5	5	49	49	5	5
Average	1	1	1	1	1	5	4.08	3.27	5	1.67
Maximum	1	1	1	1	1	5	14	9	5	3
Minimum	1	1	1	1	1	5	1	1	5	1
StDev	0	0	0	0	0	0	3.75	2.29	0	0.94
Avg/Max	1	1	1	1	1	1	0.29	0.36	1	0.56

Figure 10. Paraver histogram trace for tareador 20 cores tree strategy simulation

Analyzing the histograms, we can clearly se the difference in granularity between the leaf (figure 9) and the tree (figure 10), being the tree one finer than the leaf one.

We can also confirm that the merge and basic merge are the parts that use/produce more tasks whilst multisort and basic sort have a lot less tasks but way more intense, this can be deduced by the nature of the sorting "sub algorithm" vs the nature of the merging "sub algorithm".

2. Shared-memory parallelization with OpenMP tasks

2.1 Leaf Strategy with OpenMP

2.1.1 Scalability plots

The speedup is far from the ideal, in fact is really bad. However, the execution time 1.03 is a great improvement. We can see improvement up to 8-9 threads.

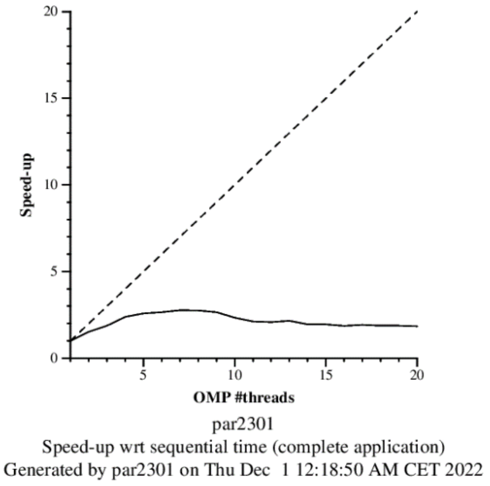


Figure 11. Speedup complete application Leaf Strategy

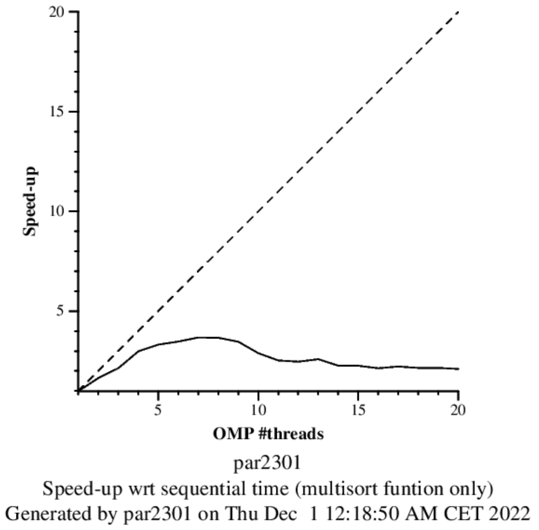
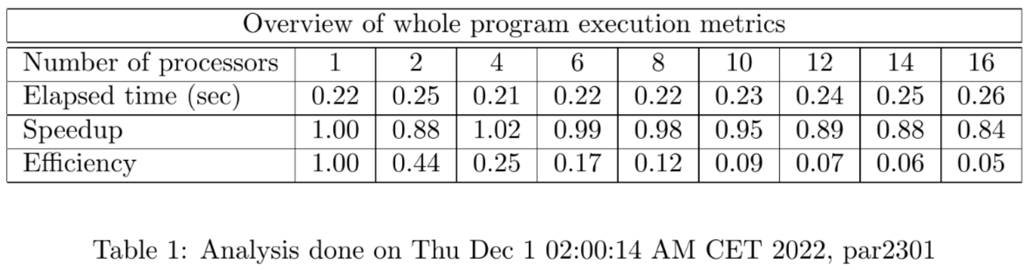


Figure 12. Speedup multisort function Leaf Strategy

2.1.2 Analysis with modelfactors

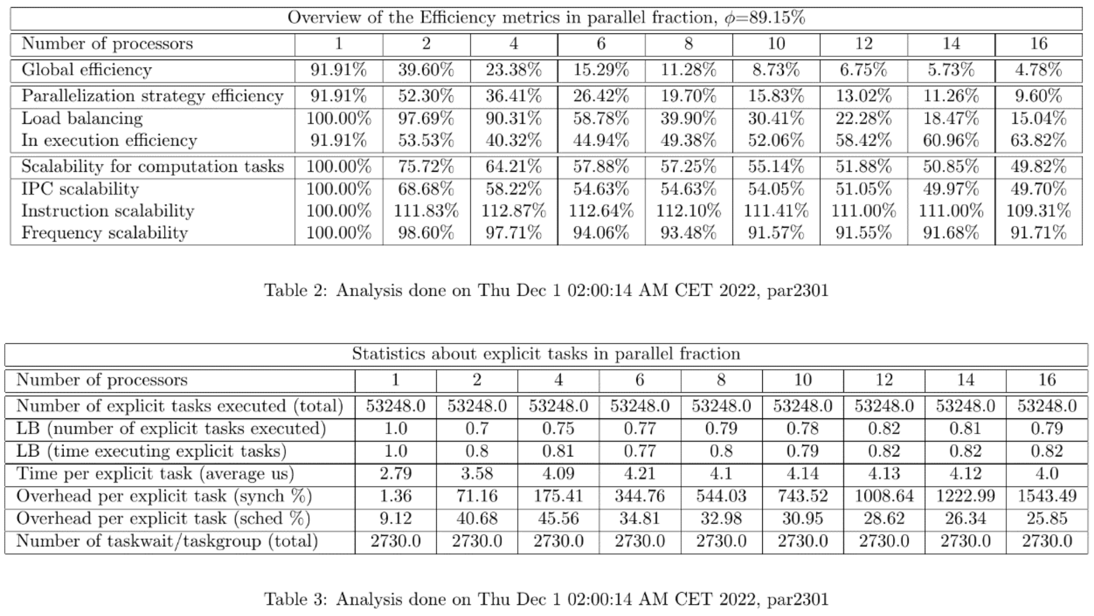
Despite the parallel fraction begin not the best one (at 89.15%) as we can see in figure 14 the main ballast of this code is the parallelization efficiency (at 9.6%) specifically the load balancing (at 15.04%) as we can see in figure 14. This has sense because the code is quite coarse (compared with the tree), so it requires a huge investment in that specific part.



Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.22	0.25	0.21	0.22	0.22	0.23	0.24	0.25	0.26
Speedup	1.00	0.88	1.02	0.99	0.98	0.95	0.89	0.88	0.84
Efficiency	1.00	0.44	0.25	0.17	0.12	0.09	0.07	0.06	0.05

Table 1: Analysis done on Thu Dec 1 02:00:14 AM CET 2022, par2301

Figure 13. Leaf Strategy Modelfactors table 1



Overview of the Efficiency metrics in parallel fraction, ϕ=89.15%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	91.91%	39.60%	23.38%	15.29%	11.28%	8.73%	6.75%	5.73%	4.78%
Parallelization strategy efficiency	91.91%	52.30%	36.41%	26.42%	19.70%	15.83%	13.02%	11.26%	9.60%
Load balancing	100.00%	97.69%	90.31%	58.78%	39.90%	30.41%	22.28%	18.47%	15.04%
In execution efficiency	91.91%	53.53%	40.32%	44.94%	49.38%	52.06%	58.42%	60.96%	63.82%
Scalability for computation tasks	100.00%	75.72%	64.21%	57.88%	57.25%	55.14%	51.88%	50.85%	49.82%
IPC scalability	100.00%	68.68%	58.22%	54.63%	54.63%	54.05%	51.05%	49.97%	49.70%
Instruction scalability	100.00%	111.83%	112.87%	112.64%	112.10%	111.41%	111.00%	111.00%	109.31%
Frequency scalability	100.00%	98.60%	97.71%	94.06%	93.48%	91.57%	91.55%	91.68%	91.71%

Table 2: Analysis done on Thu Dec 1 02:00:14 AM CET 2022, par2301

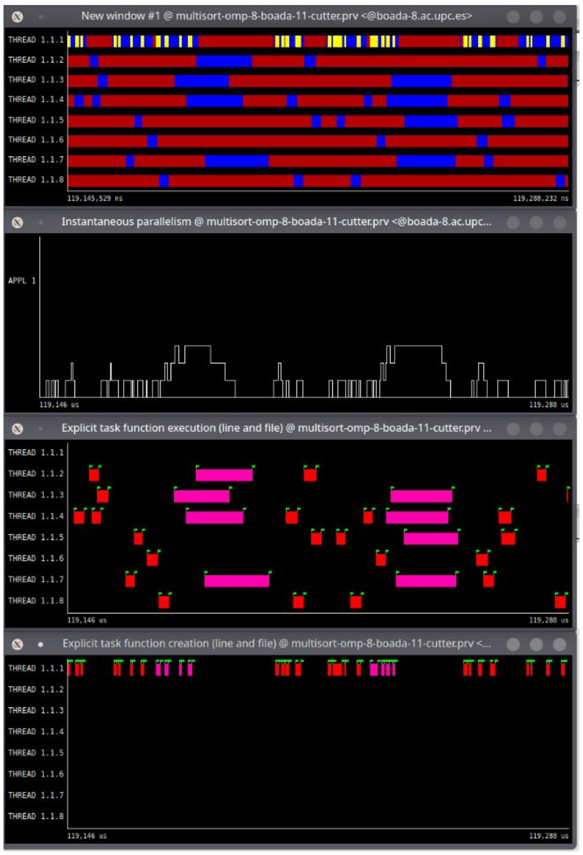
Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.7	0.75	0.77	0.79	0.78	0.82	0.81	0.79
LB (time executing explicit tasks)	1.0	0.8	0.81	0.77	0.8	0.79	0.82	0.82	0.82
Time per explicit task (average us)	2.79	3.58	4.09	4.21	4.1	4.14	4.13	4.12	4.0
Overhead per explicit task (synch %)	1.36	71.16	175.41	344.76	544.03	743.52	1008.64	1222.99	1543.49
Overhead per explicit task (sched %)	9.12	40.68	45.56	34.81	32.98	30.95	28.62	26.34	25.85
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 Thu Dec 1 02:00:14 AM CET 2022, par2301

Figure 14. Leaf Strategy Modelfactors table 2 and 3

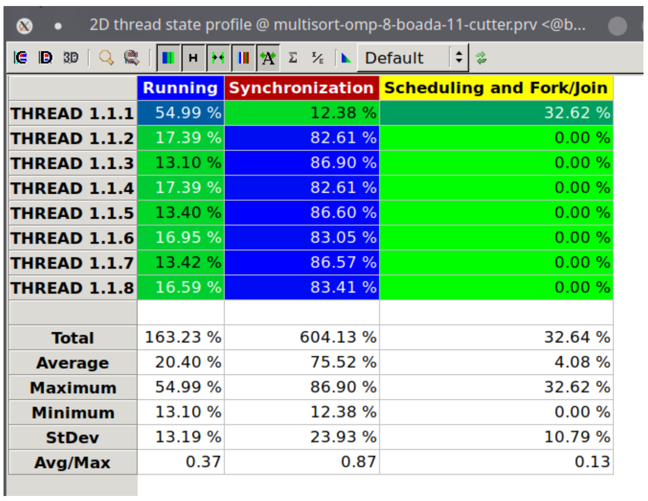
2.1.3 Analysis with Paraver

We can easily see the may problem for this program not begin good enough, and that's what we can see in figure 15 (red), were all the threads except the first one are most of the time syncing and not executing profitable code. So, we deduce that the granularity is too fine and that there are too many tasks too small to compensate for the overheads. As maximum we have up to 4 threads executing code simultaneously.



New window #1 @ multisort-omp-8-boada-11-cutter.prv <@boada-8.ac.upc.es>
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
119,145,529 ns
119,288,232 ns
Instantaneous parallelism @ multisort-omp-8-boada-11-cutter.prv <@boada-8.ac.upc...
APPL 1
119,146 us
119,288 us
Explicit task function execution (line and file) @ multisort-omp-8-boada-11-cutter.prv ...
Explicit task function creation (line and file) @ multisort-omp-8-boada-11-cutter.prv <...

Figure 15. Leaf Strategy Paraver traces



2D thread state profile @ multisort-omp-8-boada-11-cutter.prv <@b...

Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	54.99 %	12.38 %	32.62 %
THREAD 1.1.2	17.39 %	82.61 %	0.00 %
THREAD 1.1.3	13.10 %	86.90 %	0.00 %
THREAD 1.1.4	17.39 %	82.61 %	0.00 %
THREAD 1.1.5	13.40 %	86.60 %	0.00 %
THREAD 1.1.6	16.95 %	83.05 %	0.00 %
THREAD 1.1.7	13.42 %	86.57 %	0.00 %
THREAD 1.1.8	16.59 %	83.41 %	0.00 %
Total	163.23 %	604.13 %	32.64 %
Average	20.40 %	75.52 %	4.08 %
Maximum	54.99 %	86.90 %	32.62 %
Minimum	13.10 %	12.38 %	0.00 %
StDev	13.19 %	23.93 %	10.79 %
Avg/Max	0.37	0.87	0.13

Figure 16. Leaf Strategy Thread State

2.2 Tree strategy with OpenMP

2.2.1 Scalability plots

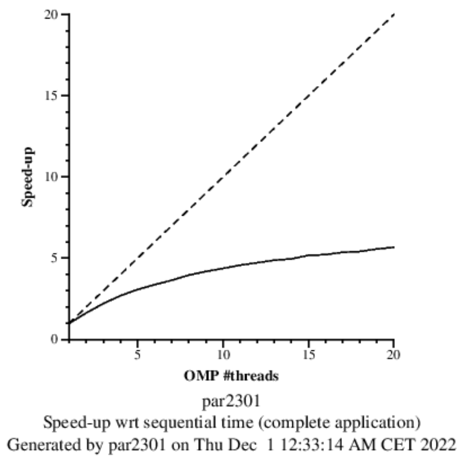


Figure 17. Speedup complete application Tree Strategy

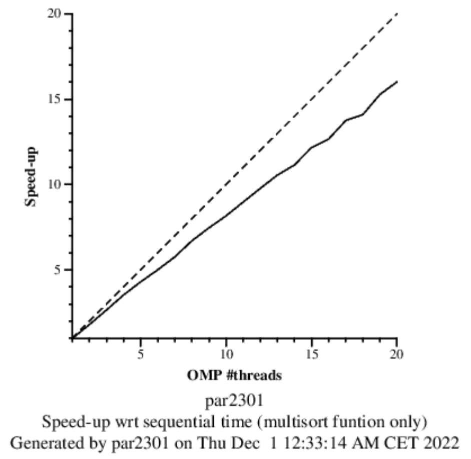
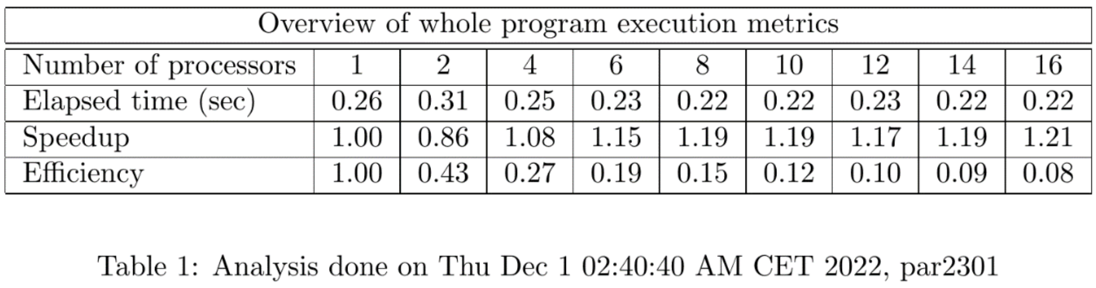


Figure 18. Speedup multisort function Tree Strategy

Now we can see that the multisort function (figure 18) is much more steep that in the leaf version and now is quite close to the ideal parallelism. Despite that the whole application (figure 17) still is far from that ideal parallelism, so we can deduce that there must be some other part of the application that should be parallized to improve this mark.

2.2.2 Analysis with modelfactors

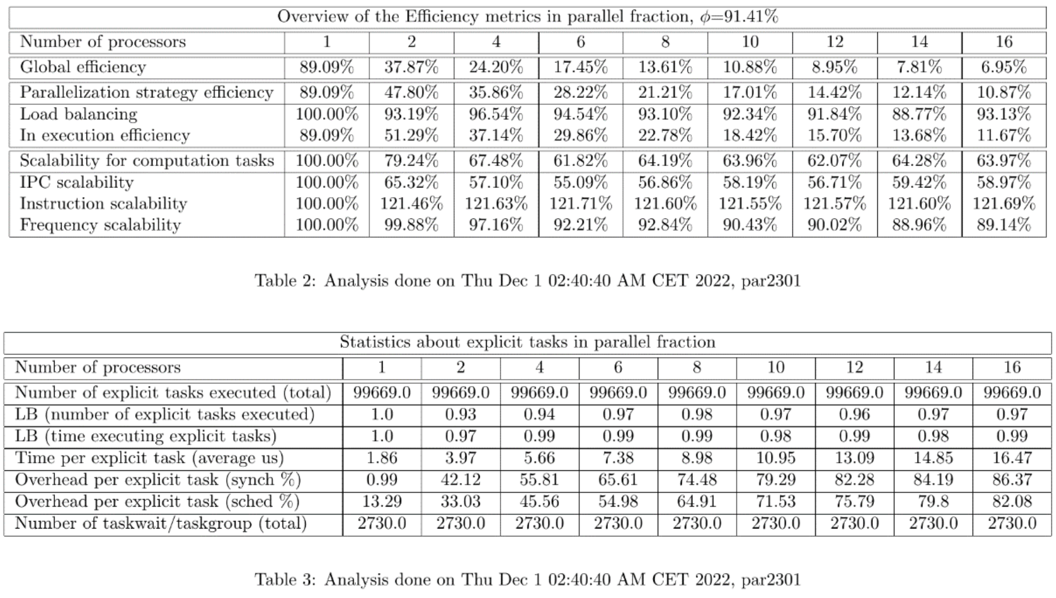
In this case we can see that the Load Balancing is not a problem (93.13% efficiency at 16 cores see figure 20). However, now the problem that prevents us from achievieng the best paralelitzation of the code is the In Execution Efficiency (11.67% efficiency at 16 cores see figure 20). We can confirm with this our deduction, that we have to pararelizice more % of the code in order to improve this mark.



Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.26	0.31	0.25	0.23	0.22	0.22	0.23	0.22	0.22
Speedup	1.00	0.86	1.08	1.15	1.19	1.19	1.17	1.19	1.21
Efficiency	1.00	0.43	0.27	0.19	0.15	0.12	0.10	0.09	0.08

Table 1: Analysis done on Thu Dec 1 02:40:40 AM CET 2022, par2301

Figure 19. Tree Strategy Modelfactors table 1



Overview of the Efficiency metrics in parallel fraction, ϕ=91.41%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	89.09%	37.87%	24.20%	17.45%	13.61%	10.88%	8.95%	7.81%	6.95%
Parallelization strategy efficiency	89.09%	47.80%	35.86%	28.22%	21.21%	17.01%	14.42%	12.14%	10.87%
Load balancing	100.00%	93.19%	96.54%	94.54%	93.10%	92.34%	91.84%	88.77%	93.13%
In execution efficiency	89.09%	51.29%	37.14%	29.86%	22.78%	18.42%	15.70%	13.68%	11.67%
Scalability for computation tasks	100.00%	79.24%	67.48%	61.82%	64.19%	63.96%	62.07%	64.28%	63.97%
IPC scalability	100.00%	65.32%	57.10%	55.09%	56.86%	58.19%	56.71%	59.42%	58.97%
Instruction scalability	100.00%	121.46%	121.63%	121.71%	121.60%	121.55%	121.57%	121.60%	121.69%
Frequency scalability	100.00%	99.88%	97.16%	92.21%	92.84%	90.43%	90.02%	88.96%	89.14%

Table 2: Analysis done on Thu Dec 1 02:40:40 AM CET 2022, par2301

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	0.93	0.94	0.97	0.98	0.97	0.96	0.97	0.97
LB (time executing explicit tasks)	1.0	0.97	0.99	0.99	0.99	0.98	0.99	0.98	0.99
Time per explicit task (average us)	1.86	3.97	5.66	7.38	8.98	10.95	13.09	14.85	16.47
Overhead per explicit task (synch %)	0.99	42.12	55.81	65.61	74.48	79.29	82.28	84.19	86.37
Overhead per explicit task (sched %)	13.29	33.03	45.56	54.98	64.91	71.53	75.79	79.8	82.08
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 Thu Dec 1 02:40:40 AM CET 2022, par2301

Figure 20. Tree Strategy Modelfactors table 2 and 3

2.2.3 Analysis with Paraver

Now we can see in the first trace of the figure 21 that the predomining type of task is execution tasks not syncronizing as in the leaf strategy. That's the first indicator of the performance improvement. In the second trace we can see that the thread usage is quite higher with spikes of almost all the threads. And finally from the last 2 traces the difference with the ones in the leaf one is really significative, now almost all the time all the threads are beign useful to the code.

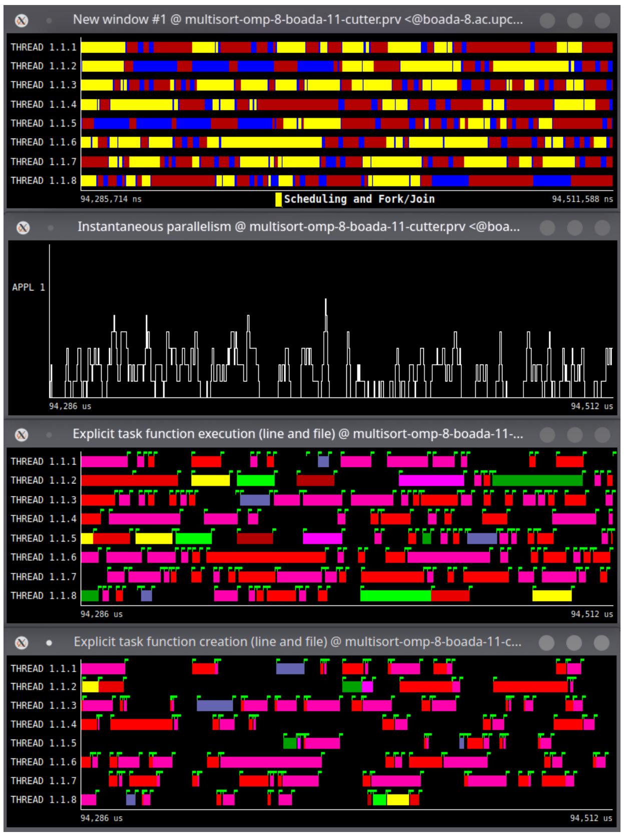

New window #1 @ multisort-omp-8-boada-11-cutter.prv <@boada-8.ac.upc...
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
94,285,714 ns
Scheduling and Fork/Join
94,511,588 ns
Instantaneous parallelism @ multisort-omp-8-boada-11-cutter.prv <@boa...
APPL 1
94,286 us
94,512 us
Explicit task function execution (line and file) @ multisort-omp-8-boada-11-...
Explicit task function creation (line and file) @ multisort-omp-8-boada-11-c...

Figure 21. Tree Strategy Paraver Traces

2.3 Controlling task granularities: cut-off mechanism

After analyzing the elapsed time plot (figure 22), we have determined that the best cut-off was 6. We can se the corresponding modelfactors tables in figure 23 and 24.

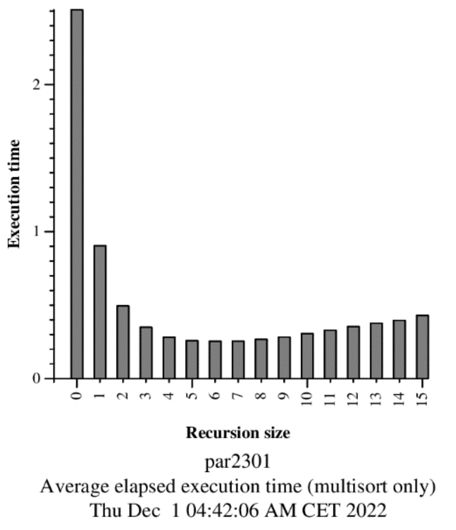


Figure 22. Tree Strategy + cut-off

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	0.25	0.23	0.22	0.23	0.22	0.22	0.22
Speedup	1.00	0.86	1.04	1.12	1.19	1.16	1.18	1.20	1.21
Efficiency	1.00	0.43	0.26	0.19	0.15	0.12	0.10	0.09	0.08

Table 1: Analysis done on Thu Dec 1 04:56:33 AM CET 2022, par2301

Figure 23. Tree Strategy + cut-off (6) Modelfactors table 1

Overview of the Efficiency metrics in parallel fraction, ϕ=91.36%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	88.98%	37.91%	23.26%	17.00%	13.59%	10.56%	9.01%	7.83%	6.90%
Parallelization strategy efficiency	88.98%	47.73%	34.29%	27.64%	21.08%	17.06%	14.48%	12.41%	10.83%
Load balancing	100.00%	93.78%	97.06%	96.00%	95.08%	92.96%	92.92%	91.79%	92.58%
In execution efficiency	88.98%	50.90%	35.33%	28.79%	22.18%	18.35%	15.58%	13.52%	11.69%
Scalability for computation tasks	100.00%	79.42%	67.84%	61.49%	64.44%	61.90%	62.24%	63.07%	63.74%
IPC scalability	100.00%	65.92%	57.13%	55.03%	57.03%	56.37%	57.49%	58.48%	59.01%
Instruction scalability	100.00%	121.45%	121.42%	121.41%	121.61%	121.47%	121.42%	121.27%	121.53%
Frequency scalability	100.00%	99.20%	97.79%	92.04%	92.90%	90.39%	89.16%	88.93%	88.88%

Table 2: Analysis done on Thu Dec 1 04:56:33 AM CET 2022, par2301

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	0.94	0.94	0.95	0.98	0.98	0.98	0.98	0.98
LB (time executing explicit tasks)	1.0	0.97	0.99	0.99	0.99	0.99	0.98	0.99	0.98
Time per explicit task (average us)	1.84	3.94	5.75	7.48	8.89	11.17	12.91	14.67	16.56
Overhead per explicit task (synch %)	1.01	41.57	57.8	65.98	74.81	78.98	81.74	83.97	85.04
Overhead per explicit task (sched %)	13.43	33.45	47.35	55.9	65.12	71.6	75.68	79.39	82.11
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 Thu Dec 1 04:56:33 AM CET 2022, par2301

Figure 24. Tree Strategy + cut-off (6) Modelfactors table 2

2.4 Optional

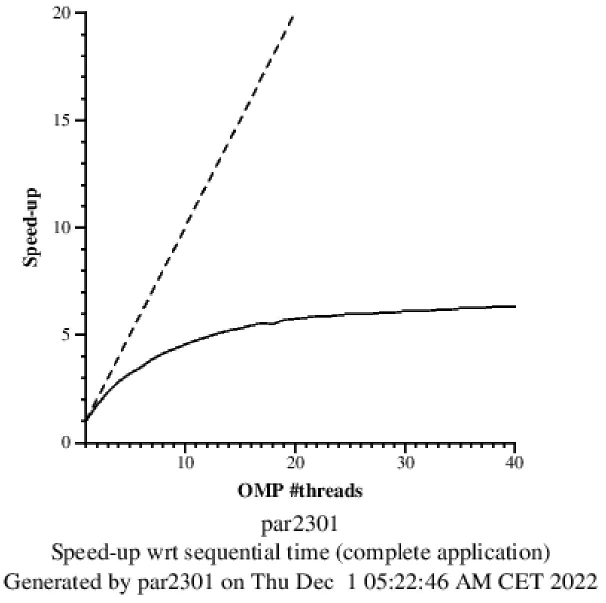
Executing with 40 threads we have got the following values, which are better than with 20 cores. We think that this happens because the intel xeon's processor's in Boada have what's called hyper threading, which allows to have a substantial performance improvement over using traditional physical cores.

Initialization time in seconds: 0.684816

Multisort execution time: 0.213462

Check sorted data execution time: 0.012919

We continue seeing the same pattern as the previus analysis, the aplication is far from ideal whilst the multisort function continues scaling. However, the scaling rate with phyisical cores vs when hyperthreading is active is much higher, after passing the 20 physicial core mark the increasing rate drops significantly as we can see in figure 25.



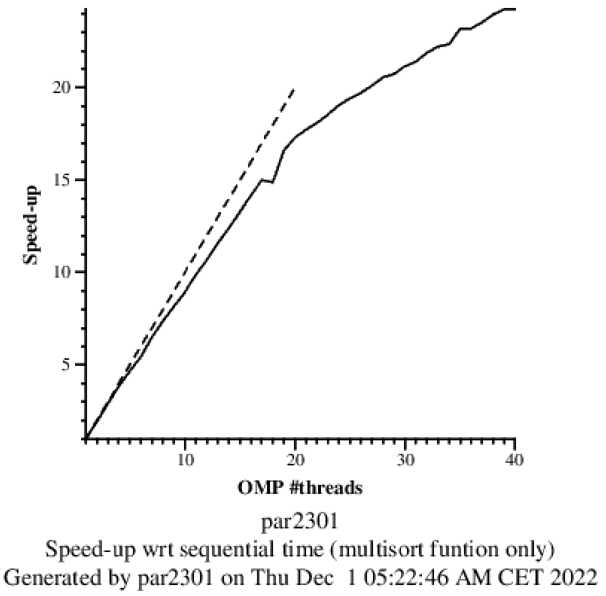


Figure 25. Tree Strategy + cut-off (6) 40 thread speedups

3. Shared-memory parallelization using dependencies

3.1 Parallelization and performance analysis

3.1.1 Scalability plots

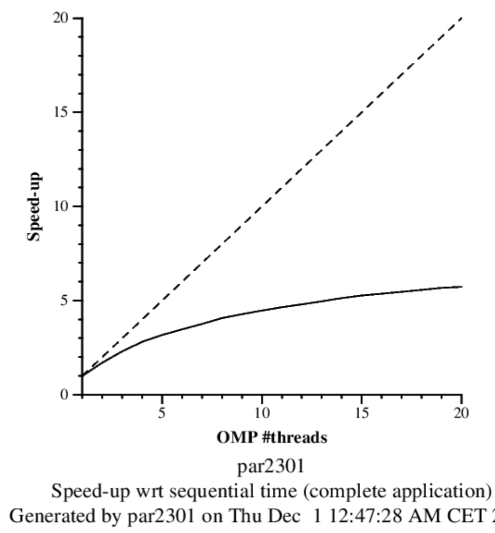


Figure 26. Speedup complete application with dependencies

The scalability plots for this new version are shown in figure 26 and figure 27. As we can see in figure 26, the speedup of the program itself barely improves when increasing the number of threads. This means that excluding multisort the code can be highly parallelized, for example we could parallelize function initialize(). Unlike multisort function which is almost ideally parallelized.

This new version isn't simpler to code than the previous one, since you need to check what the functions do in order to know if you need an *in*, an *out* or an *inout* dependency. It should be more efficient though, since you don't block a set of tasks as a group, only the ones that depend on another one, and that one hasn't finished its execution yet. The scalability plot doesn't show so, which is because this synchronization provokes overheads, "cancelling" the aforementioned small optimization.

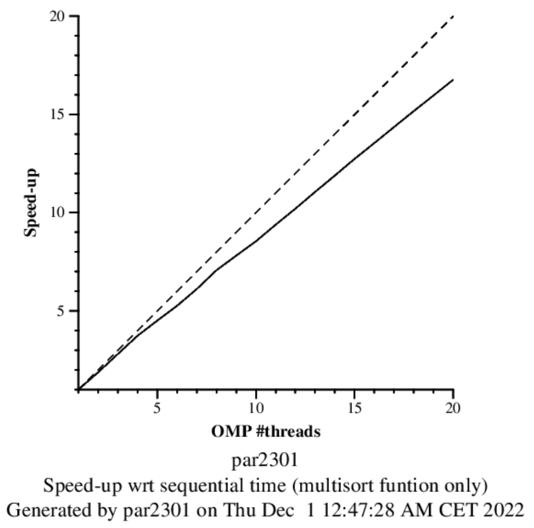
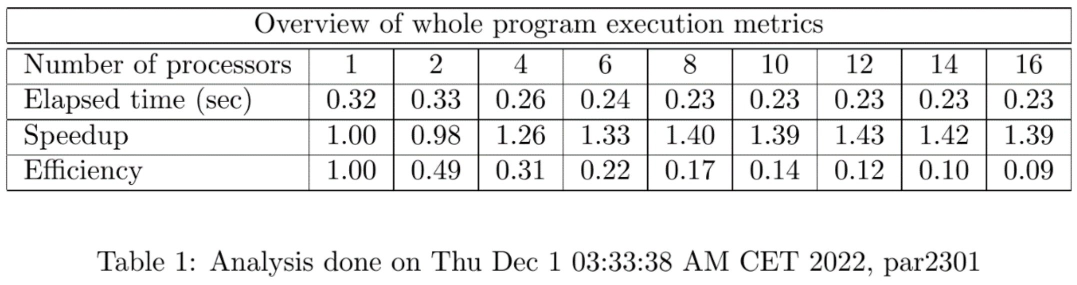


Figure 27. Speedup multisort function with dependencies

3.1.2 Analysis with modelfactors

In figure 28 and 29 we can see the modelfactor tables for the version multisort-omp-tree-cutoff-dependency.c file. Its results prove what we mentioned before: the speedup of the whole program remains quite stable, the program can be more parallelized.

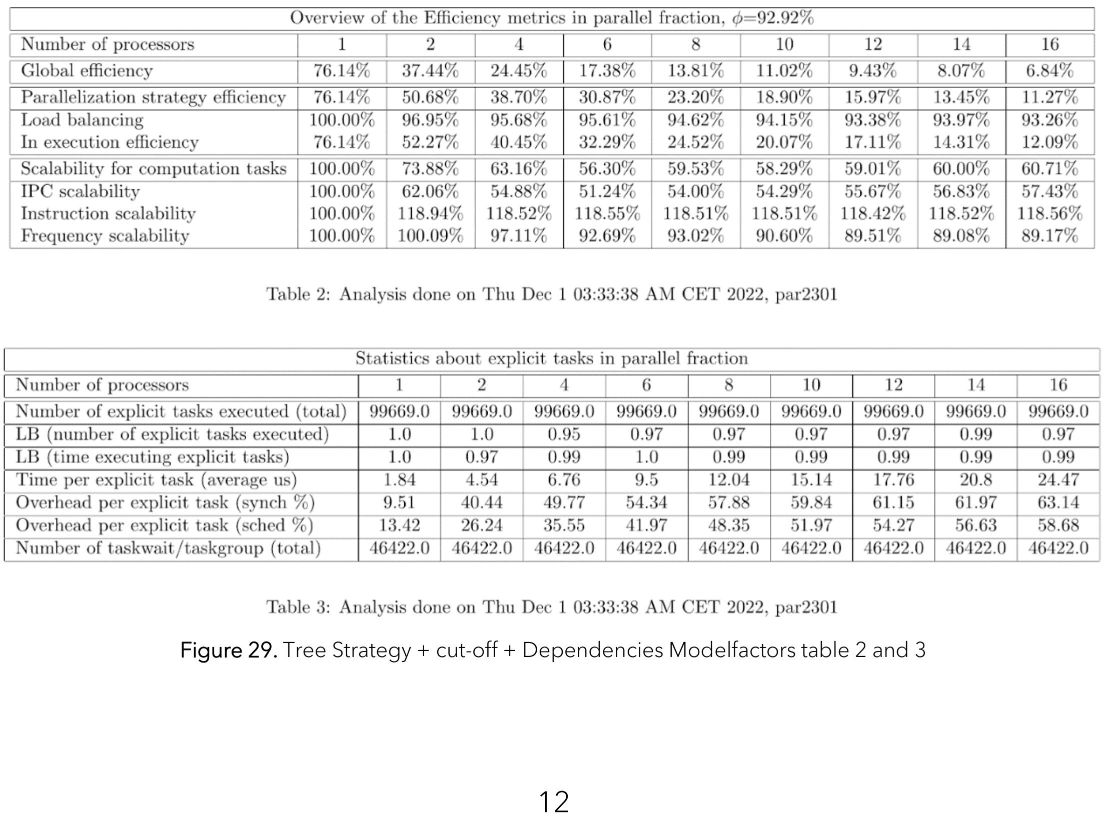


Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.32	0.33	0.26	0.24	0.23	0.23	0.23	0.23	0.23
Speedup	1.00	0.98	1.26	1.33	1.40	1.39	1.43	1.42	1.39
Efficiency	1.00	0.49	0.31	0.22	0.17	0.14	0.12	0.10	0.09

Table 1: Analysis done on Thu Dec 1 03:33:38 AM CET 2022, par2301

Figure 28. Tree Strategy + cut-off + Dependencies Modelfactors table 1

It also shows that creating too many tasks may cause overheads, contrarresting the parallelization from multisort function.



Overview of the Efficiency metrics in parallel fraction, ϕ=92.92%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	76.14%	37.44%	24.45%	17.38%	13.81%	11.02%	9.43%	8.07%	6.84%
Parallelization strategy efficiency	76.14%	50.68%	38.70%	30.87%	23.20%	18.90%	15.97%	13.45%	11.27%
Load balancing	100.00%	96.95%	95.68%	95.61%	94.62%	94.15%	93.38%	93.97%	93.26%
In execution efficiency	76.14%	52.27%	40.45%	32.29%	24.52%	20.07%	17.11%	14.31%	12.09%
Scalability for computation tasks	100.00%	73.88%	63.16%	56.30%	59.53%	58.29%	59.01%	60.00%	60.71%
IPC scalability	100.00%	62.06%	54.88%	51.24%	54.00%	54.29%	55.67%	56.83%	57.43%
Instruction scalability	100.00%	118.94%	118.52%	118.55%	118.51%	118.51%	118.42%	118.52%	118.56%
Frequency scalability	100.00%	100.09%	97.11%	92.69%	93.02%	90.60%	89.51%	89.08%	89.17%

Table 2: Analysis done on Thu Dec 1 03:33:38 AM CET 2022, par2301

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	1.0	0.95	0.97	0.97	0.97	0.97	0.99	0.97
LB (time executing explicit tasks)	1.0	0.97	0.99	1.0	0.99	0.99	0.99	0.99	0.99
Time per explicit task (average us)	1.84	4.54	6.76	9.5	12.04	15.14	17.76	20.8	24.47
Overhead per explicit task (synch %)	9.51	40.44	49.77	54.34	57.88	59.84	61.15	61.97	63.14
Overhead per explicit task (sched %)	13.42	26.24	35.55	41.97	48.35	51.97	54.27	56.63	58.68
Number of taskwait/taskgroup (total)	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0

Table 3: Analysis done on Thu Dec 1 03:33:38 AM CET 2022, par2301

Figure 29. Tree Strategy + cut-off + Dependencies Modelfactors table 2 and 3

3.1.3 Analysis with Paraver

The following figure corresponds to the paraver traces of our version of the code using the depend clause. We can see from the first trace that the overhead and the execution of the program is pretty balanced, which is the reason why its scalability is not ideal.

From the second one we see that sometimes there are no threads being executed, which is a waste of time. But it could also be because of the dependencies generated between the functions as shown in the last two traces.

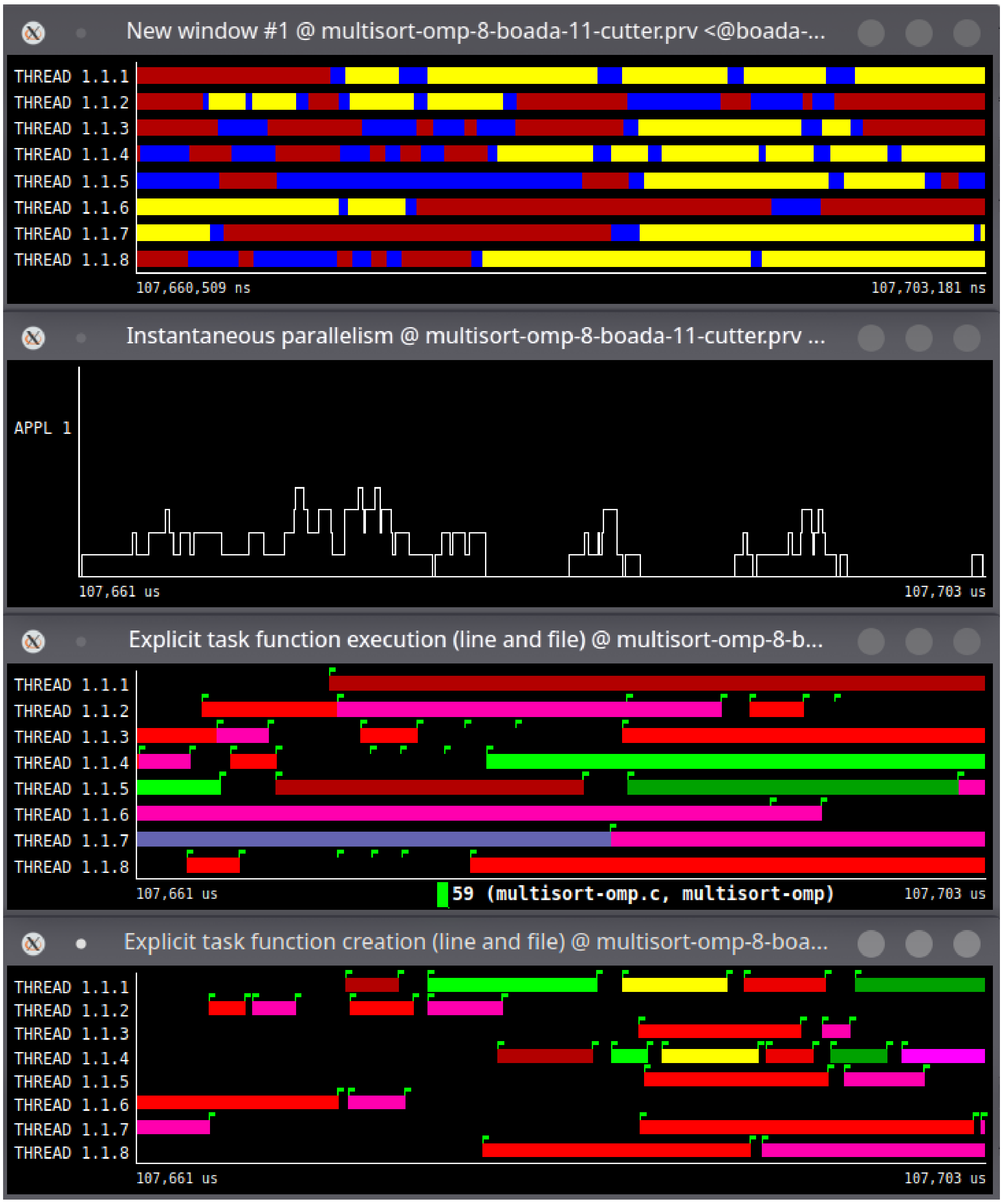


Figure 30. Tree Strategy + cut-off + Dependencies paraver traces

3.2 Overall Analysis and Conclusions

After reviewing all the versions of the algorithm we've concluded that the most efficient one is the tree + cut-off one. And regarding programmability, we've found the simpler to code the original ones, the leaf and the tree, then the cut-off versions and finally the dependence ones, those ones in specific we found them quite challenging as we needed to know what where each variable and vector doing in every part of the algorithm in order to assign the dependences.

Although it's true that the dependence version comes with a little overhead we don't think is enough to discard the high semantic complexity that this mechanism allows us to code.

The mentioned overhead can be seen between figure 23 and figure 28, in which we see that in average the version without the dependencies tends to have around 0.01s more than one with them. This can also be corroborated in the efficiency tables in figure 24 and 29 which also have the same little difference.

3.3 Optional