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

Neus Mayol and Pol Verdura
group 11 - (par1117 and par1121)

Spring 2022-23

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

1

Task decomposition analysis for Mergesort

1.1 Task decomposition analysis with *Tareador*

To achieve a leaf decomposition parallelization we've added tareador commands too create task every time that we arrive to the mergesort base cases. In other words: tasks are created when `basicmerge` and `basicsort` are called. The modified code is attached at `multisort-tareador-leaf.c`.

The resulting TDG, shown in figure 1.1, displays tasks as we expected to: we can see lots of small red tasks corresponding to each base case that the algorithm has found during its execution.

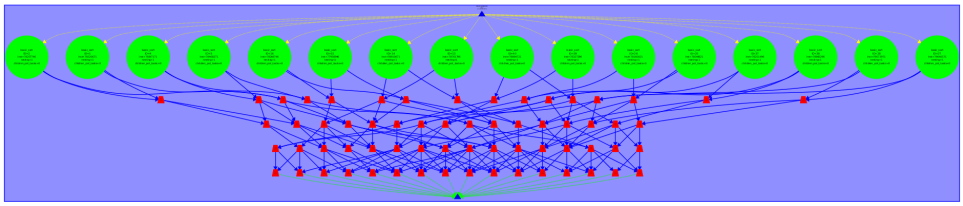

Figure 1.1: TDG using a leaf decomposition strategy.

If we want to have a tree decomposition parallelization we need to create extra tasks each time we do a new recursive call. At the end we will have an extra task each time we call the function `multisort`, `merge`. Complete code is attached at `mergesort-tareador-tree.c`.

Resulting TDG is displayed in figure 1.2, where we can see how tasks are created recursively.

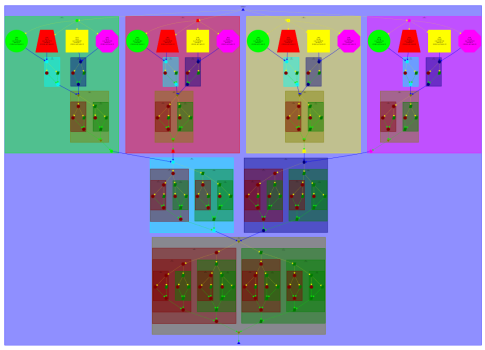

Figure 1.2: TDG using a tree decomposition strategy.

Dividing the code into tasks will later allow us to parallelize the algorithm. Anyway, there's a big data race problem when it comes to using this strategies. Since tasks need information from other tasks in order to run correctly, we need to control that tasks start by the time all their dependencies have finished. The problem should be avoided using a `taskgroup` clause to each `multisort` recursive call, since this *OpenMP* clause has an implicit `taskwait` clause that will make a task execution wait until all tasks in the group have finished. The same problem can also be solved with the clause `taskwait`, the only difference between both is that `taskwait` only waits tasks from one level below, so we will need to add one clause each time we refine tasks.

2

Shared-memory parallelisation with *OpenMP* tasks

2.1 Leaf strategy in *OpenMP*

Since we want to avoid dataraces, we need to parallelise using an *OpenMP* clause such as `taskgroup` or `taskwait`. That's why we created two different verions of `multisort-omp.c` code, that parallelise using `taskgroup` (`multisort-omp-leaf-taskgroup.c`) or `taskwait`(`multisort-omp-leaf-taskwait.c`). Modificiations on `multisort-omp-leaf-taskgroup.c` include:

- Addition of a `#pragma omp task` before each base case function (`basicsort` and `basicmerge`).
- A `taskgroup` definition grouping all recursive calls to `multisort` in procedure `multisort`.
- A `taskgroup` definition grouping both calls of `merge` in procedure `multisort`.
- A `#pragma omp parallel` clause followed by a `#pragma omp single` definition in `main` function, just before the call to `multisort`.

These changes prevent the program from causing data races, since each recursive call won't start `merge` its procedures until the previous `multisort` procedures have finished.

On the other hand, to get a leaf decomposition with `taskwait` we need to add at the end of each recursive call a `taskwait` clause so all tasks will wait the tasks that they created.

To obtain `multisort-omp-leaf-taskwait.c` we had to change the clauses `#pragma omp taskgroup` from `multisort-omp-leaf-taskgroup.c` to `#pragma omp taskwait` clauses and move them at the end of the recursive calls to add this virtual barrier.
Since we avoided data races in two different ways, we found a significant difference between the use of the clauses when it comes to `multisort` execution time. Execution times of the clauses are displayed on table 2.1, where we can see that `taskgroup` command performance is better than `taskwait` in both 2 and 4 thread execution. Then, following the assignment statement, we ran the `submit-strong-omp.sh`

		Used clause	
		`taskgroup`	`taskwait`
# threads	2	3.147	4.409
	4	2.208	4.328

Table 2.1: `multisort` execution time (in seconds), using leaf decomposition strategy and decomposed using two different clauses (`taskgroup` o `taskwait`), for 2 and 4 threads.

script, which executes the binary with different numbers of threads. Resulting speedup displayed in output files is not reasonable: it stops increasing as soon as we arrive to 10 threads. Let's analyze it with `modelfactors`!

2.1.1 Analysis with `modelfactors` and *Paraver*

The execution and the `modelfactors` analysis has resulted into Table 2.2, Table 2.3 and Table 2.4. At first sight, we can see that the elapsed time shown in Table 2.2 is far away as good as what we expected to. The addition of new processors does no give us any improvement. Speedup has the same tendency, sometimes is worst than the original version! That's why the global program efficiency, displayed also in Table 2.2, is also poor.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.21	0.25	0.23	0.21	0.22	0.25	0.23	0.31	0.25
Speedup	1.00	0.85	0.94	1.01	0.99	0.87	0.93	0.70	0.86
Efficiency	1.00	0.42	0.24	0.17	0.12	0.09	0.08	0.05	0.05

Table 2.2: Analysis done on Mon May 1 12:26:47 PM CEST 2023, par1117

Table 2.3 displays some data that made us suspect: Load balancing gets worst every time that we increase the number of processors. Table 2.3 also shows a poor scalability for computation tasks. And moreover we can appreciate that the parallel fraction of our program is almost 90% so we won't be able to fully-parallelise because it's the time we need to initialize tasks.

Overview of the Efficiency metrics in parallel fraction, ϕ=89.38%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	92.01%	41.09%	23.16%	16.46%	11.87%	7.90%	7.26%	6.12%	5.06%
Parallelization strategy efficiency	92.01%	54.16%	36.77%	27.46%	21.12%	16.11%	13.79%	11.88%	10.10%
Load balancing	100.00%	97.86%	93.21%	63.67%	39.79%	24.91%	23.62%	19.45%	16.20%
In execution efficiency	92.01%	55.34%	39.45%	43.13%	53.09%	64.65%	58.36%	61.06%	62.32%
Scalability for computation tasks	100.00%	75.86%	62.98%	59.94%	56.18%	49.04%	52.63%	51.53%	50.14%
IPC scalability	100.00%	68.09%	56.49%	56.42%	53.54%	48.14%	51.51%	50.49%	49.30%
Instruction scalability	100.00%	112.39%	113.95%	112.92%	111.84%	110.65%	111.15%	111.22%	110.88%
Frequency scalability	100.00%	99.13%	97.84%	94.08%	93.80%	92.06%	91.94%	91.75%	91.72%

Table 2.3: Analysis done on Mon May 1 12:26:47 PM CEST 2023, par1117

We think that the most alarming row in Table 2.4 is overhead per explicit task synchronisation, which increases brutally every time that processor number is increased.

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	0.73	0.75	0.78	0.78	0.77	0.79	0.8	0.81
LB (time executing explicit tasks)	1.0	0.81	0.81	0.77	0.78	0.77	0.78	0.8	0.82
Time per explicit task (average us)	2.79	3.6	4.18	4.15	4.11	4.1	4.13	4.15	4.14
Overhead per explicit task (synch %)	0.87	67.4	170.6	318.74	509.51	837.07	928.66	1128.02	1399.8
Overhead per explicit task (sched %)	9.43	35.48	46.02	33.72	27.22	22.8	25.27	23.62	24.35
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Table 2.4: Analysis done on Mon May 1 12:26:47 PM CEST 2023, par1117

As we can see on Figure 2.1, the average amount of instant simultaneous tasks is almost 4, which means that we don't fulfill the resources of each thread.

Moreover, thread 1 creates the tasks. Threads 2 to 8 are fully dedicated to the execution of tasks, but thread 1 also executes some tasks as we can see on Figure 2.1, that's why we have such a poor load balance.

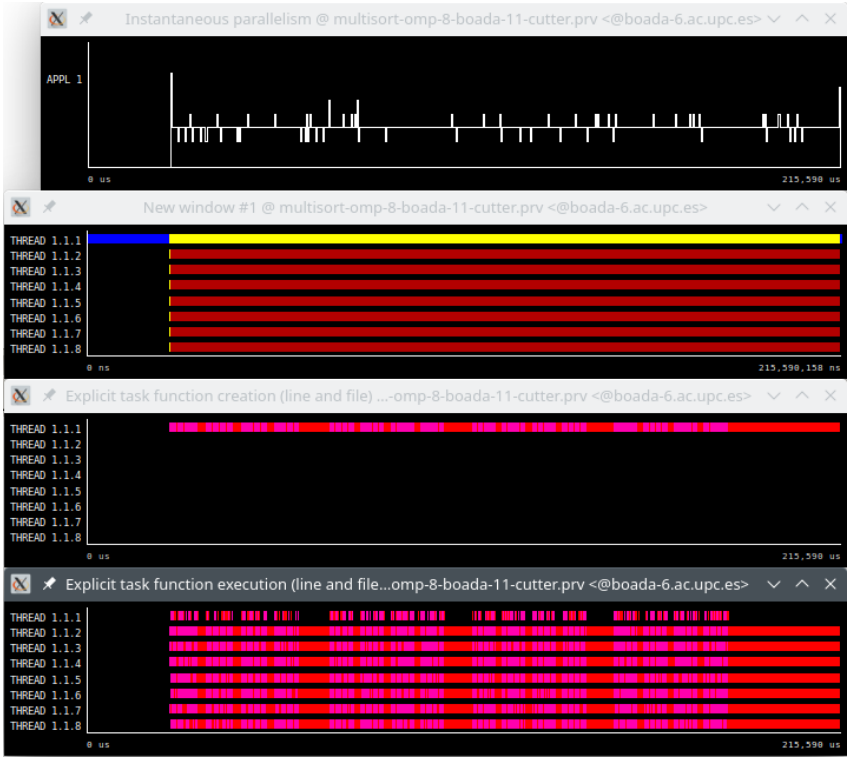

Figure 2.1: Explicit task creation and execution of leaf strategy

2.2 Tree strategy in *OpenMP*

In Table 2.6, we see that with tree strategy load balance almost doesn't decrease whereas with leaf strategy was notorious when we increased the number of processes, that may be because before, only one task generated all the tasks. Moreover, we can see that the parallel fraction increases because the task generation is parallelised.

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	0.23	0.22	0.23	0.22	0.22
Speedup	1.00	0.85	1.06	1.15	1.16	1.18	1.17	1.20	1.18
Efficiency	1.00	0.42	0.26	0.19	0.14	0.12	0.10	0.09	0.07

Table 2.5: Analysis done on Tue May 2 09:50:42 AM CEST 2023, par1117

Overview of the Efficiency metrics in parallel fraction, ϕ=92.83%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	75.89%	37.28%	24.14%	18.17%	14.14%	11.28%	9.43%	8.38%	7.04%
Parallelization strategy efficiency	75.89%	50.95%	40.31%	30.39%	24.43%	19.65%	16.44%	14.34%	12.21%
Load balancing	100.00%	98.00%	96.91%	94.16%	91.99%	92.34%	93.30%	91.68%	92.46%
In execution efficiency	75.89%	51.99%	41.60%	32.28%	26.56%	21.28%	17.62%	15.64%	13.20%
Scalability for computation tasks	100.00%	73.17%	59.87%	59.78%	57.89%	57.41%	57.36%	58.45%	57.68%
IPC scalability	100.00%	62.48%	50.84%	53.38%	51.34%	52.69%	53.55%	54.68%	54.08%
Instruction scalability	100.00%	119.23%	119.31%	119.27%	119.44%	119.52%	119.30%	119.16%	119.26%
Frequency scalability	100.00%	98.23%	98.72%	93.91%	94.39%	91.17%	89.79%	89.71%	89.43%

Table 2.6: Analysis done on Tue May 2 09:50:42 AM CEST 2023, par1117

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	0.97	0.98	0.98	0.96	0.97	0.97
LB (time executing explicit tasks)	1.0	0.97	0.99	0.99	0.99	0.99	0.99	0.97	0.99
Time per explicit task (average us)	1.81	4.38	6.59	8.78	11.32	14.34	17.08	19.28	22.93
Overhead per explicit task (synch %)	10.16	42.86	50.39	56.09	59.15	60.57	62.64	63.32	64.39
Overhead per explicit task (sched %)	13.45	25.52	35.12	43.34	48.42	52.57	55.54	57.45	59.48
Number of taskwait/taskgroup (total)	49152.0	49152.0	49152.0	49152.0	49152.0	49152.0	49152.0	49152.0	49152.0

Table 2.7: Analysis done on Tue May 2 09:50:42 AM CEST 2023, par1117

On Figure 2.2, we see that the average instant parallelism is 6 tasks, taking into account that the number of CPU's is 8, we don't have enough tasks to feed all threads simultaneously, but it's better than leaf strategy. But this time, we can also see that all threads generate tasks. The time execution is still bad because now ll threads spend most of the time synchronizing the tasks.

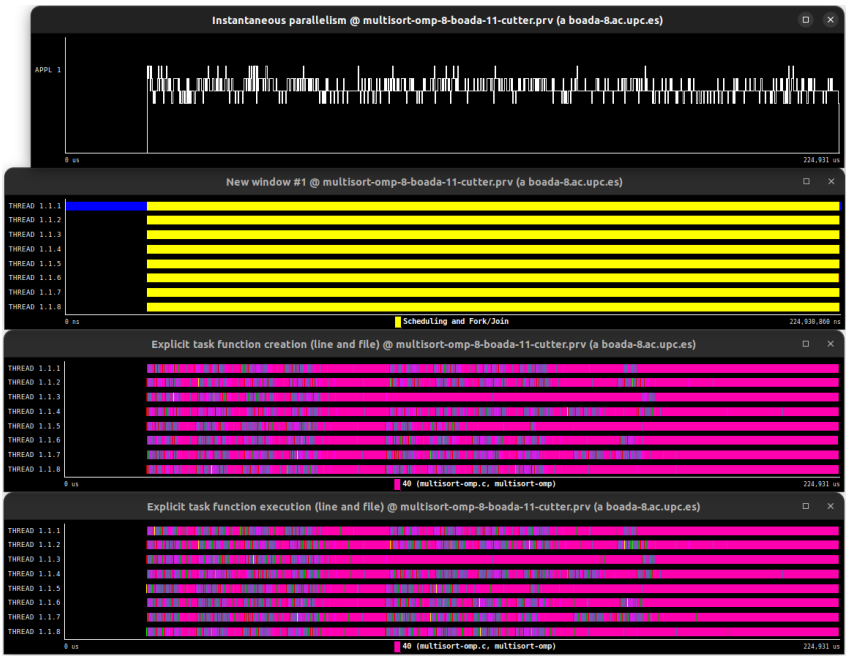

Figure 2.2: Explicit task creation and execution of tree strategy

In Tables 2.4 and 2.7 we can see that the number of explicit tasks created is 53248 and 99669 with leaf and tree strategy respectively. We generate so many tasks because the granularity is really fine grained and moreover, in tree strategy we generate extra tasks to parallelise the task creation.

2.3 Task granularity control: the *cut-off* mechanism

Furthermore, we implemented a *cut-off* mechanism to regulate the number of created tasks by adding a new variable `level` that calculates the level of the recursion. This variable is used by the clause `# pragma omp task final(level >= CUTOFF)`, that we will use to cut the task creation. Following the lab statement, we ran the strategy using five different values for the `CUTOFF` variable: 0, 1, 2, 4 and 8. We have detected some difference between the number of tasks created. For `CUTOFF = 0` value is 7, for `CUTOFF = 1` we created 41 tasks, `CUTOFF = 2` has 189 tasks, while `CUTOFF = 4` created 33170 tasks and `CUTOFF = 8` 57173.

These values alsomake the number of threads to change significally, that's why we believed that the best way to immplement the *cut-off* strategy is assigning `CUTOFF = 4`.
Execution data of `CUTOFF = 4` is displayed in the following `modelfactors` tables. First, we compare

table 2.5 (the no *cut-off* tree strategy that we executed above) and table 2.8. We have noticed a positive increment of the efficiency and the speedup, which means that now the parallelisation strategy its working well: We are reducing execution time.

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	1.68	2.65	3.08	3.51	3.73	3.96	4.13	4.29
Efficiency	1.00	0.84	0.66	0.51	0.44	0.37	0.33	0.30	0.27

Table 2.8: Analysis done on Tue May 15 09:40:01 AM CEST 2023, par1117

Continuing with the analysis, we compare table 2.6 and table 2.9, where we first notice that implementing the *cut-off* strategy implies a decrement of the parallel fraction. Besides that, almost all fields in 2.9 show how does the implementation of the *cut-off* strategy improves the program overall performance: we have a better scalability, parallelization strategy efficiency and a small improvement in load balancing.

Overview of the Efficiency metrics in parallel fraction, ϕ=85.73%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	98.95%	93.42%	90.21%	81.78%	79.84%	74.42%	72.62%	69.42%	67.80%
Parallelization strategy efficiency	98.95%	95.22%	93.72%	91.02%	88.78%	85.39%	83.77%	80.35%	78.65%
Load balancing	100.00%	99.70%	99.60%	99.50%	98.99%	98.04%	95.29%	97.21%	97.34%
In execution efficiency	98.95%	95.50%	94.09%	91.48%	89.69%	87.09%	87.91%	82.66%	80.80%
Scalability for computation tasks	100.00%	98.11%	96.26%	89.85%	89.93%	87.16%	86.70%	86.40%	86.20%
IPC scalability	100.00%	97.65%	96.75%	96.03%	95.46%	95.08%	94.64%	94.28%	94.07%
Instruction scalability	100.00%	101.22%	101.23%	101.22%	101.21%	101.21%	101.20%	101.21%	101.20%
Frequency scalability	100.00%	99.26%	98.28%	92.44%	93.08%	90.57%	90.52%	90.55%	90.55%

Table 2.9: Analysis done on Tue May 15 09:40:01 AM CEST 2023, par1117

Finally, we see that the number of tasks created in table 2.10 is much lower than the one displayed in table 2.7, since now the cut-off strategy limits the number of created tasks. This also implies a significant decrement of the number of `taskwait` and `taskgroup`, which can also be appreciated in both tables. We can also see how does the "Time for explicit task" increase. That may be the reason why the overhead percentages are more reasonable than the ones displayed in table 2.7, which means that we balanced the overhead time and the task execution time.

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	0.93	0.93	0.93	0.94
LB (time executing explicit tasks)	1.0	1.0	0.99	1.0	0.99	0.98	0.97	0.97	0.97
Time per explicit task (average us)	40.23	41.71	42.7	46.05	46.3	48.16	48.7	49.5	50.72
Overhead per explicit task (synch %)	0.37	3.63	4.79	7.3	9.07	12.37	13.86	17.52	16.97
Overhead per explicit task (sched %)	0.7	1.35	1.83	2.38	3.23	4.16	4.75	5.68	8.22
Number of taskwait/taskgroup (total)	682.0	682.0	682.0	682.0	682.0	682.0	682.0	682.0	682.0

Table 2.10: Analysis done on Tue May 15 09:40:01 AM CEST 2023, par1117

Let's change the point of view of the analysis: now we'll take a look at the traces generated with *Paraver* (figure 2.3 right) and compare them with the traces obtained above with the tree strategy (2.3

left). We see that the execution time signfically decreases and the parallelism stays high during all the execution with *cut-off* mode: we now have a larger instantaneous parallelism.

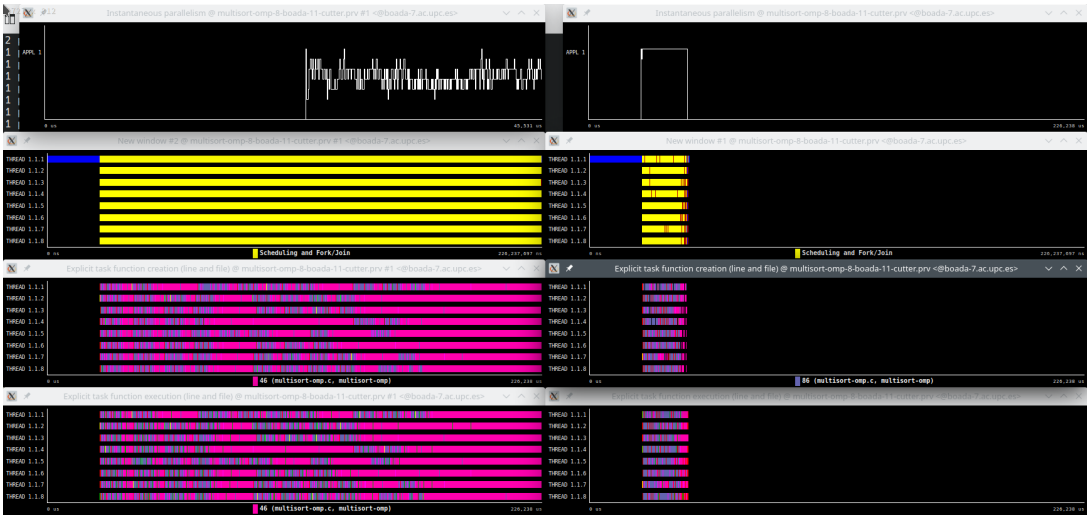

Figure 2.3: Explicit task creation and execution of tree strategy (left) and explicit task creation and execution of tree strategy with `CUTOFF = 4` (right).

Following with the assignement statement, we now run an execution with 8 threads an using the file `submit-cut-off-omp.sh`, which stores the elapsed time using different *cut-off* values. The output displayed is:

```
0 2.286458 \\ 1 0.864340 \\ 2 0.521262  \\ 3 0.349203 \\ 4 0.299432  \\ 5 0.273970 \\
6 0.263795 \\ 7 0.259532 \\ 8 0.276022 \\ 9 0.288265 \\ 10 0.314776 \\
11 0.343509 \\ 12 0.368868 \\ 13 0.397888 \\ 14 0.428587 \\ 15 0.459453
```

Which means that the best execution performance has happened when the *cut-off* value was 7. We'll use this value to execute `submit-omp-strong.sh` and see how the scalability plot turns out.
We have obtained two plots, displayed in figure 2.4. Right plot shows a significant improvement in performance of `multisort` function, which has almost an ideal speedup. But overall performance is speedup is poor, although it increases discretely. The reason behind the poor performance could be that the rest of the program is not parallelised and prevents the speedup from increasing ideally.

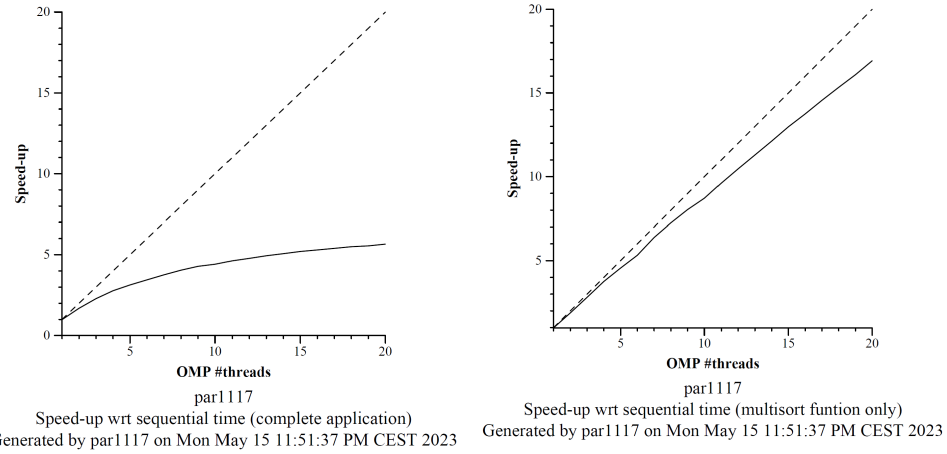

Figure 2.4: Speedup plots of sequential execution (left) and parallel tree execution with `CUTOFF = 7` (right).

3

Shared-memory parallelisation with *OpenMP* task using dependencies

Finally, we add som new OMP clauses to stablish dependencies between positions of the vector. The execution of the code results on the following `modelfactors` tables, which do not show a significant improvement compared with the tables of last chapter.

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	1.70	2.66	3.12	3.45	3.72	3.92	4.06	4.17
Efficiency	1.00	0.85	0.67	0.52	0.43	0.37	0.33	0.29	0.26

Table 3.1: Analysis done on Mon May 15 11:13:35 PM CEST 2023, par1121

Overview of the Efficiency metrics in parallel fraction, ϕ=85.69%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.07%	94.66%	91.07%	82.16%	76.35%	72.53%	69.35%	66.00%	60.84%
Parallelization strategy efficiency	99.07%	95.22%	92.72%	90.06%	85.54%	83.11%	80.10%	77.09%	71.18%
Load balancing	100.00%	100.00%	98.97%	98.58%	97.66%	96.32%	92.07%	96.00%	95.09%
In execution efficiency	99.07%	95.22%	93.68%	91.36%	87.59%	86.29%	87.00%	80.30%	74.86%
Scalability for computation tasks	100.00%	99.42%	98.22%	91.23%	89.26%	87.27%	86.58%	85.61%	85.47%
IPC scalability	100.00%	97.62%	97.12%	96.04%	95.65%	95.20%	94.50%	94.12%	93.88%
Instruction scalability	100.00%	100.96%	100.93%	100.90%	100.88%	100.86%	100.84%	100.81%	100.81%
Frequency scalability	100.00%	100.88%	100.21%	94.15%	92.50%	90.89%	90.85%	90.24%	90.32%

Table 3.2: Analysis done on Mon May 15 11:13:35 PM CEST 2023, par1121

Overview of the Efficiency metrics in parallel fraction, ϕ=85.69%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.07%	94.66%	91.07%	82.16%	76.35%	72.53%	69.35%	66.00%	60.84%
Parallelization strategy efficiency	99.07%	95.22%	92.72%	90.06%	85.54%	83.11%	80.10%	77.09%	71.18%
Load balancing	100.00%	100.00%	98.97%	98.58%	97.66%	96.32%	92.07%	96.00%	95.09%
In execution efficiency	99.07%	95.22%	93.68%	91.36%	87.59%	86.29%	87.00%	80.30%	74.86%
Scalability for computation tasks	100.00%	99.42%	98.22%	91.23%	89.26%	87.27%	86.58%	85.61%	85.47%
IPC scalability	100.00%	97.62%	97.12%	96.04%	95.65%	95.20%	94.50%	94.12%	93.88%
Instruction scalability	100.00%	100.96%	100.93%	100.90%	100.88%	100.86%	100.84%	100.81%	100.81%
Frequency scalability	100.00%	100.88%	100.21%	94.15%	92.50%	90.89%	90.85%	90.24%	90.32%

Table 3.3: Analysis done on Mon May 15 11:13:35 PM CEST 2023, par1121

As we see over Figure 3.1 lasts 46.828.935 instructions, whereas without dependencies lasts 226.237.697

tasks.

Figure 3.1: Explicit task decomposition with dependencies

This last version was the worst to program and the improvement, as shown in figure 3.2 speedup plots, is not noticeable in comparision with the ones obtained using only the *cut-off* strategy.

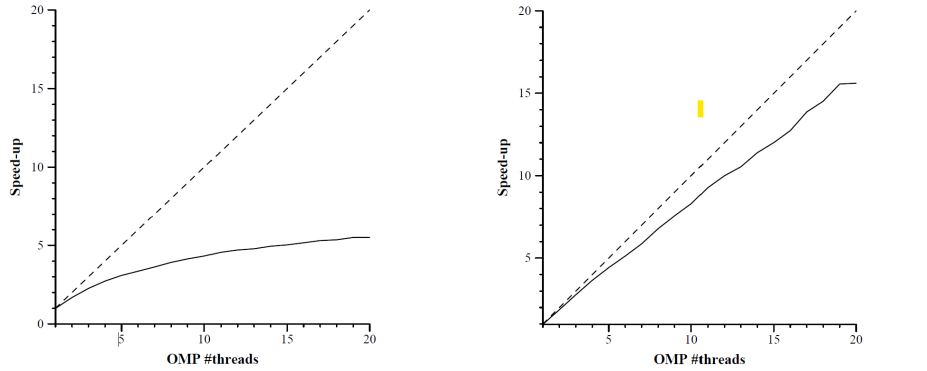

Figure 3.2: Speedup plots applying *cut-off* an dependencies strategy