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

Walter J. Troiani Vargas (2318) and Marc Tacons Vega(2316)

May 15, 2023

Index

1 Introduction

The aim of this session is to explore the use of parallelism in divide-and-conquer algorithms and recursive programs. We will explore 2 different approaches in recursive task decomposition: Leaf strategy and Tree strategy.

The algorithm selected for this analysis will be a slightly different version of the famous MergeSort. This procedure consist in dividing the given list or array into two halves, continue splitting them recursively and then when the size of the array is small enough (a trivial size like 2 elements). Another procedure called merge, joins both sorted halves and combine them into a new bigger sorted one.

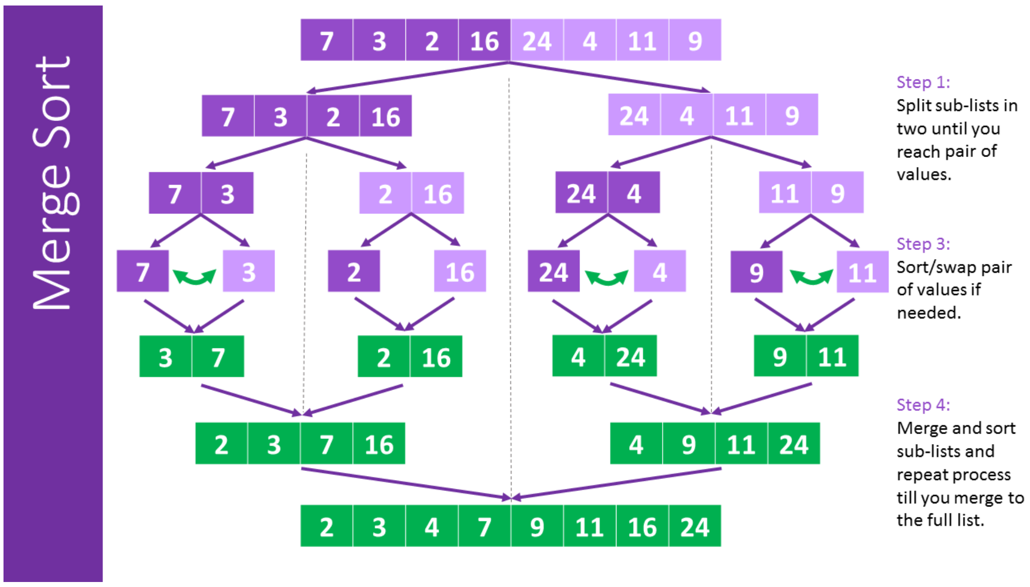

Merge Sort
7 3 2 16 24 4 11 9
7 3 2 16
24 4 11 9
7 3
2 16
24 4
11 9
7 3
2 16
24 4
9 11
3 7
2 16
4 24
9 11
2 3 7 16
4 9 11 24
2 3 4 7 9 11 16 24
Step 1:
Split sub-lists in two until you reach pair of values.
Step 3:
Sort/swap pair of values if needed.
Step 4:
Merge and sort sub-lists and repeat process till you merge to the full list.

Figure 1: Merge sort Procedure scheme

Our algorithm multisort uses this idea but in a different fashion: The array will be split into 4 slices which will also be split into 4 slices each one and so on. This recursive process will stop when it reaches a certain minimum size, which is parameterized. When that size is reached, then the original merge sort will be used to sort those slices and after being sorted we will merge them in groups of for using the merge procedure 2 by 2 (Then 3 merges are needed to merge 4 arrays into new sorted bigger one).

2 Task decomposition analysis with Tareador

Once we are familiar with the *multisort-seq.c* code we can start analysing the potential task decomposition with the Tareador program. To do this we are going to modify the *multisort-tareador.c* file to implement the different strategies.

In this laboratory we were told to explore two distinct recursive task decomosition strategies: **Leaf** and **Tree**:

- ***Leaf:*** In this strategy we are going to define the task creation in the invocations of the basic cases(*basicsort* and *basicmerge*) of both *multisort* and *merge*.
- ***Tree:*** In the tree strategy the tasks creation are going to be defined during the recursive calls of *multisort* and *merge*.

2.1 Leaf strategy

After adding the proper Tareador clauses to create the tasks for the leaf strategy, we compile it, and we execute it with the *run-tareador.sh* script, which generates the following task dependence graph(see Figure 2).

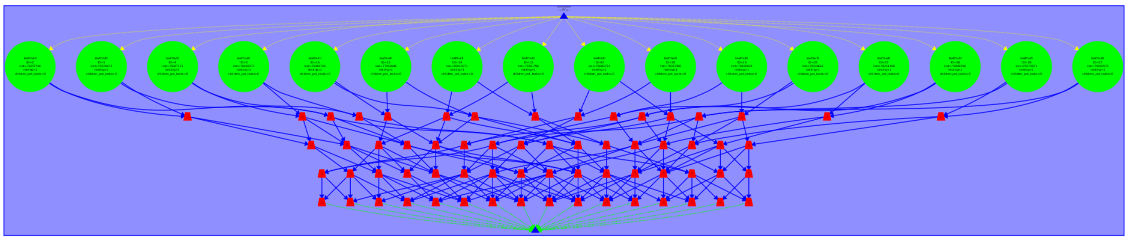


Figure 2: Task Dependency Graph for the Leaf Strategy

If we take a closer look into the graph, we can see that the tasks created by the **basicsort** base case are drawn in green and all of them are fully parallelized. The rest of red tasks correspond to the **basicmerge** base case of the merge function. In this case, we can clearly see that some dependencies are generated because the basicmerge function requires both sides of the array to be sorted. Therefore, to synchronize all tasks, we will need to put some *#pragma omp taskwait* clauses before the **merge** function invocations to guarantee these dependencies. In the incoming section will be discussed more

2.2 Tree strategy

Now we can explore the other recursive task decomposition, the tree strategy. To analyze it with Tareador we need to add at every recursive call from merge and multisort the tareador start task and end task clauses. After compiling and executing with the *run-tareador.sh* script, we obtain the following graph (see Figure 3).

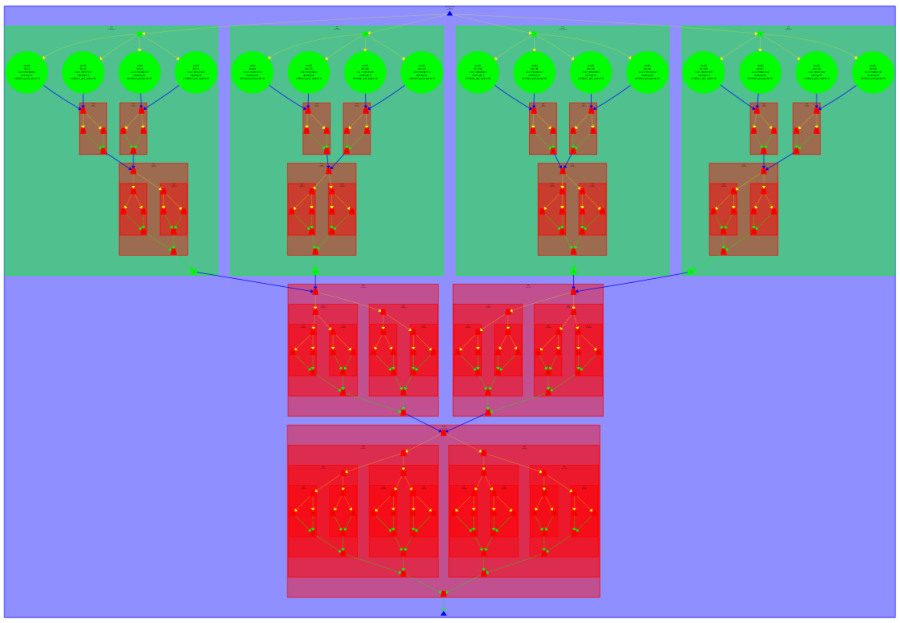


Figure 3: Task Dependency Graph for the Tree Strategy

We clearly can see a very different TDG compared to that of the previous strategy. This is because a task is generated in every recursive call. Looking into the graph we can see that the multisort tasks (see the green circles and the ones that are below each branch in Figure 3) don't create any dependencies due to they don't depend on the vector like merge. On the other hand, merge tasks still create the same dependencies as in the leaf strategy. So we are going to need the *#pragma omp taskwait* or the *#pragma omp taskgroup* clauses to guarantee this dependencies and syncronize all tasks. Moreover, it is important to ramark that the *taskgroup* clause contains an implicit *taskwait* at the end. This way the synchronisation is guaranteed. In the following section will take a closer look in this strategy.

3 Leaf and Tree Strategies Implementations

Now we will proceed to show both strategies and an additional implementation for the Tree, in order to improve its performance:

- Leaf Strategy
- Tree Strategy
- Tree Strategy (Parameterized with a CUT-OFF)

For each one, we will discuss its performance, parallelism and other metrics plotting the scalability, showing tables of all of its statistics and showing the results of its execution on Paraver.

3.1 Leaf Strategy

In this implementation, tasks are created in the leaves of the recursion tree of both the MultiSort and auxiliary MergeSort procedure. The scalability is shown in the following figure.

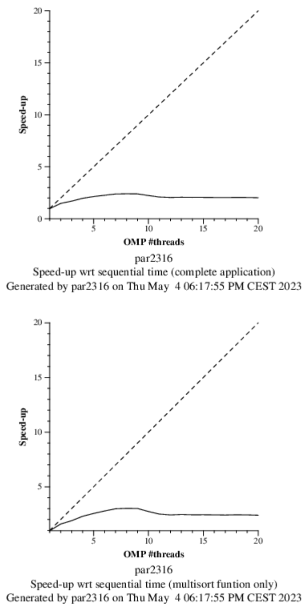


Figure 4: Scalability Plots for the Leaf Strategy

It is crystal clear that there's no strong scalability behaviour on this implementation and the maximum speed-up, before going down , is not that high. That is due to the fact that the leaf strategy depends on one single thread (the master thread) to create all the tasks, thus profiting from parallelization at the end of the program and not in the whole execution. Also the other threads have to wait for tasks aviable on the task pool so there is additional wait time. Looking at the following modelfactor tables and Paraver execution, we can easily identify that the overhead caused by synchronizations is going to be a huge limiting factor for this implementation:

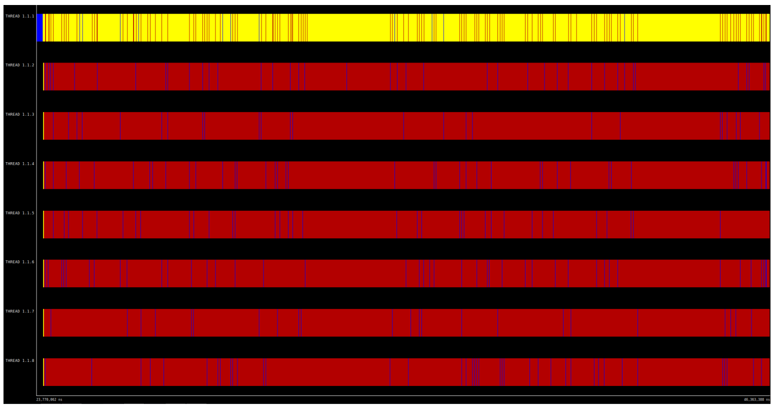


Figure 5: Execution of this implementation on 8 threads

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	0.21	0.21	0.22	0.22	0.24	0.24	0.25
Speedup	1.00	0.87	0.99	1.02	0.98	0.98	0.90	0.90	0.84
Efficiency	1.00	0.43	0.25	0.17	0.12	0.10	0.08	0.06	0.05

Table 1: Analysis done on Fri May 12 05:57:40 PM CEST 2023, par2318

Overview of the Efficiency metrics in parallel fraction, ϕ=89.33%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	91.87%	39.39%	22.76%	15.76%	11.39%	9.09%	6.90%	5.87%	4.78%
Parallelization strategy efficiency	91.87%	53.30%	36.08%	26.61%	20.45%	15.45%	13.32%	11.54%	9.75%
Load balancing	100.00%	99.02%	89.39%	62.07%	38.49%	33.38%	22.63%	18.82%	15.53%
In execution efficiency	91.87%	53.83%	40.36%	42.87%	53.14%	46.28%	58.86%	61.31%	62.76%
Scalability for computation tasks	100.00%	73.91%	63.08%	59.23%	55.67%	58.87%	51.77%	50.89%	49.05%
IPC scalability	100.00%	66.82%	56.24%	55.55%	52.65%	56.39%	50.70%	49.89%	48.01%
Instruction scalability	100.00%	112.39%	113.19%	113.08%	111.64%	112.73%	110.70%	110.98%	110.95%
Frequency scalability	100.00%	98.42%	99.08%	94.28%	94.71%	92.60%	92.24%	91.92%	92.08%

Table 2: Analysis done on Fri May 12 05:57:40 PM CEST 2023, par2318

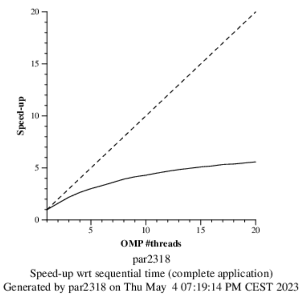
Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.79	0.73	0.78	0.79	0.82	0.79	0.8	0.82
LB (time executing explicit tasks)	1.0	0.82	0.81	0.77	0.78	0.83	0.79	0.79	0.81
Time per explicit task (average us)	2.75	3.66	4.1	4.12	4.05	3.98	4.06	4.08	4.1
Overhead per explicit task (synch %)	1.28	70.54	179.56	336.76	539.0	731.04	990.22	1186.04	1485.29
Overhead per explicit task (sched %)	9.23	35.57	44.63	33.24	26.39	33.28	24.29	22.41	23.64
Number of taskwait/taskgroup (total)	4095.0	4095.0	4095.0	4095.0	4095.0	4095.0	4095.0	4095.0	4095.0

Table 3: Analysis done on Fri May 12 05:57:40 PM CEST 2023, par2318

We can also see that even though the parallel fraction is pretty good , it doesn't make up for the bad load balancing and the huge synchronizations overhead these implementations suffer from. The efficiency drastically drops when more than one thread is used, and becomes unacceptable after reaching 8 threads. It's clear on the Paraver execution that threads spend little time executing tasks.

3.2 Tree Strategy

In this implementation, tasks are created in the whole tree transversal by several threads and not just the master thread.



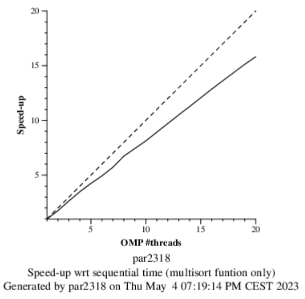


Figure 6: Scalability Plots for the tree implementation

Judging by the plot, we can see that this implementation has far better scalability (grows in a sublinear fashion) and gives good results when the number of processors is increased, because the transversal of the recursion tree is accelerated.

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	0.23	0.22	0.22	0.22
Speedup	1.00	0.85	1.07	1.16	1.18	1.17	1.19	1.20	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 May 4 07:42:23 PM CEST 2023, par2316

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	88.92%	37.51%	23.96%	17.58%	13.41%	10.63%	9.04%	7.81%	6.94%
Parallelization strategy efficiency	88.92%	47.82%	35.88%	26.11%	20.41%	16.52%	13.96%	12.15%	10.52%
Load balancing	100.00%	95.22%	97.23%	93.15%	91.86%	90.10%	92.59%	90.98%	91.49%
In execution efficiency	88.92%	50.22%	36.91%	28.03%	22.22%	18.34%	15.08%	13.35%	11.50%
Scalability for computation tasks	100.00%	78.44%	66.77%	67.34%	65.69%	64.36%	64.72%	64.29%	65.98%
IPC scalability	100.00%	65.24%	55.76%	58.97%	57.68%	58.28%	59.07%	58.85%	60.46%
Instruction scalability	100.00%	121.80%	121.81%	121.75%	121.78%	121.91%	121.86%	121.93%	121.88%
Frequency scalability	100.00%	98.71%	98.30%	93.79%	93.51%	90.59%	89.91%	89.60%	89.53%

Table 2: Analysis done on Thu May 4 07:42:23 PM CEST 2023, par2316

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	0.99	0.95	0.98	0.98	0.98	0.98	0.97	0.99
LB (time executing explicit tasks)	1.0	0.99	1.0	0.99	0.99	0.99	0.99	0.99	0.98
Time per explicit task (average us)	1.83	3.97	5.65	7.17	9.07	11.17	12.98	14.74	16.41
Overhead per explicit task (synch %)	1.05	42.83	56.9	68.43	74.65	78.88	81.14	84.46	86.61
Overhead per explicit task (sched %)	13.57	32.67	45.15	57.86	66.3	72.4	76.81	79.86	82.62
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 May 4 07:42:23 PM CEST 2023, par2316

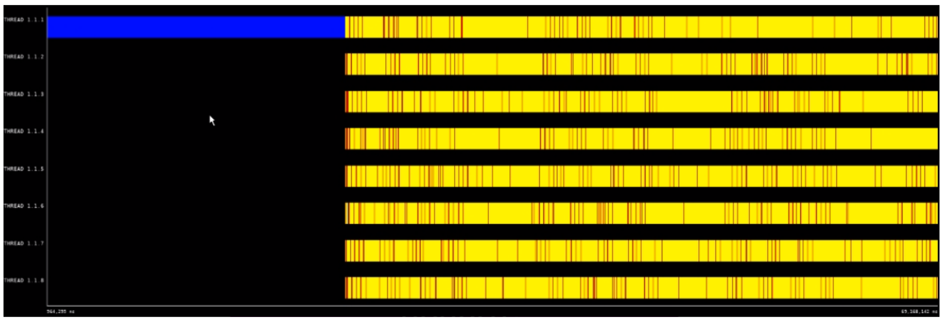


Figure 7: Execution of the tree implementation with 8 threads

The execution is far better, but it's also not that good at all. There's still a significant amount of time spent on task creation (almost all the time) and task synchronization, and that's the reason that causes the global efficiency to drastically drop, even though the load balancing and parallel fraction are just fine. The more threads are being used, the more time is spent in a single task. Here is the reason of the cut-off strategy, in order to avoid this huge number of tasks being generated all the execution and increase the granularity.

3.3 Tree vs Leaf

Now that we have seen both implementations so far, we clearly see that both are not feasible at all, specially the leaf one, so we will show in the next session how to improve the granularity of the tree implementation. Both have a tiny granularity that generate a lot of scheduling and synchronization overhead, but the leaf implementation generates fewer tasks (60000 vs 99660). The tree one suffers from both overheads, but the task creation is the main one, and the inverse for the leaf one, but in this case the task synchronization grows exponentially.

In conclusion, both are pretty bad in terms of performance, but only tree strategy is feasible and can be improved easily without changing the structure and flow of the program drastically.

3.4 Tree strategy with cut-off mechanism

In this section, we are going to modify our Tree strategy implementation to have a better control in our task granularity. The objective is to create a cut-off mechanism that allow us not only to increase the task granularity, but also control the maximum recursion level for task generation. This way, we need to edit the *multisort-omp.c* Tree strategy implementation to implement the cut-off mechanism. Once implemented, we compile it and then we executed with the *submit-omp.sh* script to validate its correctness. After that, we can commence the analysis of this implementation.

We are going to start by analysing the modelfactors tables generated with the *submit-strong-extrae.sh* script. With this we are also going to check the number of tasks generated for different cut-off levels(0, 1, 2, 4 and 8). To do this we will need to modify the cut-off variable in the *submit-strong-extrae.sh* script to test the different levels.

- **Cut-Off = 0**
  When executing with a cut-off value of 0 we obtain the following modelfactor tables. We can see that the number of executed task is equal to 7, as we expected.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.07	0.07	0.07	0.07	0.07	0.07
Speedup	1.00	1.67	2.42	2.31	2.31	2.30	2.28	2.29	2.28
Efficiency	1.00	0.84	0.60	0.39	0.29	0.23	0.19	0.16	0.14

Table 1: Analysis done on Sun May 14 08:46:07 PM CEST 2023, par2316

Overview of the Efficiency metrics in parallel fraction, ϕ=85.08%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.97%	95.07%	81.05%	51.04%	38.19%	30.42%	25.27%	21.57%	18.89%
Parallelization strategy efficiency	99.97%	95.40%	82.76%	54.96%	41.59%	33.60%	27.90%	23.86%	20.94%
Load balancing	100.00%	95.53%	85.32%	56.54%	45.80%	44.23%	33.28%	28.53%	24.85%
In execution efficiency	99.97%	99.86%	97.00%	97.20%	90.81%	75.95%	83.85%	83.61%	84.26%
Scalability for computation tasks	100.00%	99.65%	97.93%	92.86%	91.81%	90.56%	90.56%	90.41%	90.19%
IPC scalability	100.00%	99.33%	99.00%	98.70%	98.98%	99.42%	99.39%	99.24%	99.05%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.98%	99.98%	99.98%	99.97%	99.97%
Frequency scalability	100.00%	100.32%	98.93%	94.10%	92.78%	91.11%	91.13%	91.12%	91.09%

Table 2: Analysis done on Sun May 14 08:46:07 PM CEST 2023, par2316

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)	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0
LB (number of explicit tasks executed)	1.0	0.88	0.58	0.58	0.7	0.58	0.58	0.58	0.58
LB (time executing explicit tasks)	1.0	0.96	0.85	0.85	0.73	0.74	0.67	0.67	0.66
Time per explicit task (average us)	18995.52	19059.89	19393.54	20447.77	20680.12	20964.45	20960.86	20992.55	21036.64
Overhead per explicit task (synch %)	0.01	4.77	20.77	81.9	140.44	197.72	258.49	319.37	377.83
Overhead per explicit task (sched %)	0.02	0.03	0.04	0.04	0.04	0.05	0.05	0.07	0.06
Number of taskwait/taskgroup (total)	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0

Table 3: Analysis done on Sun May 14 08:46:07 PM CEST 2023, par2316

- **Cut-Off = 1**
  In the cut-off = 1 case we obtain that the number of executed tasks is 41 as we see down below.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.05	0.05	0.05	0.05	0.05	0.05
Speedup	1.00	1.74	2.57	2.91	2.86	3.27	3.26	3.28	3.25
Efficiency	1.00	0.87	0.64	0.48	0.36	0.33	0.27	0.23	0.20

Table 1: Analysis done on Sun May 14 08:49:17 PM CEST 2023, par2316

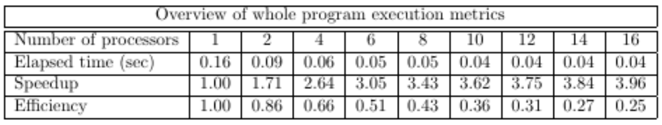
Overview of the Efficiency metrics in parallel fraction, ϕ=85.51%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.93%	99.20%	87.17%	74.12%	54.72%	56.46%	47.27%	40.22%	35.49%
Parallelization strategy efficiency	99.93%	99.60%	90.23%	80.14%	60.32%	63.18%	52.93%	45.09%	39.83%
Load balancing	100.00%	99.92%	90.88%	89.96%	76.64%	70.23%	56.43%	53.26%	55.12%
In execution efficiency	99.93%	99.68%	99.29%	89.08%	78.70%	89.97%	93.80%	84.65%	72.26%
Scalability for computation tasks	100.00%	99.60%	96.61%	92.49%	90.71%	89.36%	89.30%	89.21%	89.11%
IPC scalability	100.00%	99.69%	98.96%	99.07%	98.44%	98.59%	98.53%	98.43%	98.21%
Instruction scalability	100.00%	100.01%	100.00%	100.00%	99.99%	99.99%	99.99%	99.99%	99.98%
Frequency scalability	100.00%	99.90%	97.62%	93.36%	92.16%	90.64%	90.64%	90.64%	90.75%

Table 2: Analysis done on Sun May 14 08:49:17 PM CEST 2023, par2316

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)	41.0	41.0	41.0	41.0	41.0	41.0	41.0	41.0	41.0
LB (number of explicit tasks executed)	1.0	0.98	0.85	0.85	0.73	0.68	0.68	0.49	0.64
LB (time executing explicit tasks)	1.0	1.0	0.94	0.87	0.78	0.73	0.62	0.58	0.64
Time per explicit task (average us)	3232.73	3247.73	3476.2	3625.77	3750.05	3758.11	3991.08	3996.78	4197.15
Overhead per explicit task (synch %)	0.02	0.28	10.25	23.72	62.29	55.89	80.33	110.16	130.3
Overhead per explicit task (sched %)	0.04	0.1	0.15	0.15	0.2	0.18	0.28	0.23	0.23
Number of taskwait/taskgroup (total)	18.0	18.0	18.0	18.0	18.0	18.0	18.0	18.0	18.0

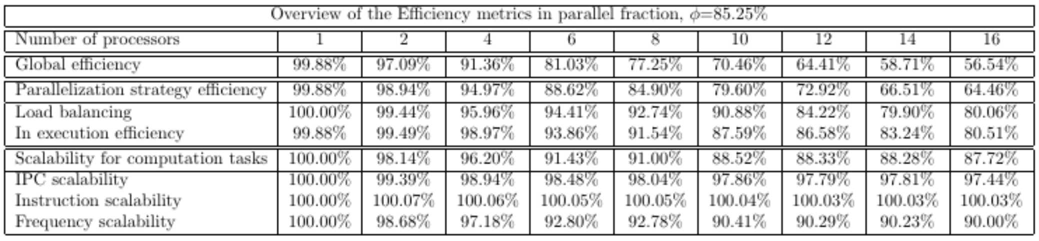
Table 3: Analysis done on Sun May 14 08:49:17 PM CEST 2023, par2316

- **Cut-Off = 2**
  With Cut-off = 2 we obtain that there are 189 explicit tasks executed.



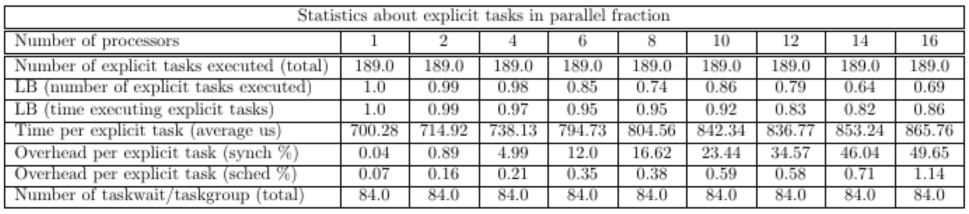
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.05	0.05	0.04	0.04	0.04	0.04
Speedup	1.00	1.71	2.64	3.05	3.43	3.62	3.75	3.84	3.96
Efficiency	1.00	0.86	0.66	0.51	0.43	0.36	0.31	0.27	0.25

Table 1: Analysis done on Sun May 14 08:52:33 PM CEST 2023, par2316



Overview of the Efficiency metrics in parallel fraction, ϕ=85.25%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.88%	97.09%	91.36%	81.03%	77.25%	70.46%	64.41%	58.71%	56.54%
Parallelization strategy efficiency	99.88%	98.94%	94.97%	88.62%	84.90%	79.60%	72.92%	66.51%	64.46%
Load balancing	100.00%	99.44%	95.96%	94.41%	92.74%	90.88%	84.22%	79.90%	80.06%
In execution efficiency	99.88%	99.49%	98.97%	93.86%	91.54%	87.59%	86.58%	83.24%	80.51%
Scalability for computation tasks	100.00%	98.14%	96.20%	91.43%	91.00%	88.52%	88.33%	88.28%	87.72%
IPC scalability	100.00%	99.39%	98.94%	98.48%	98.04%	97.86%	97.79%	97.81%	97.44%
Instruction scalability	100.00%	100.07%	100.06%	100.05%	100.05%	100.04%	100.03%	100.03%	100.03%
Frequency scalability	100.00%	98.68%	97.18%	92.80%	92.78%	90.41%	90.29%	90.23%	90.00%

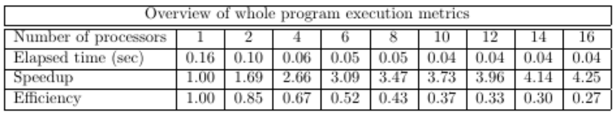
Table 2: Analysis done on Sun May 14 08:52:33 PM CEST 2023, par2316



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)	189.0	189.0	189.0	189.0	189.0	189.0	189.0	189.0	189.0
LB (number of explicit tasks executed)	1.0	0.99	0.98	0.85	0.74	0.86	0.79	0.64	0.69
LB (time executing explicit tasks)	1.0	0.99	0.97	0.95	0.95	0.92	0.83	0.82	0.86
Time per explicit task (average us)	700.28	714.92	738.13	794.73	804.56	842.34	836.77	853.24	865.76
Overhead per explicit task (synch %)	0.04	0.89	4.99	12.0	16.62	23.44	34.57	46.04	49.65
Overhead per explicit task (sched %)	0.07	0.16	0.21	0.35	0.38	0.59	0.58	0.71	1.14
Number of taskwait/taskgroup (total)	84.0	84.0	84.0	84.0	84.0	84.0	84.0	84.0	84.0

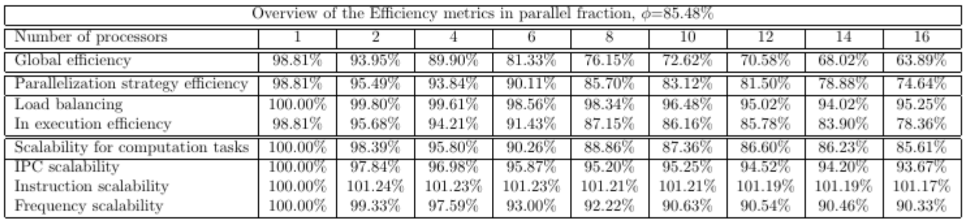
Table 3: Analysis done on Sun May 14 08:52:33 PM CEST 2023, par2316

- **Cut-Off = 4**
  In the case of cut-off value = 4 we can see looking at the modelfactor tables generated that the number of explicit tasks executed is 3317.



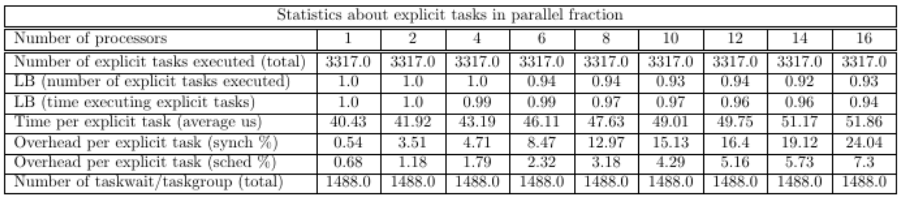
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.10	0.06	0.05	0.05	0.04	0.04	0.04	0.04
Speedup	1.00	1.69	2.66	3.09	3.47	3.73	3.96	4.14	4.25
Efficiency	1.00	0.85	0.67	0.52	0.43	0.37	0.33	0.30	0.27

Table 1: Analysis done on Sun May 14 09:03:51 PM CEST 2023, par2316



Overview of the Efficiency metrics in parallel fraction, ϕ=85.48%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	98.81%	93.95%	89.90%	81.33%	76.15%	72.62%	70.58%	68.02%	63.89%
Parallelization strategy efficiency	98.81%	95.49%	93.84%	90.11%	85.70%	83.12%	81.50%	78.88%	74.64%
Load balancing	100.00%	99.80%	99.61%	98.56%	98.34%	96.48%	95.02%	94.02%	95.25%
In execution efficiency	98.81%	95.68%	94.21%	91.43%	87.15%	86.16%	85.78%	83.90%	78.36%
Scalability for computation tasks	100.00%	98.39%	95.80%	90.26%	88.86%	87.36%	86.60%	86.23%	85.61%
IPC scalability	100.00%	97.84%	96.98%	95.87%	95.20%	95.25%	94.52%	94.20%	93.67%
Instruction scalability	100.00%	101.24%	101.23%	101.23%	101.21%	101.21%	101.19%	101.19%	101.17%
Frequency scalability	100.00%	99.33%	97.59%	93.00%	92.22%	90.63%	90.54%	90.46%	90.33%

Table 2: Analysis done on Sun May 14 09:03:51 PM CEST 2023, par2316



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)	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0
LB (number of explicit tasks executed)	1.0	1.0	1.0	0.94	0.94	0.93	0.94	0.92	0.93
LB (time executing explicit tasks)	1.0	1.0	0.99	0.99	0.97	0.97	0.96	0.96	0.94
Time per explicit task (average us)	40.43	41.92	43.19	46.11	47.63	49.01	49.75	51.17	51.86
Overhead per explicit task (synch %)	0.54	3.51	4.71	8.47	12.97	15.13	16.4	19.12	24.04
Overhead per explicit task (sched %)	0.68	1.18	1.79	2.32	3.18	4.29	5.16	5.73	7.3
Number of taskwait/taskgroup (total)	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0

Table 3: Analysis done on Sun May 14 09:03:51 PM CEST 2023, par2316

- **Cut-Off = 8**
  Finally, with a cut-off level of 8 the number of explicit tasks executed is 57173 as we see in the tables below.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.24	0.22	0.17	0.15	0.14	0.14	0.14	0.14	0.14
Speedup	1.00	1.12	1.45	1.66	1.70	1.73	1.72	1.72	1.73
Efficiency	1.00	0.56	0.36	0.28	0.21	0.17	0.14	0.12	0.11

Table 1: Analysis done on Sun May 14 08:57:31 PM CEST 2023, par2316

Overview of the Efficiency metrics in parallel fraction, ϕ=90.39%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	87.82%	49.93%	33.29%	26.39%	20.33%	16.65%	13.80%	11.81%	10.41%
Parallelization strategy efficiency	87.82%	59.16%	47.05%	37.89%	30.30%	25.40%	20.94%	17.98%	15.40%
Load balancing	100.00%	96.53%	98.20%	95.78%	92.70%	90.72%	89.95%	90.67%	91.77%
In execution efficiency	87.82%	61.29%	47.91%	39.56%	32.68%	28.00%	23.28%	19.83%	16.78%
Scalability for computation tasks	100.00%	84.39%	70.75%	69.66%	67.09%	65.56%	65.90%	65.70%	67.58%
IPC scalability	100.00%	74.21%	62.32%	63.85%	61.79%	61.84%	62.80%	62.63%	64.74%
Instruction scalability	100.00%	114.05%	114.13%	114.08%	114.08%	114.09%	114.06%	114.08%	114.08%
Frequency scalability	100.00%	99.71%	99.48%	95.64%	95.17%	92.91%	92.00%	91.96%	91.50%

Table 2: Analysis done on Sun May 14 08:57:31 PM CEST 2023, par2316

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)	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0
LB (number of explicit tasks executed)	1.0	1.0	0.98	0.97	0.98	0.99	0.96	0.98	0.94
LB (time executing explicit tasks)	1.0	0.98	0.99	0.99	0.99	0.99	0.98	0.98	0.98
Time per explicit task (average us)	2.85	5.05	7.25	9.02	11.67	14.19	17.08	19.97	22.55
Overhead per explicit task (synch %)	7.42	34.86	43.93	50.56	54.83	57.74	60.53	61.64	63.33
Overhead per explicit task (sched %)	9.01	19.86	30.23	37.61	44.5	48.95	52.89	55.92	58.56
Number of taskwait/taskgroup (total)	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0

Table 3: Analysis done on Sun May 14 08:57:31 PM CEST 2023, par2316

After having analysed the distinct obtained tables we can conclude that the best cut-off option from those we have discussed above is 4 because of its better performance. Besides, incrementing the number of threads doesn't improve the global efficiency of the cut-off strategy probably because a lot of time is going to be spent in syncronization overheads.

Now we are going to explore with Paraver the multisort execution with 8 threads generated (see Figure 8).

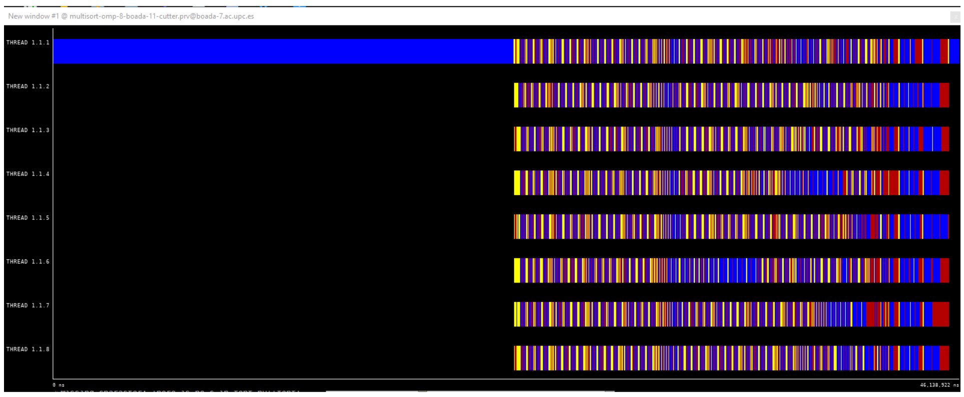


Figure 8: Execution of tree implementation with the cut-off mechanism with 8 threads

Looking at the generated Paraver trace it seems that the tree strategy with the cut-off mechanism is greater than the previous strategy because in this case more time is spent executing useful code than synchronising or doing scheduling.

Last but not least, we are going to explore different cut-off values based on the number of threads used. For this we need to execute the *submit-cutoff-omp.sh* script specifying as a parameter the number of threads that we want to use. In our case we used 8 threads. After executing the script we obtain a PostScript file which shows different execution times depending on the cut-off level. Looking at the generated plot (see Figure 9) we see that the best cut-off level for our problem size(128) and 8 threads is 6 because its execution time is the best of all.

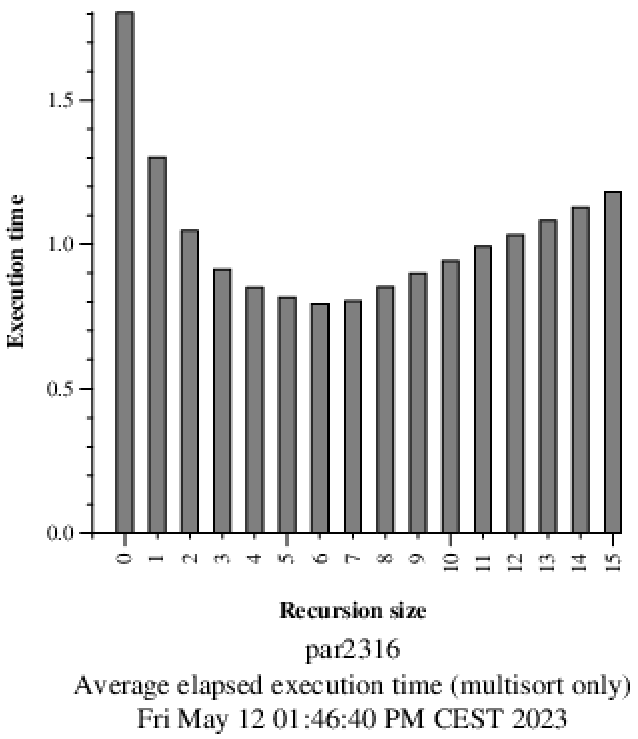
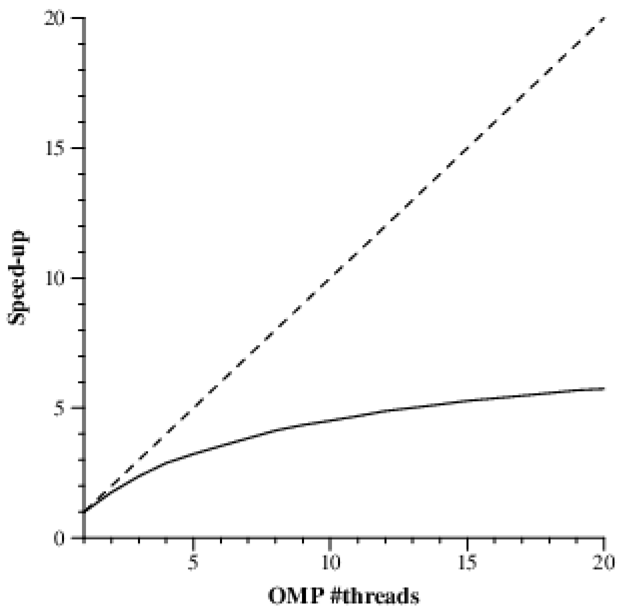
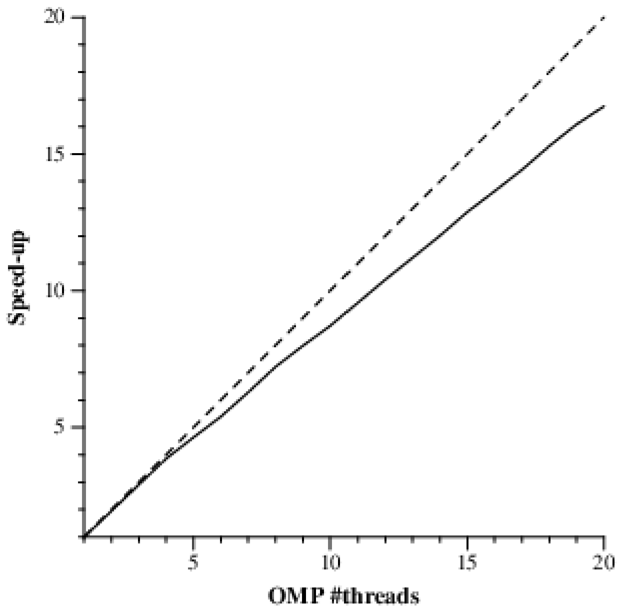


Figure 9: PostScript plot genereted with the *submit-cutoff-omp.sh* script with 8 threads

To look over the optimum cut-off value that we have obtained we are going to execute the *submit-strong-omp.sh* script with the cut-off value = 6. After that, we get the plot seen in Figure 10. With this we can see that the optimum value gives us a good speed-up for the multisort function that is close to the ideal speed-up.



par2316
Speed-up wrt sequential time (complete application)
Generated by par2316 on Mon May 15 04:17:34 PM CEST 2023



par2316
Speed-up wrt sequential time (multisort funtion only)
Generated by par2316 on Mon May 15 04:17:34 PM CEST 2023

Figure 10: Strong scalability plot for cut-off = 6

4 Shared Memory parallelization using OpenMP task dependencies

To conclude , we want to introduce another implementation of the tree strategy using the Cut-Off parameter but this time using explicit OpenMP dependencies instead of traditional task synchronization such as taskwait and taskgroup, and we will see if it's better or the same as the plain Cut-Off implementation.

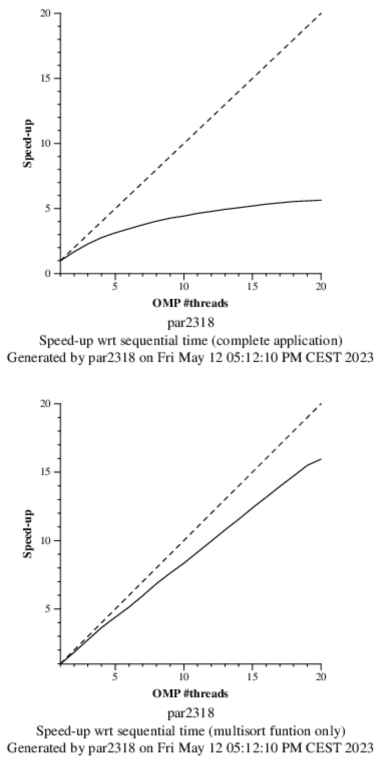


Figure 11: Scalability Plots for the Depend based implementation

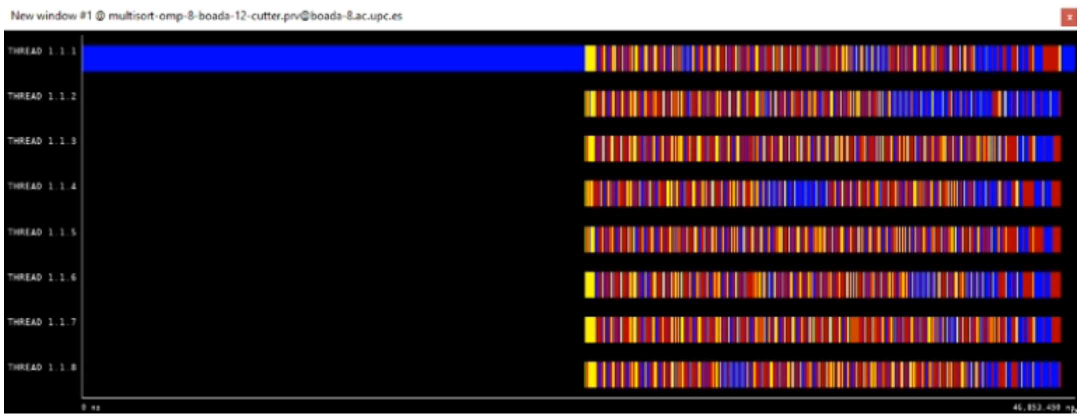
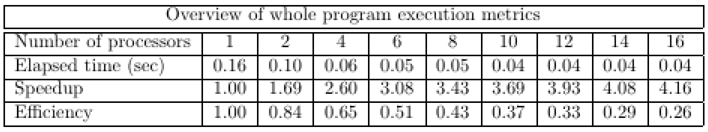
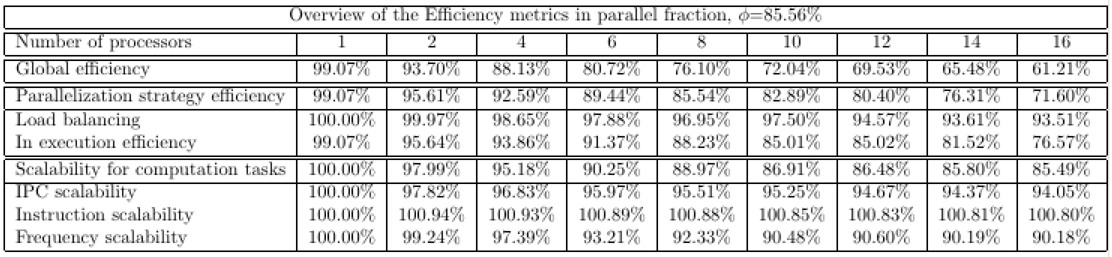


Figure 12: Execution of this implementation using 8 threads and CUTOFF=4



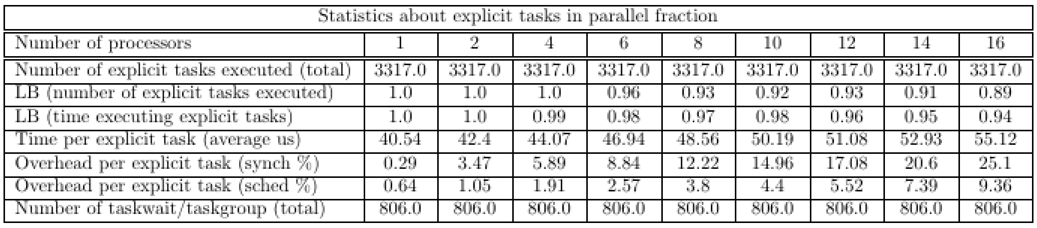
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.10	0.06	0.05	0.05	0.04	0.04	0.04	0.04
Speedup	1.00	1.69	2.60	3.08	3.43	3.69	3.93	4.08	4.16
Efficiency	1.00	0.84	0.65	0.51	0.43	0.37	0.33	0.29	0.26

Table 1: Analysis done on Sun May 14 08:32:44 PM CEST 2023, par2316



Overview of the Efficiency metrics in parallel fraction, ϕ=85.56%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.07%	93.70%	88.13%	80.72%	76.10%	72.04%	69.53%	65.48%	61.21%
Parallelization strategy efficiency	99.07%	95.61%	92.59%	89.44%	85.54%	82.89%	80.40%	76.31%	71.60%
Load balancing	100.00%	99.97%	98.65%	97.88%	96.95%	97.50%	94.57%	93.61%	93.51%
In execution efficiency	99.07%	95.64%	93.86%	91.37%	88.23%	85.01%	85.02%	81.52%	76.57%
Scalability for computation tasks	100.00%	97.99%	95.18%	90.25%	88.97%	86.91%	86.48%	85.80%	85.49%
IPC scalability	100.00%	97.82%	96.83%	95.97%	95.51%	95.25%	94.67%	94.37%	94.05%
Instruction scalability	100.00%	100.94%	100.93%	100.89%	100.88%	100.85%	100.83%	100.81%	100.80%
Frequency scalability	100.00%	99.24%	97.39%	93.21%	92.33%	90.48%	90.60%	90.19%	90.18%

Table 2: Analysis done on Sun May 14 08:32:44 PM CEST 2023, par2316



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)	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0
LB (number of explicit tasks executed)	1.0	1.0	1.0	0.96	0.93	0.92	0.93	0.91	0.89
LB (time executing explicit tasks)	1.0	1.0	0.99	0.98	0.97	0.98	0.96	0.95	0.94
Time per explicit task (average us)	40.54	42.4	44.07	46.94	48.56	50.19	51.08	52.93	55.12
Overhead per explicit task (synch %)	0.29	3.47	5.89	8.84	12.22	14.96	17.08	20.6	25.1
Overhead per explicit task (sched %)	0.64	1.05	1.91	2.57	3.8	4.4	5.52	7.39	9.36
Number of taskwait/taskgroup (total)	806.0	806.0	806.0	806.0	806.0	806.0	806.0	806.0	806.0

Table 3: Analysis done on Sun May 14 08:32:44 PM CEST 2023, par2316

Looking at this implementation we see that the complexity of the code gets higher, if the algorithm was even longer it would be unsustainable to write, and looking at the paraver and modelfactor tables we cannot say that is much better, or even better than the last one,even though the scalability remains the same as the last implementation. It looks like there's not a big amount of time spent on overheads, it has a good load balancing and a great global efficiency but it decays a bit when the number of processors is increased (More or less the statistics are the same as the CutOff one, but just slightly better).

As a conclusion, in terms of programmability, this implementation makes the programmer do more complex code, because you have to know exactly which variables cause the data dependencies and the address of that variables too which can easily lead to bugs. It's not worth the changes just for a tiny speed-up unless performance is key and the maintainability of the code is not relevant (Take for instance an algorithm that won't be changed in the future for sure).