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

Víctor Asenjo Carvajal - par1302
Ferran De La Varga Antoja - par1305

May 2022
Semester of Spring 2021-2022

Index

1 Introduction

In this laboratory assignment, we will learn and understand as much as possible how the divide and conquer strategy works with a parallel approach to a *Mergesort* program.

So the main objective of this lab is to learn how we can parallelize a divide-and-conquer strategy with using the example of the *Mergesort* algorithm.

Right now you must be thinking "That's great but what is *Mergesort*?".
Mergesort algorithm is a type of sorting algorithm that takes an unordered vector, divides the vector in smaller sized ones until it reaches a desired size and then it starts the sorting process. When all the sorting is done in the chunks we now start merging those by pairs in an ordered way. When we have those merged we reiterate the merging until we have the original vector with the modifications done. Now we have the original vector sorted in ascending order.

2 Parallelisation strategies

2.1 Task decomposition analysis for *Mergesort* with *Tareador*

To make a taste of some performance values of this program we will run multisort-seq binary defining three optional command–line arguments:

1. `-n` to define the size of the list in Kiloelements;
2. `-s` to define the size in elements of the vectors that breaks the recursions during the sort phases;
3. `-m` to define the size in elements of the vectors that breaks the recursions during the merge phases;

We just ran the binary with the default values if these arguments aren't defined and we take note of the output times to use them as reference times to check the scalability of the parallel versions in *OpenMP* we will develop.

```
par1302@boada-1:~/lab4$ ./multisort-seq -n 32768 -s 1024 -m 1024
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
*****************************************************************************************
Initialization time in seconds: 0.857916
Multisort execution time: 6.819916
Check sorted data execution time: 0.015449
Multisort program finished
*****************************************************************************************
```

Figure 1: Output of `multisort-seq` with default sizes values specified explicitly

As we can see the execution time of the sequential version is about 6.8 seconds.
Next we will investigate, using the Tareador tool, potential task decomposition strategies and their implications in terms of parallelism and task interactions required. The program we will use is *multisort-tareador.c*.

Let's see the potential parallelism that each one of the two recursive task decomposition strategies (*leaf* and *tree*) provide when applied to the sort and merge phases.

2.1.1 *Leaf* strategy

In the *leaf* strategy we have defined a task for the invocations of `basicsort()` and `basicmerge()` once the recursive divide–and–conquer decomposition stops. These changes can be seen in the file delivered multisort-tareador-leaf.c .

Then, we have obtained the task dependence graph with Tareador (Figure 2) where we can see that it generates 28 parallel tasks that are the number of calls done to `basicsort()`. After this parallel tasks we can see lots of tasks that corresponts to the different calls to `basicmerge()`.

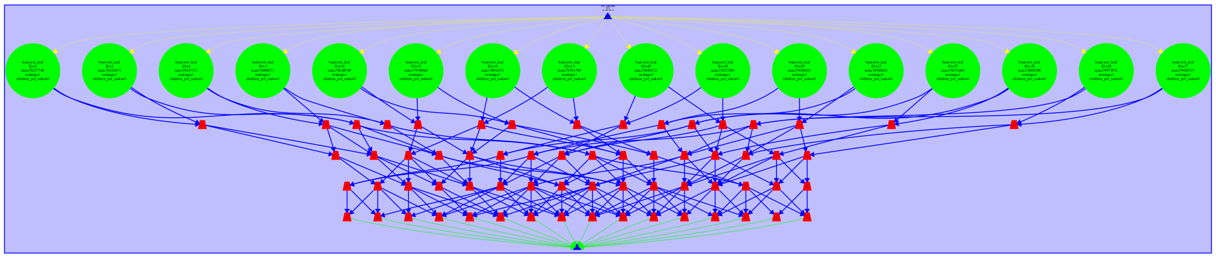


Figure 2: Task dependence graph of multisort-tareador-leaf.c, Tareador Leaf strategy

If we simulate the execution of the program with 16 processors (Figure 3), we can observe doing some zoom at the end, that the previous red tasks from Figure 2 are executed in paral·lel once the green ones that precede them have finished executing.

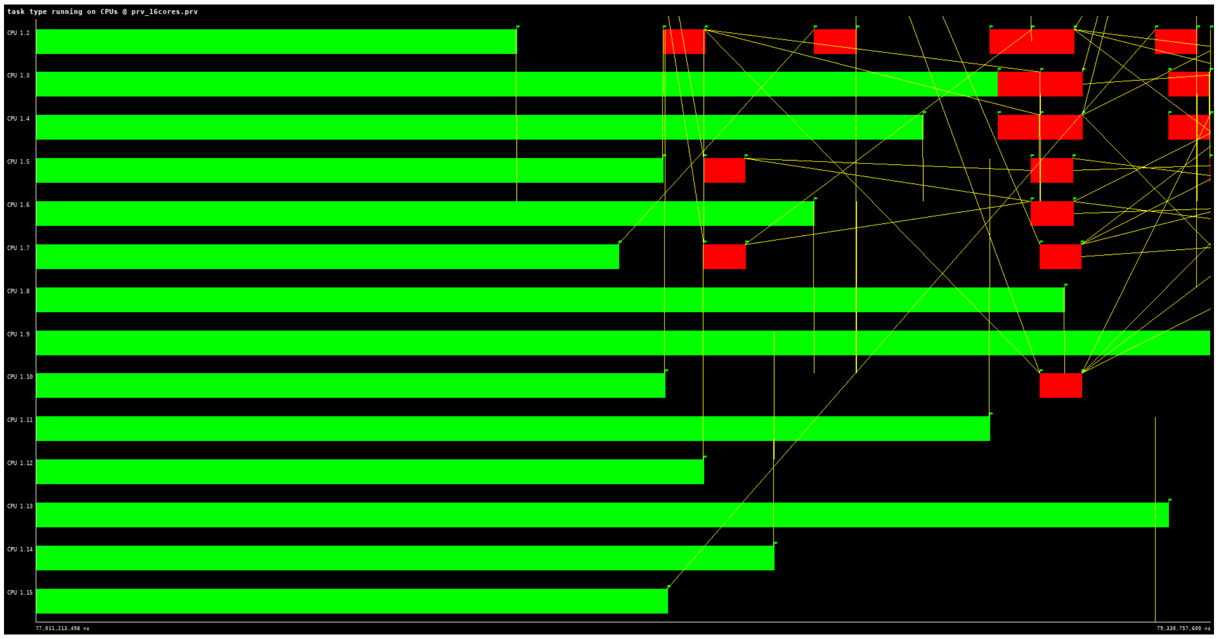


Figure 3: Zoom of the simulated execution of Leaf strategy with 16 processors

2.1.2 *Tree* strategy

In the *tree* strategy, we have defined tasks during the recursive decomposition. A task for each call to function merge() or multisort(), adding a `tareador_start_task()` and `tareador_end_task()` between the call of each function. These changes can be seen in multisort-tareador-tree.c delivered.

Analyzing the graph obtained with Tareador, we can see that the tasks are now structured in a different way. There are tasks into tasks, following a tree recursive strategy. First there are the tasks that have a big weight, those that correspond to the multisort function and then, are the others from the merge function.

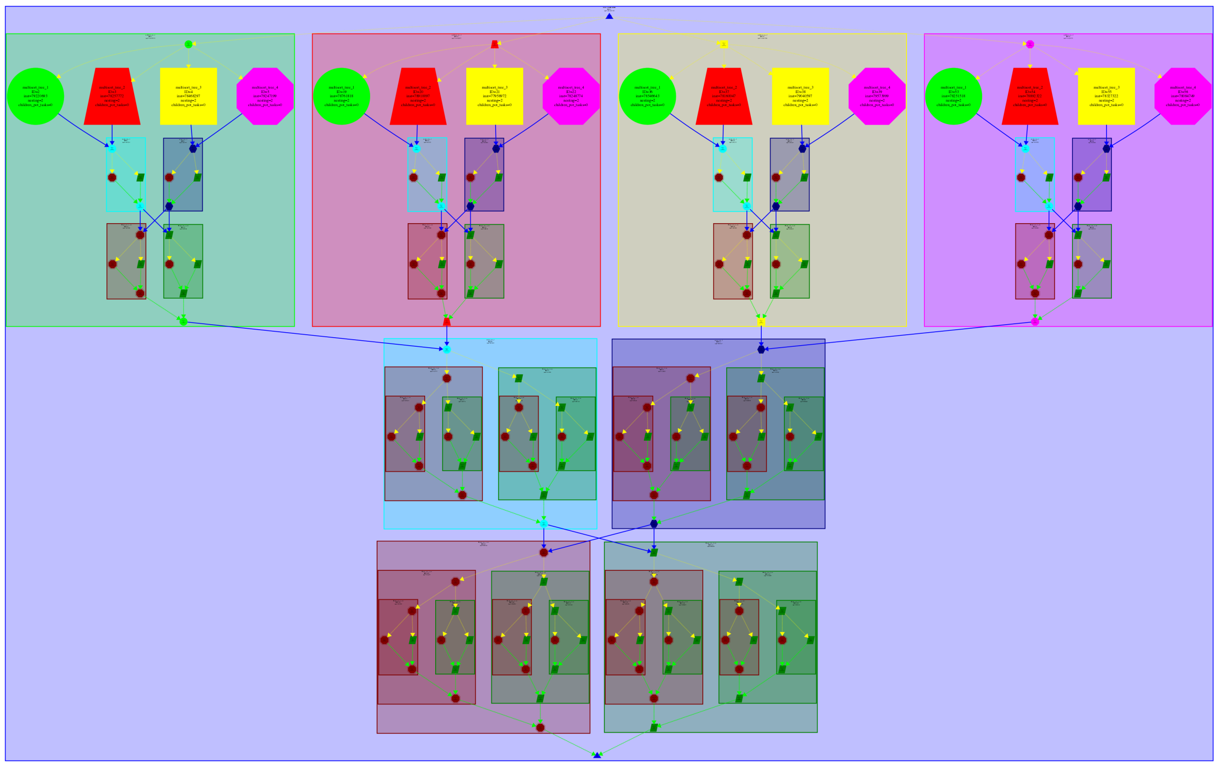


Figure 4: Task dependence graph of multisort-tareador-tree.c, Tareador Tree strategy

When it comes to the simulation with 16 processors, the biggest tasks are executed at the beginning (as the leaf strategy). If we zoom at the end, we can see all the execution of the other small tasks and in yellow their dependences.

As seen in Figure 5, there has been an improvement in the distribution of the tasks.
The tasks in deep green and deep red are from the merge calls and it is well seen that some merge functions depends on two multisort functions due to the required fragments of the vector computed. This is also seen in the final merge that needs two previously calculated merges.

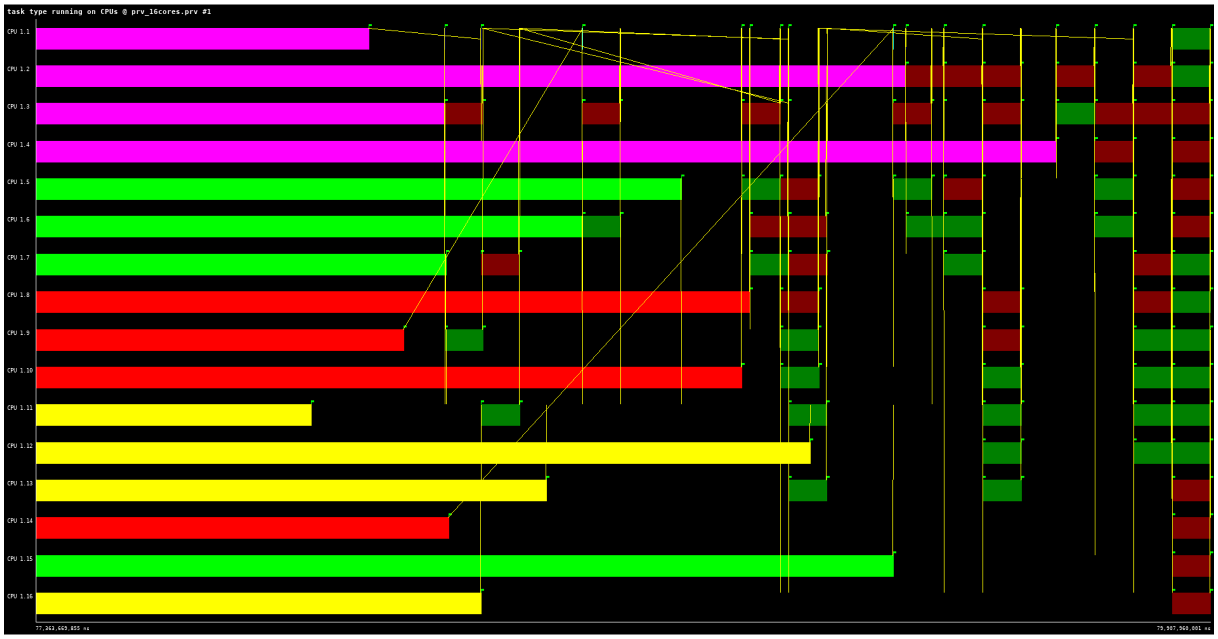


Figure 5: Zoom of the simulated execution of Tree strategy with 16 processors

3 Performance evaluation

3.1 Parallelisation and performance analysis with tasks

3.1.1 Leaf strategy in OpenMP

After adding some changes to the code we have obtained the leaf parallelisation strategy, putting a `#pramga_omp_task` before the call of basicsort and basicmerge. Also, inside multisort function we added a `#pragma_omp_taskwait` between the last call of multisort and before the merge call and another before the last call of merge. This changes can be seen in multisort-omp-leaf.c delivered.

We've checked the `submit-omp.e[...]` files and we found no errors. In to the `submit-omp.o[...]` we have seen the initialization time in seconds always around 0.85, the execution time that we have summarized below and finally we found the time of check sorted data execution that always is around 0.02 approximately. The time of initialization and data check is always the same because it's not parallelized, it's normal.

In Figure 6 we can see that there is only a thread that generates tasks; in our case thread number 3. These tasks are then executed in the rest of the threads, being the red ones executed sequentially.

As it will be shown in the next section, Leaf strategy creates fewer tasks than the tree strategy.

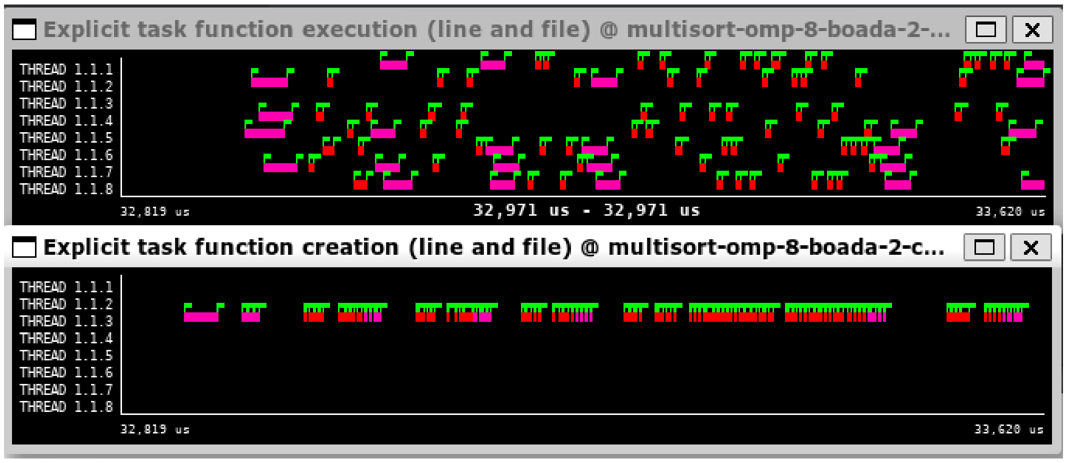
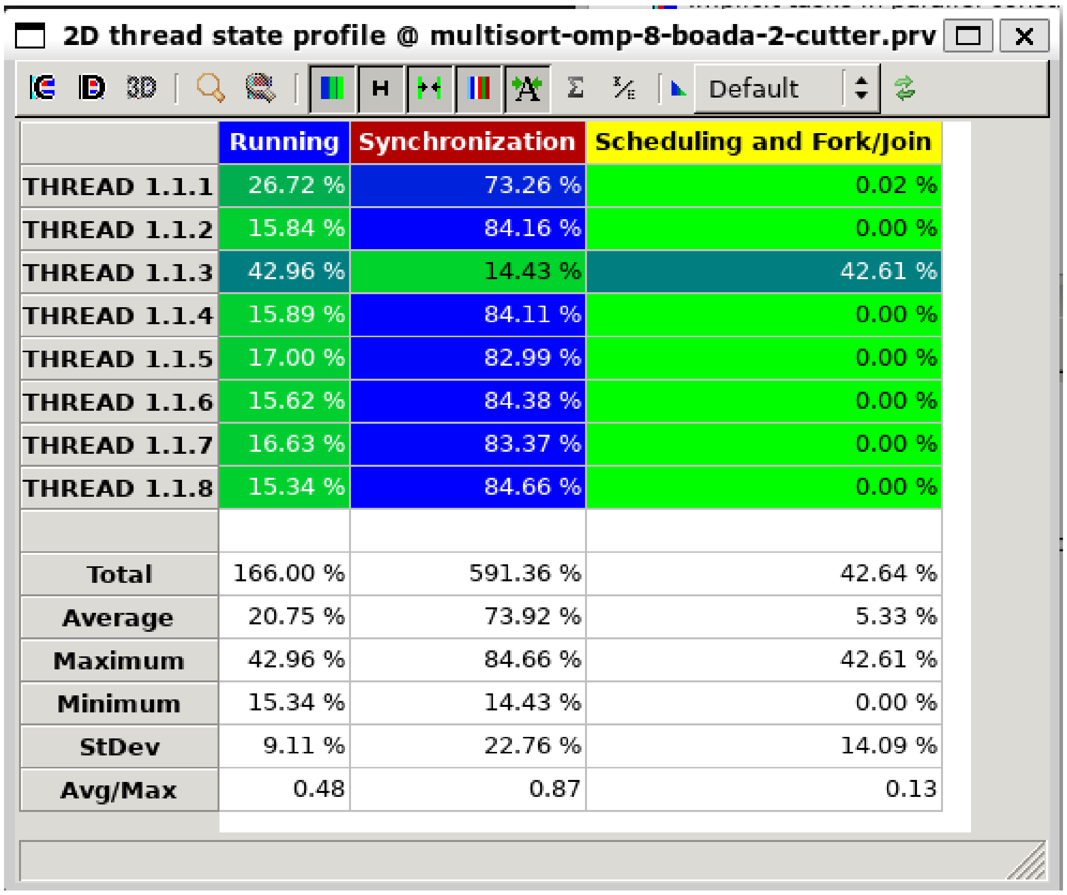


Figure 6: Explicit task execution and creation, Leaf strategy (zoomed in)

Figure 7 shows that the thread number 3 runs quite more than the other threads (43% of the overall time), being the only thread that creates the tasks (see yellow column) and consequently being the one that is less time synchronizing.



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

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	26.72 %	73.26 %	0.02 %
THREAD 1.1.2	15.84 %	84.16 %	0.00 %
THREAD 1.1.3	42.96 %	14.43 %	42.61 %
THREAD 1.1.4	15.89 %	84.11 %	0.00 %
THREAD 1.1.5	17.00 %	82.99 %	0.00 %
THREAD 1.1.6	15.62 %	84.38 %	0.00 %
THREAD 1.1.7	16.63 %	83.37 %	0.00 %
THREAD 1.1.8	15.34 %	84.66 %	0.00 %
Total	166.00 %	591.36 %	42.64 %
Average	20.75 %	73.92 %	5.33 %
Maximum	42.96 %	84.66 %	42.61 %
Minimum	15.34 %	14.43 %	0.00 %
StDev	9.11 %	22.76 %	14.09 %
Avg/Max	0.48	0.87	0.13

Figure 7: 2D state profile Leaf strategy

Number of threads	Multisort execution time
1	6.369949 s
2	3.686770 s
4	1.8976932 s

Figure 8: Execution times of `multisort-omp.c` with different numbers of threads, leaf strategy

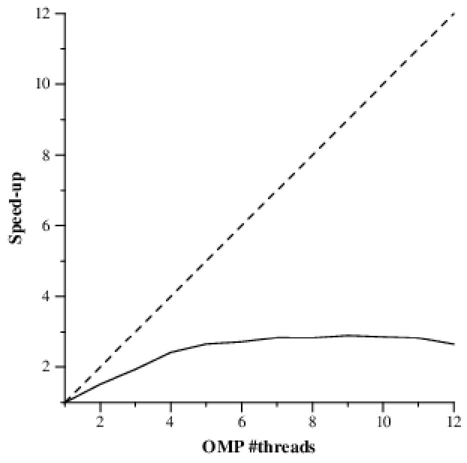
In the table above, there's the multisort execution time for 1, 2 and 4 processors. It seems that if the number of processors is increased, there's more efficiency but let's see the scalability and speed-up plots in Figure 10 for 1 to 12 processors.

For the total execution time of the program and for the multisort function only, we can see that there's an improvement between 1 and 4 processors. Then, however, adding more processors the performance will hardly improve; and even from 11 processors it will be reduced. In the next sections, different strategies will be tested and compared to this one.

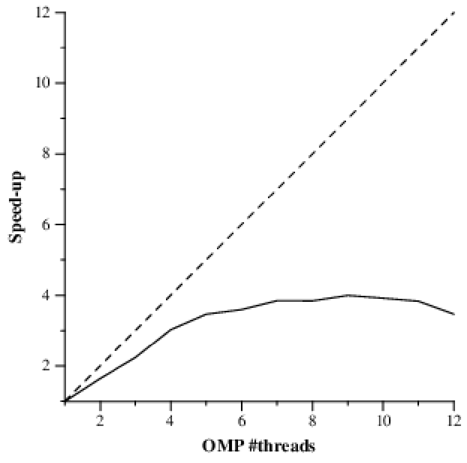
Until now, the size of the problem has been the default. For computing the modelfactors.py (Figure 10), the problem has been modified to a smaller one: -n 1024 -s 256 -m 256. As said before, we can see that there's a great scalability from 1 to 8 processors. Being the time reduced from 0.55 seconds to 0.3 s, the speed-up increased up to 1.85 and the efficiency reduced up to 0.23.

```
par1305@boada-1:~/2122q2/lab4$ grep size submit-omp.sh submit-strong-extrae.sh
submit-omp.sh:export size=32768
submit-omp.sh:export sort_size=1024
submit-omp.sh:export merge_size=1024
submit-omp.sh: ./$PROG -n $size -s $sort_size -m $merge_size >
${PROG}_${OMP_NUM_THREADS}_${HOST}.times.txt
submit-omp.sh: ./$PROG -n $size -s $sort_size -m $merge_size -c $cutoff >
${PROG}_${OMP_NUM_THREADS}_${cutoff}_${HOST}.times.txt
submit-strong-extrae.sh:export size=1024
submit-strong-extrae.sh:export sort_size=256
submit-strong-extrae.sh:export merge_size=256
submit-strong-extrae.sh:    ./$1 -n $size -s $sort_size -m $merge_size -c $cutoff
```

Figure 9: Output of `multisort` modelfactors.py size values

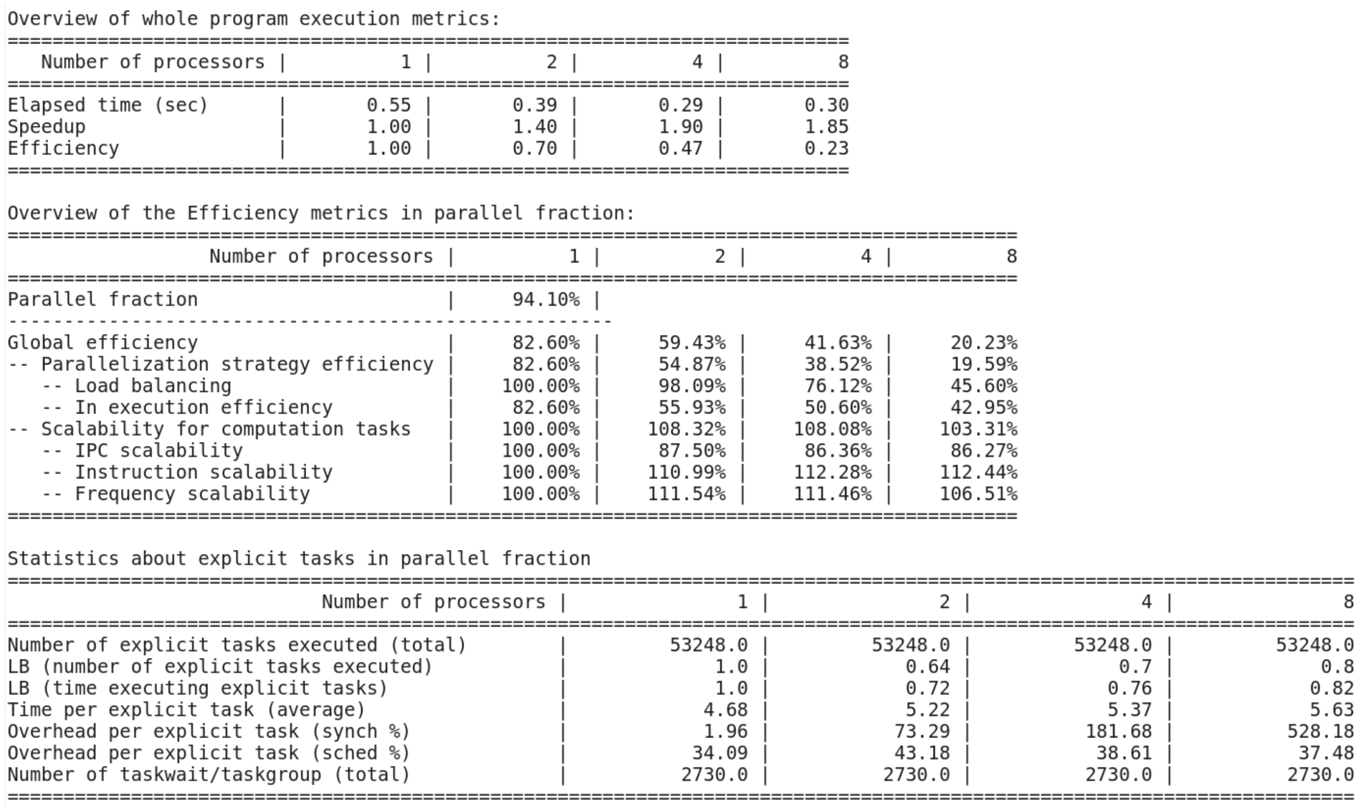


par1302
Speed-up wrt sequential time (complete application)
Fri Apr 29 11:41:49 CEST 2022



par1302
Speed-up wrt sequential time (multisort funtion only)
Fri Apr 29 11:41:49 CEST 2022

Figure 10: Scalability and speed-up plot for multisort function and the complete function, Leaf strategy



```
Overview of whole program execution metrics:
============================================================================
   Number of processors |          1 |          2 |          4 |          8
============================================================================
Elapsed time (sec)      |       0.55 |       0.39 |       0.29 |       0.30
Speedup                 |       1.00 |       1.40 |       1.90 |       1.85
Efficiency              |       1.00 |       0.70 |       0.47 |       0.23
============================================================================

Overview of the Efficiency metrics in parallel fraction:
===========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
===========================================================================================
Parallel fraction                      |     94.10% |
------------------------------------------------------
Global efficiency                      |     82.60% |     59.43% |     41.63% |     20.23%
-- Parallelization strategy efficiency |     82.60% |     54.87% |     38.52% |     19.59%
   -- Load balancing                   |    100.00% |     98.09% |     76.12% |     45.60%
   -- In execution efficiency          |     82.60% |     55.93% |     50.60% |     42.95%
-- Scalability for computation tasks   |    100.00% |    108.32% |    108.08% |    103.31%
   -- IPC scalability                  |    100.00% |     87.50% |     86.36% |     86.27%
   -- Instruction scalability          |    100.00% |    110.99% |    112.28% |    112.44%
   -- Frequency scalability            |    100.00% |    111.54% |    111.46% |    106.51%
===========================================================================================

Statistics about explicit tasks in parallel fraction
=========================================================================================================================
                                        Number of processors |                1 |                2 |                4 |                8
=========================================================================================================================
Number of explicit tasks executed (total)         |          53248.0 |          53248.0 |          53248.0 |          53248.0
LB (number of explicit tasks executed)            |              1.0 |             0.64 |              0.7 |              0.8
LB (time executing explicit tasks)                |              1.0 |             0.72 |             0.76 |             0.82
Time per explicit task (average)                  |             4.68 |             5.22 |             5.37 |             5.63
Overhead per explicit task (synch %)              |             1.96 |            73.29 |           181.68 |           528.18
Overhead per explicit task (sched %)              |            34.09 |            43.18 |            38.61 |            37.48
Number of taskwait/taskgroup (total)              |           2730.0 |           2730.0 |           2730.0 |           2730.0
=========================================================================================================================
```

Figure 11: Data extracted from modelfactors.out, Leaf strategy

3.1.2 Tree strategy in OpenMP

We've applied some changes to the previous code for achieving the tree strategy. We put a `#pragma omp task` for each call to the multisort function and merge function. In multisort, now there are two call regions with a `#pragma omp taskgroup`, to ensure that all threads have ended before starting the merge calls, and all threads have ended before starting the last call to merge. This changes can be seen in multisort-omp-tree.c .

In Figure 12 we can see that there are more tasks than the leaf strategy. There are threads generating tasks and executing them. When a thread stops running a task, almost immediately starts running other tasks or generates others.

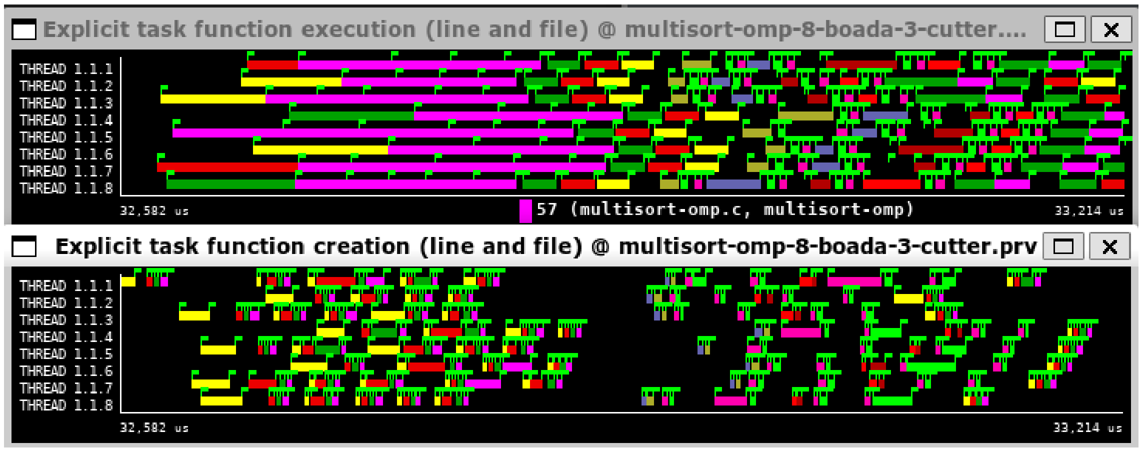
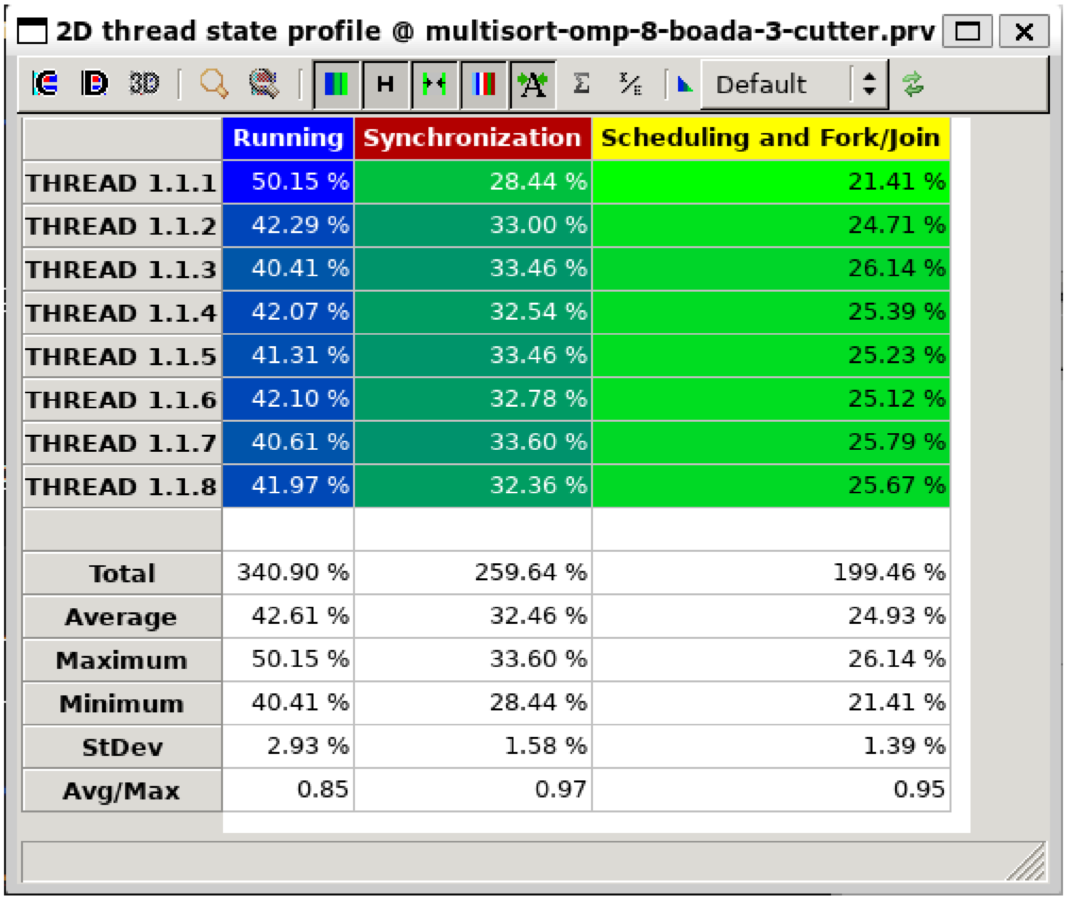


Figure 12: Explicit task execution and creation, Tree strategy (zoomed in)

Now, for this strategy, all threads have their percentage of Scheduling and Fork/Join. Synchronization and running is more balanced except thread number 1 that has the running percentage a little increased and the synchronization a bit decreased compared to the others (see Figure 13).


2D thread state profile @ multisort-omp-8-boada-3-cutter.prv
Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	50.15 %	28.44 %	21.41 %
THREAD 1.1.2	42.29 %	33.00 %	24.71 %
THREAD 1.1.3	40.41 %	33.46 %	26.14 %
THREAD 1.1.4	42.07 %	32.54 %	25.39 %
THREAD 1.1.5	41.31 %	33.46 %	25.23 %
THREAD 1.1.6	42.10 %	32.78 %	25.12 %
THREAD 1.1.7	40.61 %	33.60 %	25.79 %
THREAD 1.1.8	41.97 %	32.36 %	25.67 %
Total	340.90 %	259.64 %	199.46 %
Average	42.61 %	32.46 %	24.93 %
Maximum	50.15 %	33.60 %	26.14 %
Minimum	40.41 %	28.44 %	21.41 %
StDev	2.93 %	1.58 %	1.39 %
Avg/Max	0.85	0.97	0.95

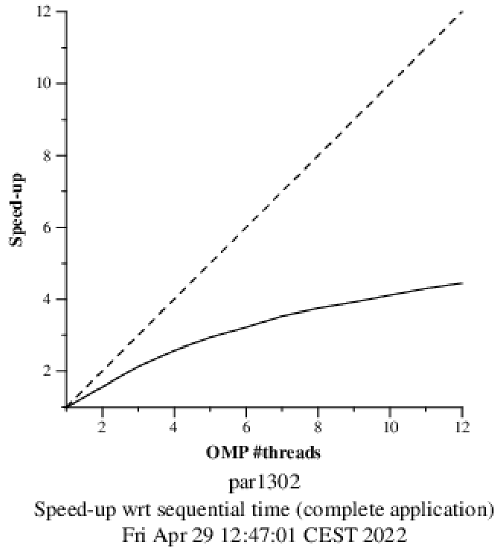
Figure 13: 2D state profile Tree strategy

Number of threads	Multisort execution time
1	6.398295 s
2	3.696333 s
4	1.84861 s

Figure 14: Executions times of `multisort-omp.c` with different numbers of threads, tree strategy

In the table above, we can find that the execution time of multisrot is quite similar to the leaf strategy, but next graphs show that the speed up up to 12 processors are a lot better compared to the leaf strategy, specially for the multisort function speed-up.

Modelfactors (problem size reduced) shows that speed up has reached up to 3.79 for 8 processors and a time reduced to 0.22 seconds. Tree Strategy has a better performance compared to Leaf Strategy.



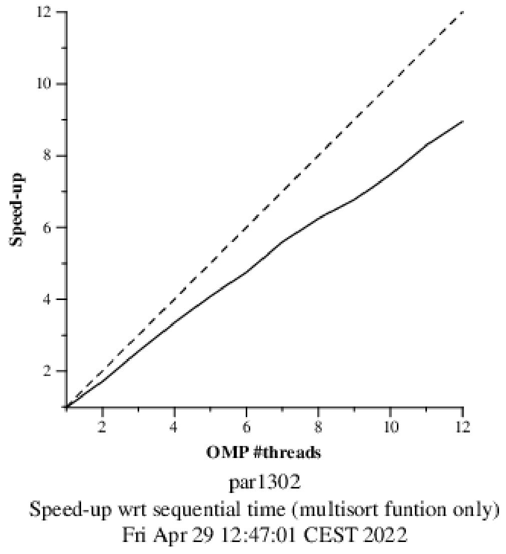
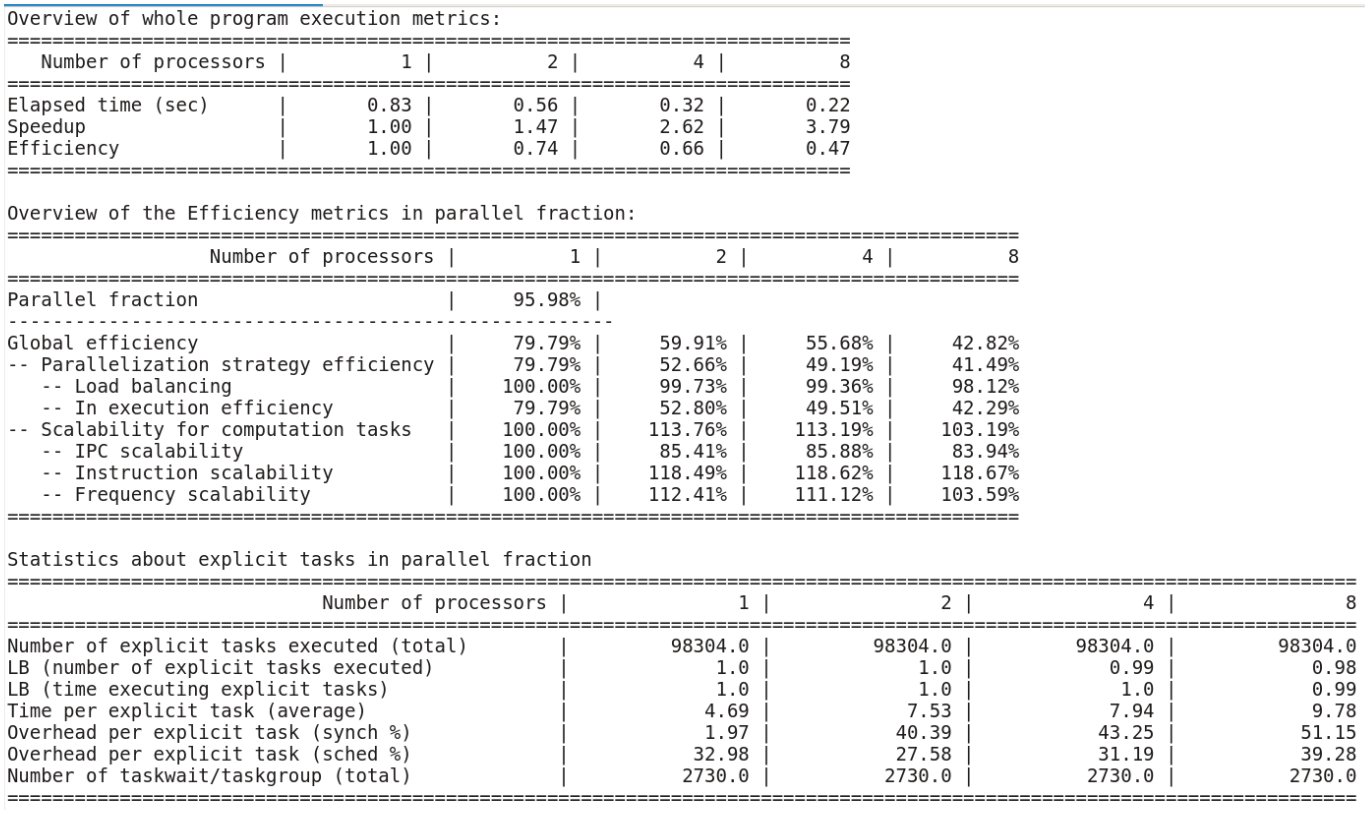


Figure 15: Scalability and speed-up plot for multisort function and the complete function, Tree strategy



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.83 |       0.56 |       0.32 |       0.22
Speedup                 |       1.00 |       1.47 |       2.62 |       3.79
Efficiency              |       1.00 |       0.74 |       0.66 |       0.47
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     95.98% |
------------------------------------------------------
Global efficiency                      |     79.79% |     59.91% |     55.68% |     42.82%
-- Parallelization strategy efficiency |     79.79% |     52.66% |     49.19% |     41.49%
   -- Load balancing                   |    100.00% |     99.73% |     99.36% |     98.12%
   -- In execution efficiency          |     79.79% |     52.80% |     49.51% |     42.29%
-- Scalability for computation tasks   |    100.00% |    113.76% |    113.19% |    103.19%
   -- IPC scalability                  |    100.00% |     85.41% |     85.88% |     83.94%
   -- Instruction scalability          |    100.00% |    118.49% |    118.62% |    118.67%
   -- Frequency scalability            |    100.00% |    112.41% |    111.12% |    103.59%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                1 |                2 |                4 |                8
========================================================================================================================
Number of explicit tasks executed (total)        |          98304.0 |          98304.0 |          98304.0 |          98304.0
LB (number of explicit tasks executed)           |              1.0 |              1.0 |             0.99 |             0.98
LB (time executing explicit tasks)               |              1.0 |              1.0 |              1.0 |             0.99
Time per explicit task (average)                 |             4.69 |             7.53 |             7.94 |             9.78
Overhead per explicit task (synch %)             |             1.97 |            40.39 |            43.25 |            51.15
Overhead per explicit task (sched %)             |            32.98 |            27.58 |            31.19 |            39.28
Number of taskwait/taskgroup (total)             |           2730.0 |           2730.0 |           2730.0 |           2730.0
========================================================================================================================
```

Figure 16: Data extracted from modelfactors.out, Tree strategy

3.2 Controlling task granularities: cut–off mechanism

Through the cut-off mechanism, we avoid generating overhead costs when performing mergesort.

For the implementation of cut-off, we used the final clause. This clause specifies the depth of recursive level so that we can manage the recursiveness. So, to know the depth level we have added an extra parameter to the multisort and merge functions calls. We initialized this parameter to zero and we added one in each recursive call to control the depth.

When the desired depth is reached, we cover the functions with the condition $if(!ompinfinale())$. In order to activate the final flag, we have changed the *#pragma omp* call adding *taskfinal(depth>=CUTOFF)*.

As a result of this code and using different values for the variable CUTOFF, we have obtained the graphic of the Figure 17. We can see that the ideal size of cut-off is 5 where we observe the minimum execution time.

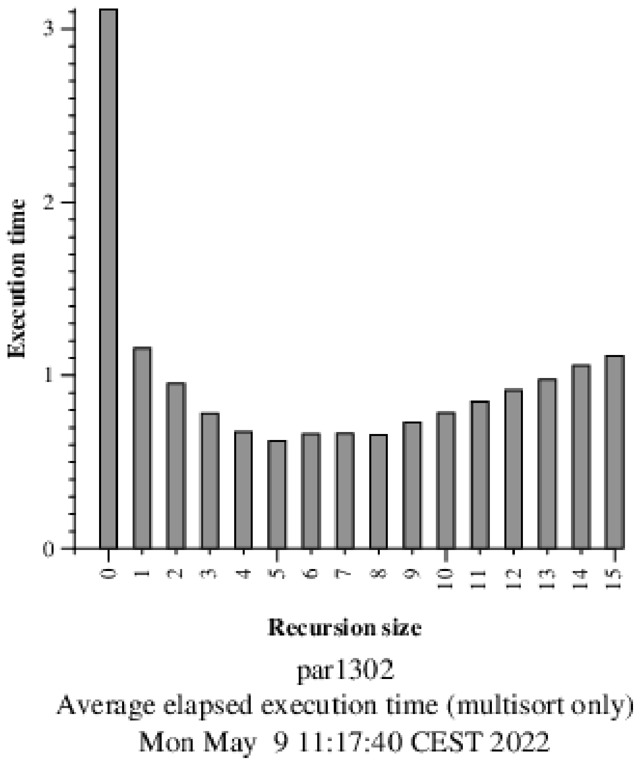


Figure 17: Graphic of time execution depending on CUTOFF value

To check the correctness of our code we executed the zero cut-off size with 8 processors. When we analized the 0 cut-off case, we saw that only 7 tasks were created. Those tasks corresponds to the seven first tasks created by the 4 mutlisort function calls and the 3 merge calls. We didn't observe more tasks because of the final clause: it stops the recursion as we said before.

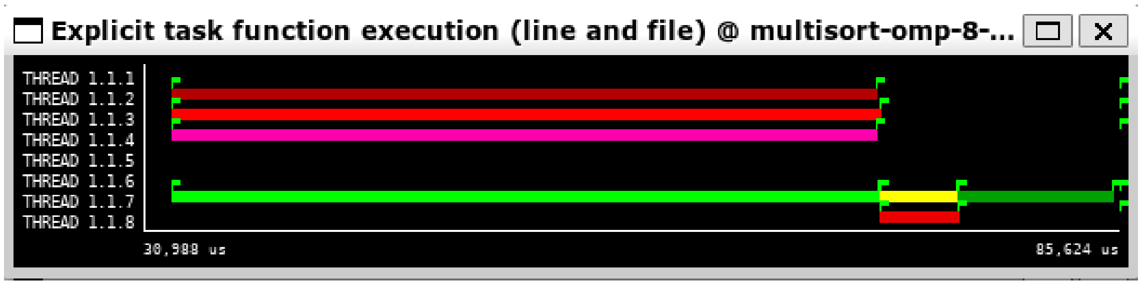


Figure 18: Explicit task execution and creation, Tree strategy with cut-off=0 (zoomed in)

Following this line of work, we checked the program with cut-off=1 and cut-off=8 too. We obtained the Figure 19 and 20. This time we can see the creation of more tasks related with the recursiveness calls.

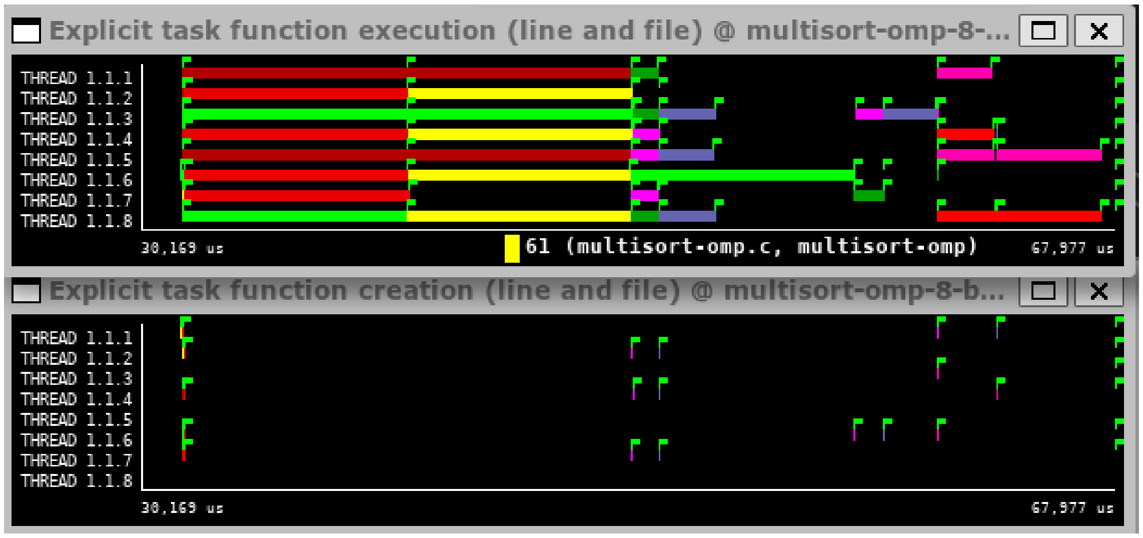


Figure 19: Explicit task execution and creation, Tree strategy with cut-off=1 (zoomed in)

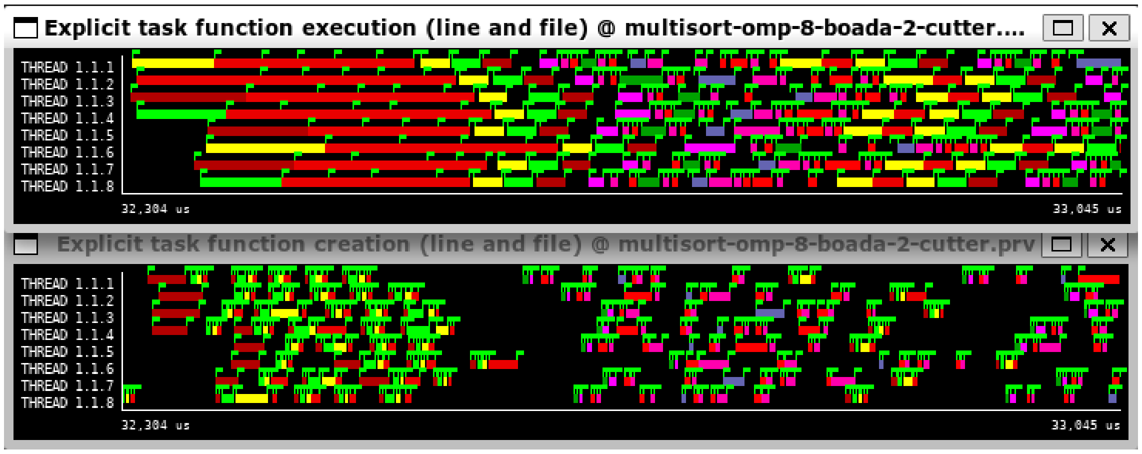
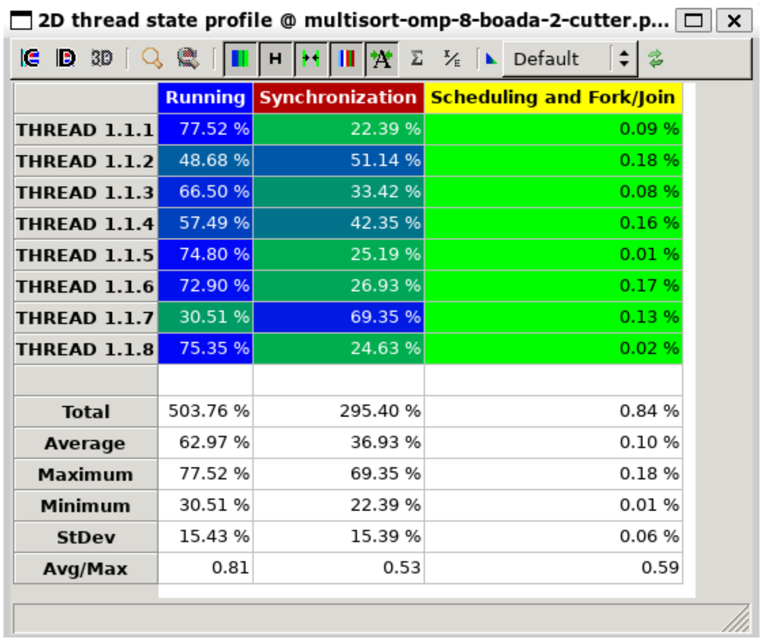

Explicit task function execution (line and file) @ multisort-omp-8-boada-2-cutter....
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
32,304 us
33,045 us
Explicit task function creation (line and file) @ multisort-omp-8-boada-2-cutter.prv
32,304 us
33,045 us

Figure 20: Explicit task execution and creation, Tree strategy with cut-off=8 (zoomed in)

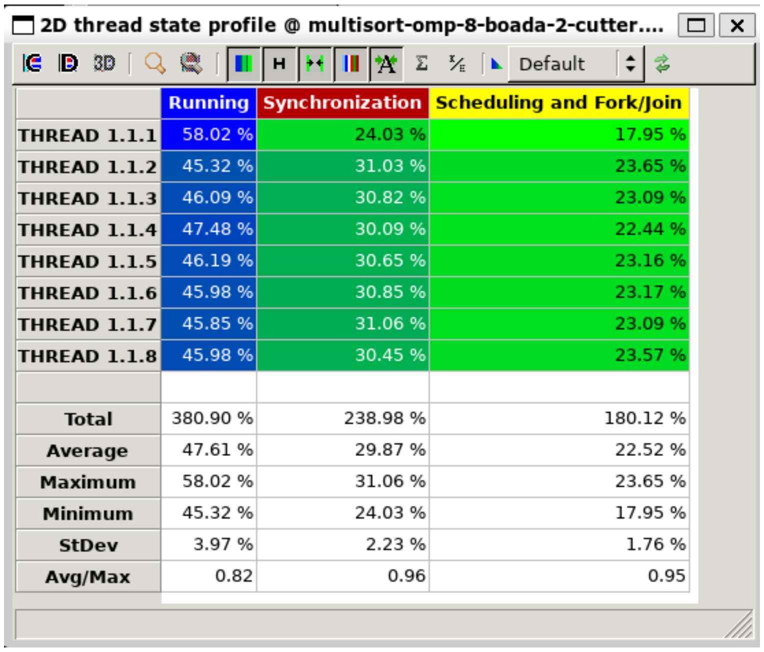
It's hard to read and to interpret the data of the previous graphs so we decided to draw the following tables to better analyze what was happening (Figures 21 and 22).

What we have seen is that for cut-off=8, the running time is quite the same except for thread number 1, that it has it a bit higher, having the synchronisation and Scheduling and Fork/Join percentages a bit lower. However, with cut-off=1 there are three threads that run a little less but with more percentage of synchronization. The main difference between each two are that with cut-off=1, the scheduling and fork/join time are less than with 8.


2D thread state profile @ multisort-omp-8-boada-2-cutter.p...
Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	77.52 %	22.39 %	0.09 %
THREAD 1.1.2	48.68 %	51.14 %	0.18 %
THREAD 1.1.3	66.50 %	33.42 %	0.08 %
THREAD 1.1.4	57.49 %	42.35 %	0.16 %
THREAD 1.1.5	74.80 %	25.19 %	0.01 %
THREAD 1.1.6	72.90 %	26.93 %	0.17 %
THREAD 1.1.7	30.51 %	69.35 %	0.13 %
THREAD 1.1.8	75.35 %	24.63 %	0.02 %
Total	503.76 %	295.40 %	0.84 %
Average	62.97 %	36.93 %	0.10 %
Maximum	77.52 %	69.35 %	0.18 %
Minimum	30.51 %	22.39 %	0.01 %
StDev	15.43 %	15.39 %	0.06 %
Avg/Max	0.81	0.53	0.59

Figure 21: 2D state profile Tree strategy with cut-off=1

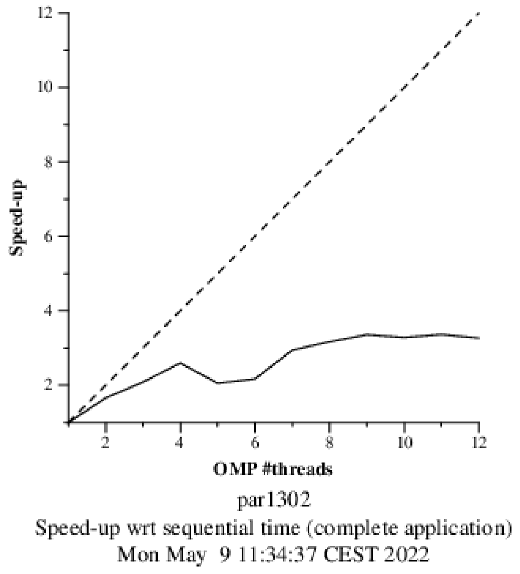

2D thread state profile @ multisort-omp-8-boada-2-cutter....
Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	58.02 %	24.03 %	17.95 %
THREAD 1.1.2	45.32 %	31.03 %	23.65 %
THREAD 1.1.3	46.09 %	30.82 %	23.09 %
THREAD 1.1.4	47.48 %	30.09 %	22.44 %
THREAD 1.1.5	46.19 %	30.65 %	23.16 %
THREAD 1.1.6	45.98 %	30.85 %	23.17 %
THREAD 1.1.7	45.85 %	31.06 %	23.09 %
THREAD 1.1.8	45.98 %	30.45 %	23.57 %
Total	380.90 %	238.98 %	180.12 %
Average	47.61 %	29.87 %	22.52 %
Maximum	58.02 %	31.06 %	23.65 %
Minimum	45.32 %	24.03 %	17.95 %
StDev	3.97 %	2.23 %	1.76 %
Avg/Max	0.82	0.96	0.95

Figure 22: 2D state profile Tree strategy with cut-off=8

As for scalability and speed-up plots for the tree-strategy with the cut-off mechanism, we've computed which would be the speed-up depending on the number of threads and for the cut-off values of 1, 4 and 8.

For cut-off=1, the speed-up is not so good, but with a value of 8, the speed-up in the complete application is better than in the tree strategy because we do not lose efficiency in some little tasks that could have been executed directly by the thread instead of creating tasks and losing time with the synchronization.



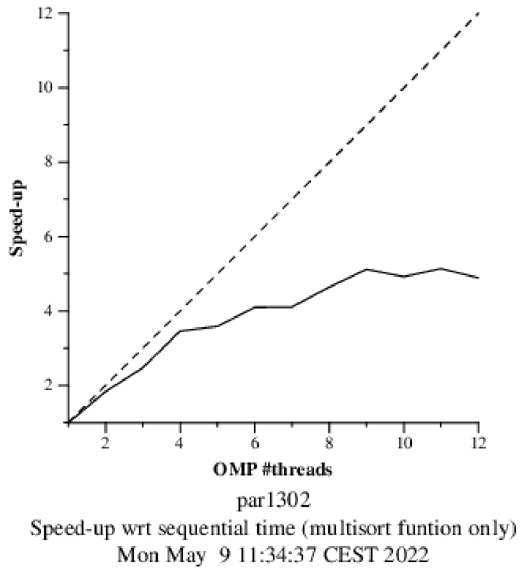
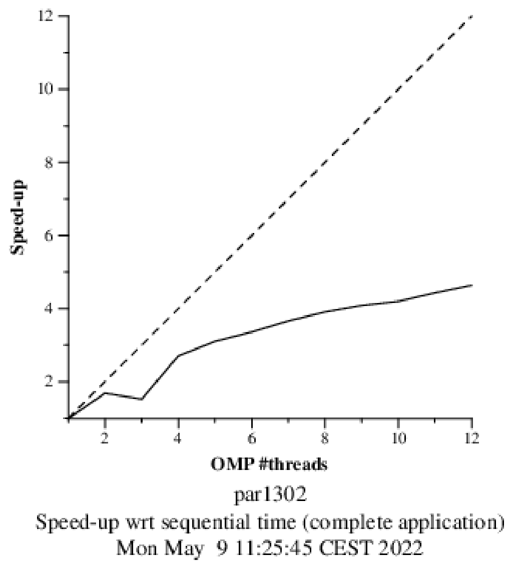


Figure 23: Scalability and speed-up plot for multisort function and the complete function, Cut-off = 1



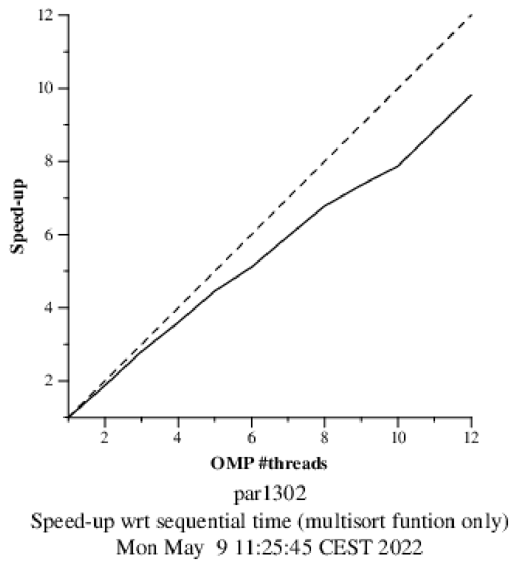
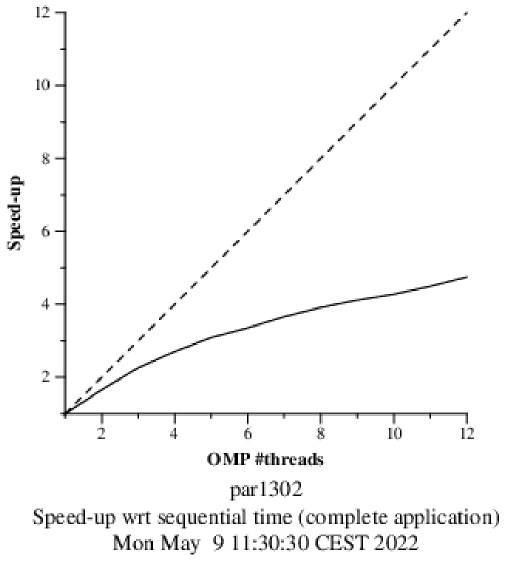


Figure 24: Scalability and speed-up plot for multisort function and the complete function, Cut-off = 4



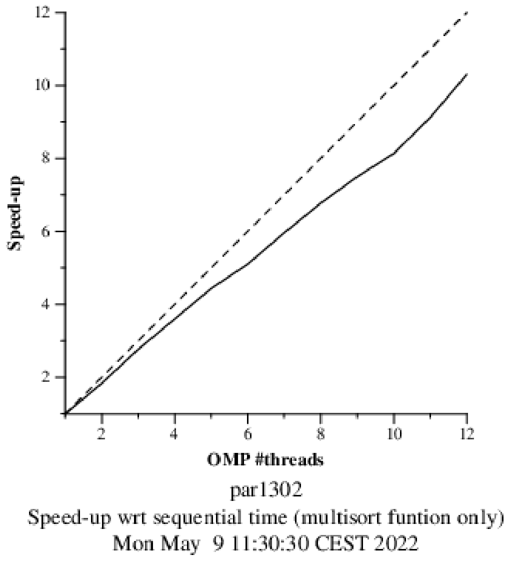
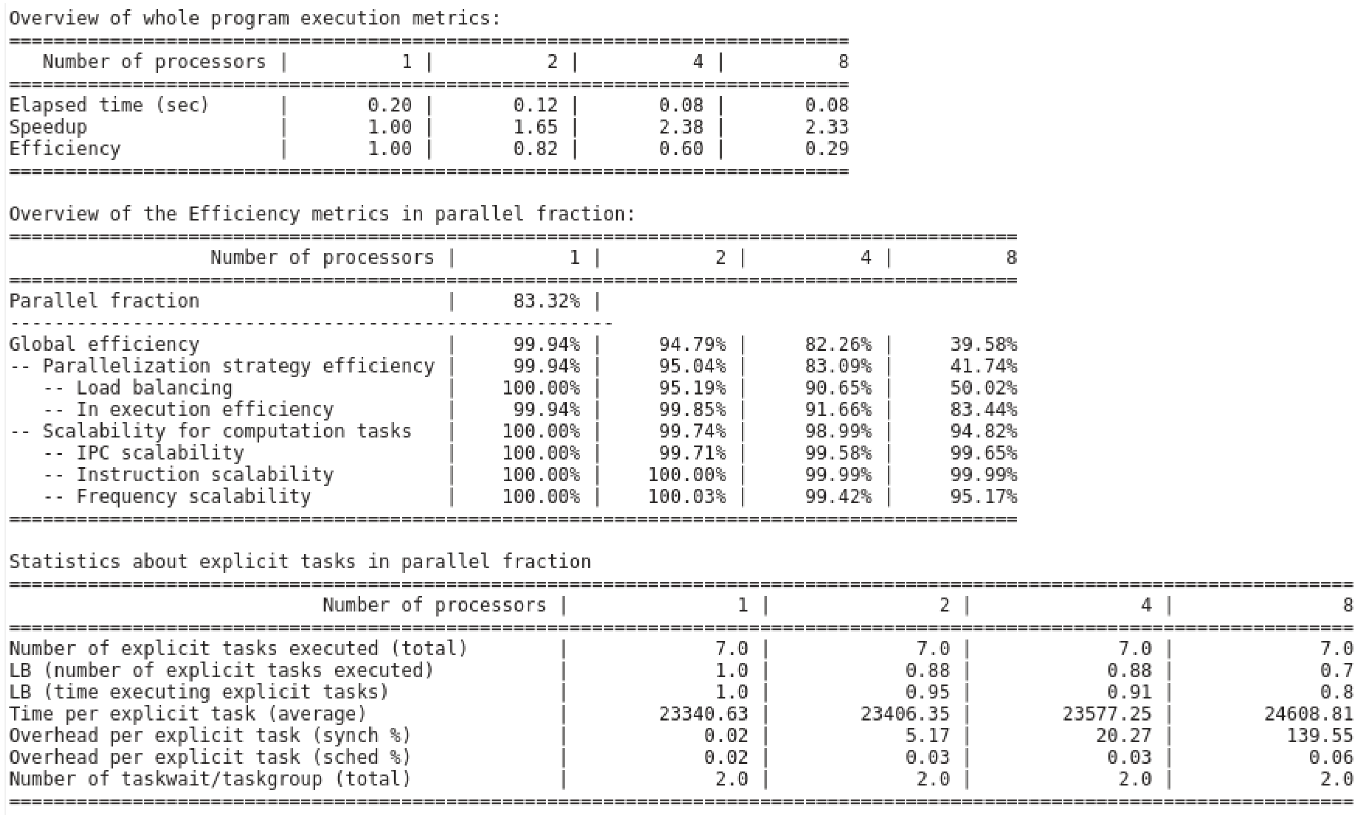


Figure 25: Scalability and speed-up plot for multisort function and the complete function, Cut-off = 8

Now, we provide the evolution of modelfactors.out for different values of cut-off: 0, 1, 2, 4 and 8.

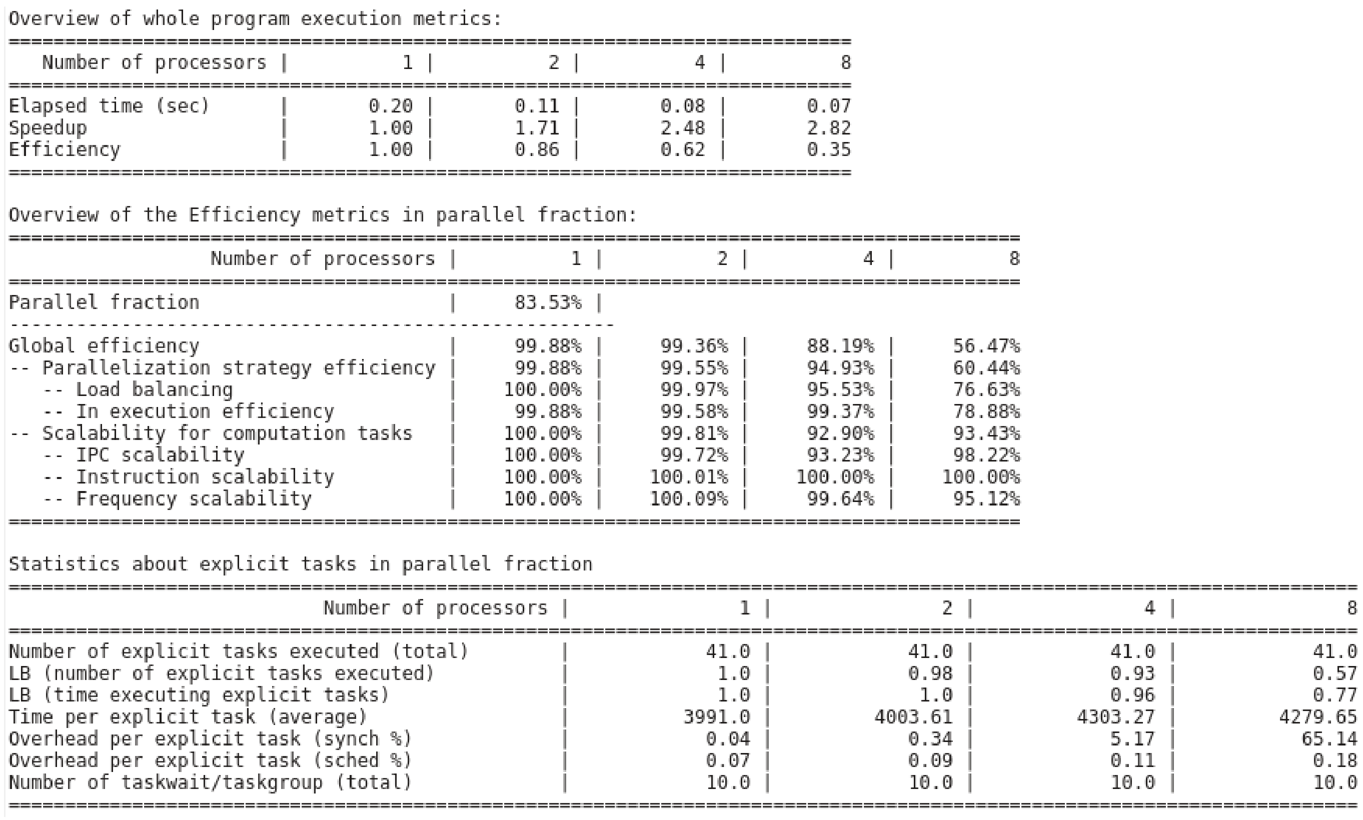
We can observe that with 8 processors and cut-off=8, the speed-up is almost 4 and the execution time is reduced to 0.14s. If we opt for 4 processors, a value of cut-off=4 could be a little better than with cut-off=8.

We can observe also that, obviously, if we increase the cut-off value, the number of tasks executed are increased. With cut-off=0, the number of explicit tasks executed is 7, then, with cut-off=1, 41, cut-off=2, 189, and and so achieving the 57173 tasks with cut-off=8. In cut-off=8 the parallel fraction increases from 85% to 94%.



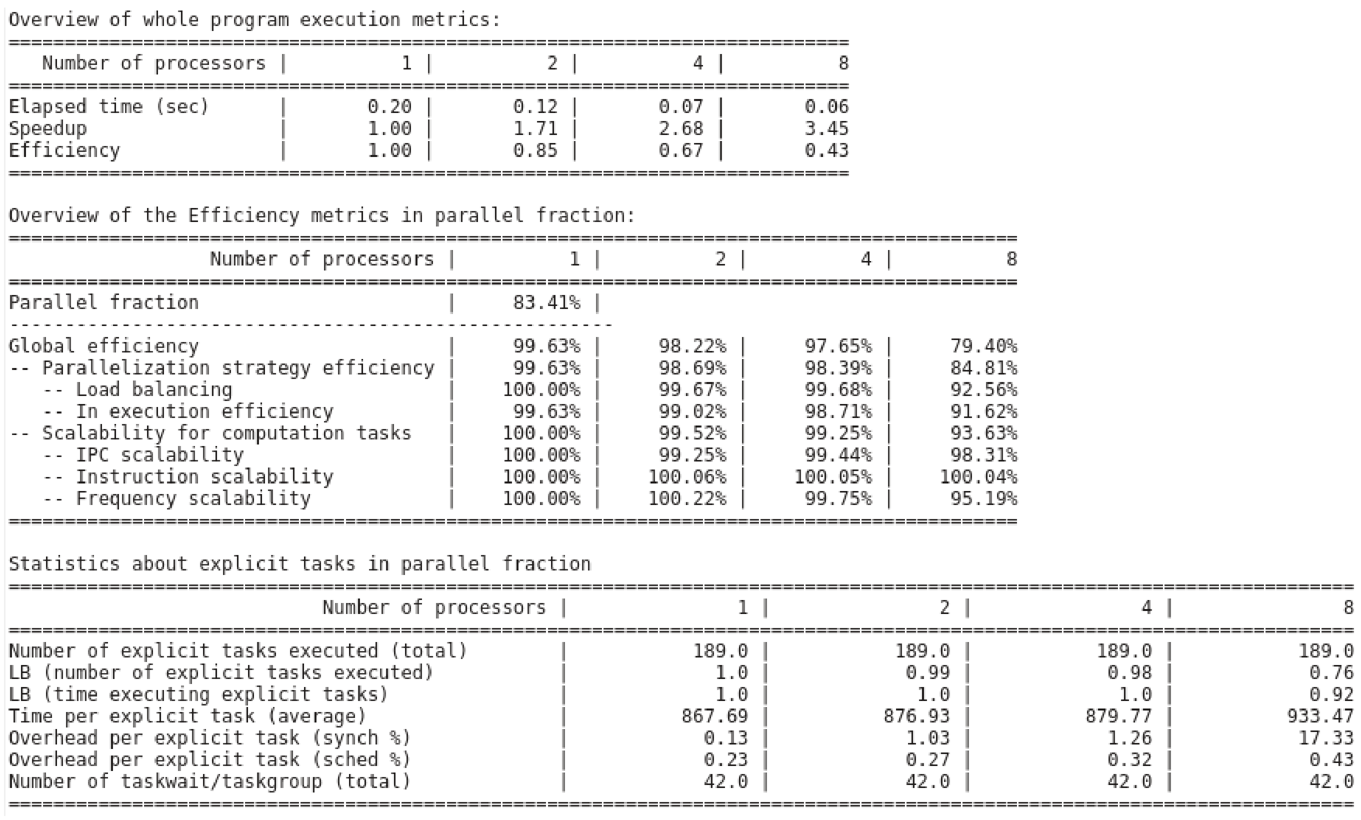
```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.20 |       0.12 |       0.08 |       0.08
Speedup                 |       1.00 |       1.65 |       2.38 |       2.33
Efficiency              |       1.00 |       0.82 |       0.60 |       0.29
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     83.32% |
------------------------------------------------------
Global efficiency                      |     99.94% |     94.79% |     82.26% |     39.58%
-- Parallelization strategy efficiency |     99.94% |     95.04% |     83.09% |     41.74%
   -- Load balancing                   |    100.00% |     95.19% |     90.65% |     50.02%
   -- In execution efficiency          |     99.94% |     99.85% |     91.66% |     83.44%
-- Scalability for computation tasks   |    100.00% |     99.74% |     98.99% |     94.82%
   -- IPC scalability                  |    100.00% |     99.71% |     99.58% |     99.65%
   -- Instruction scalability          |    100.00% |    100.00% |     99.99% |     99.99%
   -- Frequency scalability            |    100.00% |    100.03% |     99.42% |     95.17%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                1 |                2 |                4 |                8
========================================================================================================================
Number of explicit tasks executed (total)        |              7.0 |              7.0 |              7.0 |              7.0
LB (number of explicit tasks executed)           |              1.0 |             0.88 |             0.88 |              0.7
LB (time executing explicit tasks)               |              1.0 |             0.95 |             0.91 |              0.8
Time per explicit task (average)                 |         23340.63 |         23406.35 |         23577.25 |         24608.81
Overhead per explicit task (synch %)             |             0.02 |             5.17 |            20.27 |           139.55
Overhead per explicit task (sched %)             |             0.02 |             0.03 |             0.03 |             0.06
Number of taskwait/taskgroup (total)             |              2.0 |              2.0 |              2.0 |              2.0
========================================================================================================================
```

Figure 26: Data extracted from modelfactors.out, Cut-off = 0



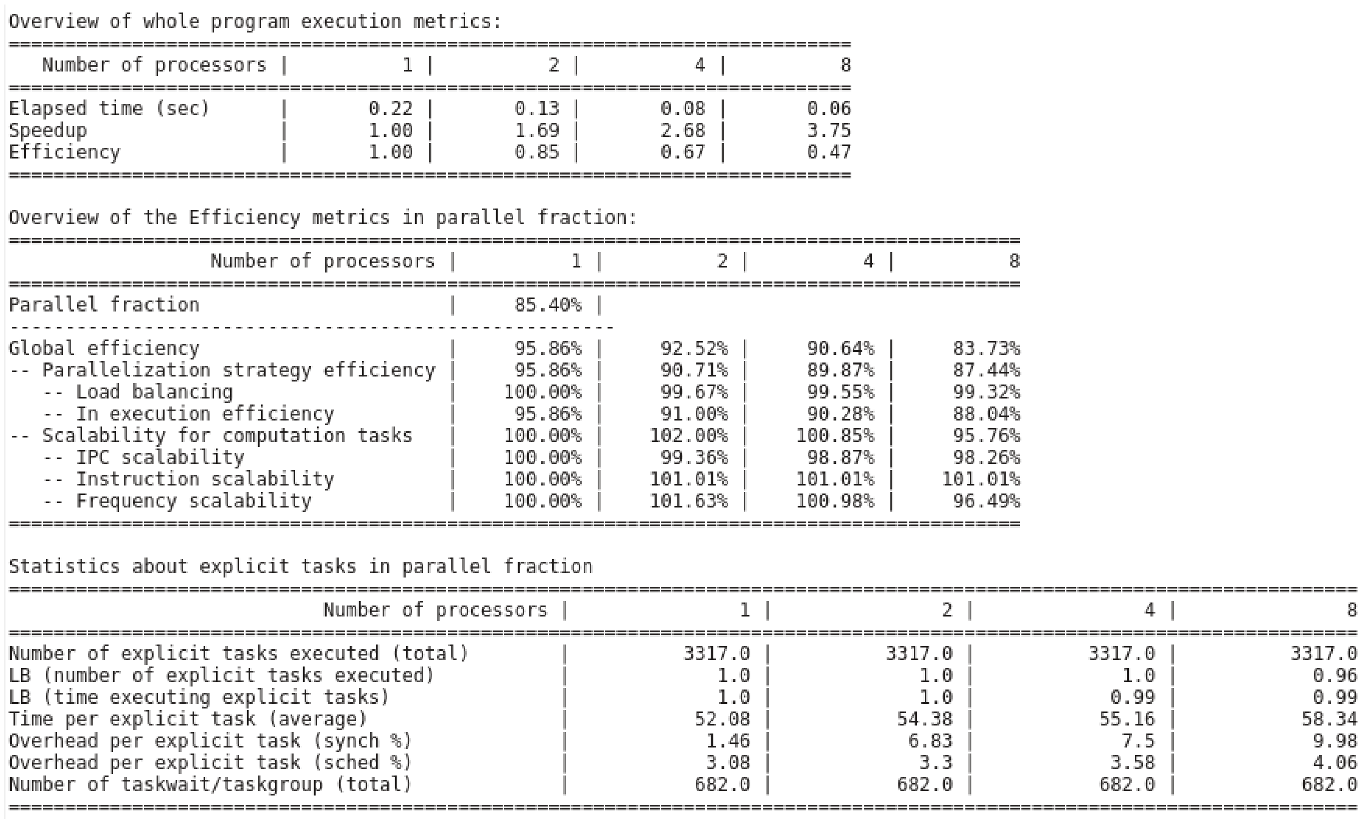
```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.20 |       0.11 |       0.08 |       0.07
Speedup                 |       1.00 |       1.71 |       2.48 |       2.82
Efficiency              |       1.00 |       0.86 |       0.62 |       0.35
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     83.53% |
------------------------------------------------------
Global efficiency                      |     99.88% |     99.36% |     88.19% |     56.47%
-- Parallelization strategy efficiency |     99.88% |     99.55% |     94.93% |     60.44%
   -- Load balancing                   |    100.00% |     99.97% |     95.53% |     76.63%
   -- In execution efficiency          |     99.88% |     99.58% |     99.37% |     78.88%
-- Scalability for computation tasks   |    100.00% |     99.81% |     92.90% |     93.43%
   -- IPC scalability                  |    100.00% |     99.72% |     93.23% |     98.22%
   -- Instruction scalability          |    100.00% |    100.01% |    100.00% |    100.00%
   -- Frequency scalability            |    100.00% |    100.09% |     99.64% |     95.12%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                1 |                2 |                4 |                8
========================================================================================================================
Number of explicit tasks executed (total)        |             41.0 |             41.0 |             41.0 |             41.0
LB (number of explicit tasks executed)           |              1.0 |             0.98 |             0.93 |             0.57
LB (time executing explicit tasks)               |              1.0 |              1.0 |             0.96 |             0.77
Time per explicit task (average)                 |           3991.0 |          4003.61 |          4303.27 |          4279.65
Overhead per explicit task (synch %)             |             0.04 |             0.34 |             5.17 |            65.14
Overhead per explicit task (sched %)             |             0.07 |             0.09 |             0.11 |             0.18
Number of taskwait/taskgroup (total)             |             10.0 |             10.0 |             10.0 |             10.0
========================================================================================================================
```

Figure 27: Data extracted from modelfactors.out, Cut-off = 1



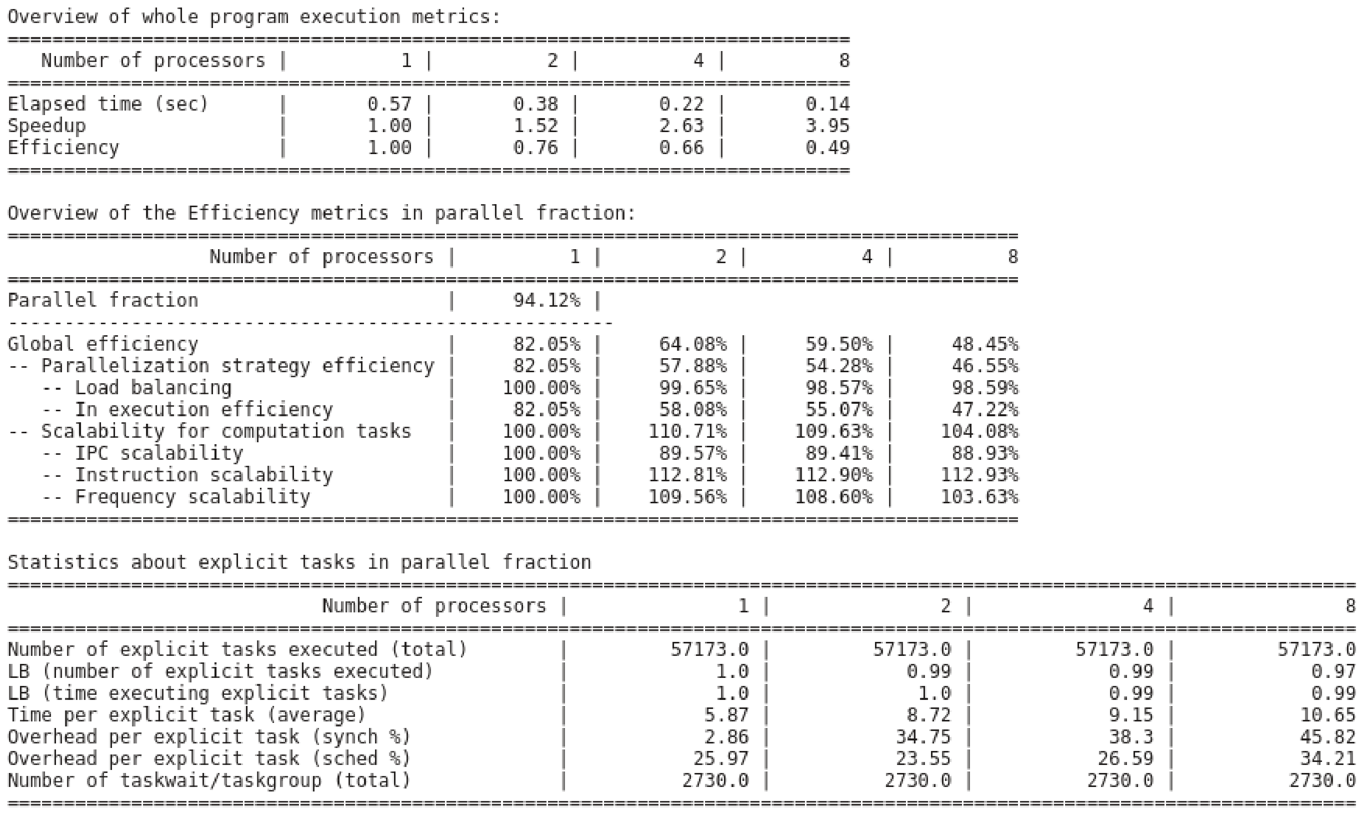
```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.20 |       0.12 |       0.07 |       0.06
Speedup                 |       1.00 |       1.71 |       2.68 |       3.45
Efficiency              |       1.00 |       0.85 |       0.67 |       0.43
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     83.41% |
------------------------------------------------------
Global efficiency                      |     99.63% |     98.22% |     97.65% |     79.40%
-- Parallelization strategy efficiency |     99.63% |     98.69% |     98.39% |     84.81%
   -- Load balancing                   |    100.00% |     99.67% |     99.68% |     92.56%
   -- In execution efficiency          |     99.63% |     99.02% |     98.71% |     91.62%
-- Scalability for computation tasks   |    100.00% |     99.52% |     99.25% |     93.63%
   -- IPC scalability                  |    100.00% |     99.25% |     99.44% |     98.31%
   -- Instruction scalability          |    100.00% |    100.06% |    100.05% |    100.04%
   -- Frequency scalability            |    100.00% |    100.22% |     99.75% |     95.19%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                           Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)       |             189.0 |             189.0 |             189.0 |             189.0
LB (number of explicit tasks executed)          |               1.0 |              0.99 |              0.98 |              0.76
LB (time executing explicit tasks)              |               1.0 |               1.0 |               1.0 |              0.92
Time per explicit task (average)                |            867.69 |            876.93 |            879.77 |            933.47
Overhead per explicit task (synch %)            |              0.13 |              1.03 |              1.26 |             17.33
Overhead per explicit task (sched %)            |              0.23 |              0.27 |              0.32 |              0.43
Number of taskwait/taskgroup (total)            |              42.0 |              42.0 |              42.0 |              42.0
========================================================================================================================
```

Figure 28: Data extracted from modelfactors.out, Cut-off = 2



```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |         1 |         2 |         4 |         8
===========================================================================
Elapsed time (sec)      |      0.22 |      0.13 |      0.08 |      0.06
Speedup                 |      1.00 |      1.69 |      2.68 |      3.75
Efficiency              |      1.00 |      0.85 |      0.67 |      0.47
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |         1 |         2 |         4 |         8
==========================================================================================
Parallel fraction                      |    85.40% |
------------------------------------------------------
Global efficiency                      |    95.86% |    92.52% |    90.64% |    83.73%
-- Parallelization strategy efficiency |    95.86% |    90.71% |    89.87% |    87.44%
   -- Load balancing                   |   100.00% |    99.67% |    99.55% |    99.32%
   -- In execution efficiency          |    95.86% |    91.00% |    90.28% |    88.04%
-- Scalability for computation tasks   |   100.00% |   102.00% |   100.85% |    95.76%
   -- IPC scalability                  |   100.00% |    99.36% |    98.87% |    98.26%
   -- Instruction scalability          |   100.00% |   101.01% |   101.01% |   101.01%
   -- Frequency scalability            |   100.00% |   101.63% |   100.98% |    96.49%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)        |            3317.0 |            3317.0 |            3317.0 |            3317.0
LB (number of explicit tasks executed)           |               1.0 |               1.0 |               1.0 |              0.96
LB (time executing explicit tasks)               |               1.0 |               1.0 |              0.99 |              0.99
Time per explicit task (average)                 |             52.08 |             54.38 |             55.16 |             58.34
Overhead per explicit task (synch %)             |              1.46 |              6.83 |               7.5 |              9.98
Overhead per explicit task (sched %)             |              3.08 |               3.3 |              3.58 |              4.06
Number of taskwait/taskgroup (total)             |             682.0 |             682.0 |             682.0 |             682.0
========================================================================================================================
```

Figure 29: Data extracted from modelfactors.out, Cut-off = 4



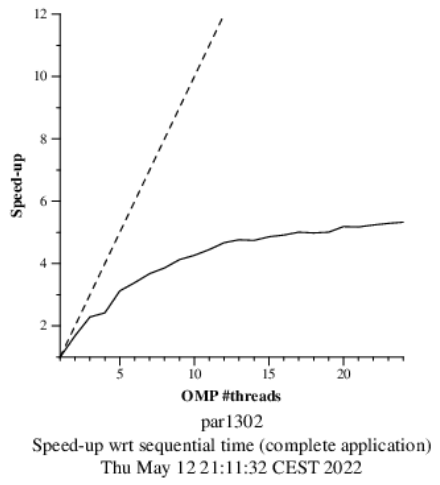
```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.57 |       0.38 |       0.22 |       0.14
Speedup                 |       1.00 |       1.52 |       2.63 |       3.95
Efficiency              |       1.00 |       0.76 |       0.66 |       0.49
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     94.12% |
------------------------------------------------------
Global efficiency                      |     82.05% |     64.08% |     59.50% |     48.45%
-- Parallelization strategy efficiency |     82.05% |     57.88% |     54.28% |     46.55%
   -- Load balancing                   |    100.00% |     99.65% |     98.57% |     98.59%
   -- In execution efficiency          |     82.05% |     58.08% |     55.07% |     47.22%
-- Scalability for computation tasks   |    100.00% |    110.71% |    109.63% |    104.08%
   -- IPC scalability                  |    100.00% |     89.57% |     89.41% |     88.93%
   -- Instruction scalability          |    100.00% |    112.81% |    112.90% |    112.93%
   -- Frequency scalability            |    100.00% |    109.56% |    108.60% |    103.63%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                                  Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)              |           57173.0 |           57173.0 |           57173.0 |           57173.0
LB (number of explicit tasks executed)                 |               1.0 |              0.99 |              0.99 |              0.97
LB (time executing explicit tasks)                     |               1.0 |               1.0 |              0.99 |              0.99
Time per explicit task (average)                       |              5.87 |              8.72 |              9.15 |             10.65
Overhead per explicit task (synch %)                   |              2.86 |             34.75 |              38.3 |             45.82
Overhead per explicit task (sched %)                   |             25.97 |             23.55 |             26.59 |             34.21
Number of taskwait/taskgroup (total)                   |            2730.0 |            2730.0 |            2730.0 |            2730.0
========================================================================================================================
```

Figure 30: Data extracted from modelfactors.out, Cut-off = 8

3.2.1 Optional 1

We have explored the scalability of our *tree* implementation with *cut-off* when using up to 24 threads. The results are shown in the graph of Figure 31. We set the cut-off value to 5 since it was the optimal value.

Before executing the exercise, we thought that the results would be falsified by the boada limits, but then we realised that it exceeded them. The speed-up grows almost linearly until 12 threads. We found that beyond 12 threads, the CPU can't balance the workload between all threads and that causes the semilinear increase progression, causing it to depart from the ideal speed-up. It is caused due to the overhead generated. It will grow up slower, but not as much as in threads 1 to 12. The speed-up still increasing its values because the synchronization overheads don't increase due to cut-off.



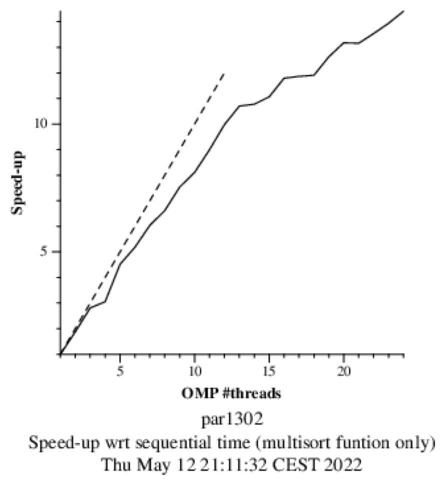


Figure 31: Scalability and speed-up plot using the cut-off mechanism in the tree strategy with 24 threads

3.3 Using *OpenMP* task dependencies

Through the previous tree strategy code with cut-off mechanism, we've generated the multisort-omp-depend-task.c included in this delivery. We've added some dependency clauses for minimizing the amount of taskgroup and taskwait the program needs. This dependency clauses protect certain memory regions for the parallelisation.

Multisort function only modifies the values at the end, when basicsort function is executed. For this reason, due to the recursion, the calls between multisort doesn't modify any memory address so we do not need to specify a depend in clause but we need a depend out for the vector data. The variable tmp is not required to take into account because it is a private variable for each recursion execution.

The following two merge functions requires a depend in clause of half of the current data vector, because it could be modified by other merge functions at the same time and we need to avoid data race. Now, as tmp is used at the execution of the final merge, we had put a depend out clause.

In the last merge we have a depend in of the entire tmp vector. However, we don't have a depend out clause because we have a `pragma_omp_taskwait` that ensures that there won't be any data race inside the data vector.

After compiling it and submitting it to Boada for parallel execution using 8 processors, with the level of cut-off=5 considered appropriate from the previous section, we made sure that the program verified the result of the sort process and did not throw errors about unordered positions. The execution time for the multisort function is 0.926 seconds. Here is the output:

```
make: 'multisort-omp' is up to date.
::::::::::::::
multisort-omp_8_5_boada-2.times.txt
::::::::::::::
*************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=5
Number of threads in OpenMP:          OMP_NUM_THREADS=8
*************************************************************************************
Initialization time in seconds: 0.856774
Multisort execution time: 0.925651
Check sorted data execution time: 0.017616
Multisort program finished
*************************************************************************************
```

Figure 32: Output of `multisort-omp-depend-task`, cut-off=5, 8 processors

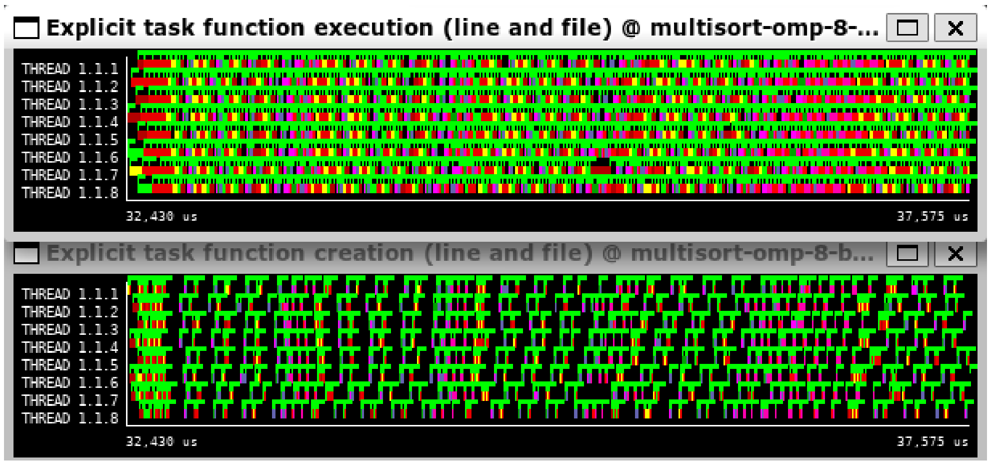
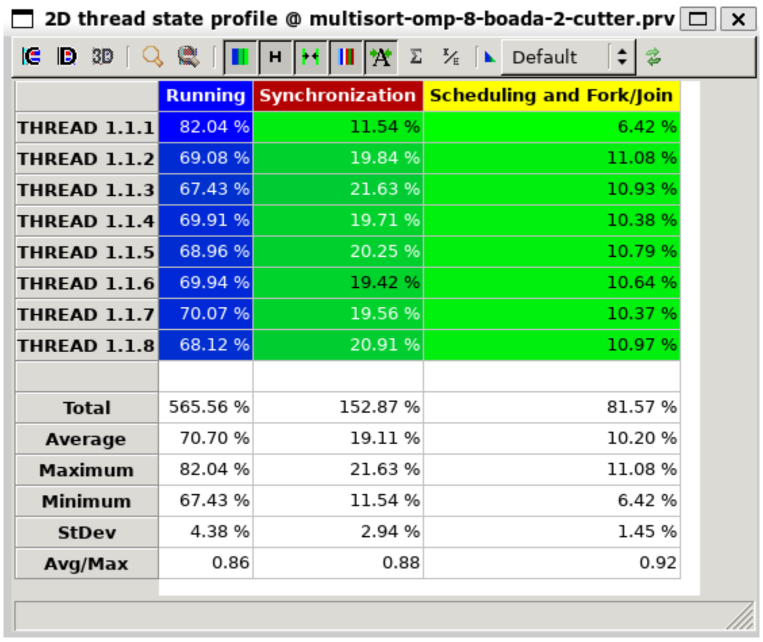


Figure 33: Explicit task execution and creation, cut-off = 5, using task dependencies (zoomed in)

In Figure 33 we can see that lots of tasks are created; 13461 tasks are executed (seen in Figure 36 modelfactors.out). Figure 34 shows that lots of threads are running, one a bit more than the others.

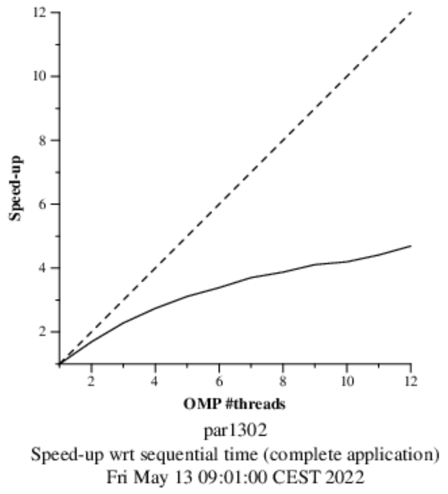
Looking at modelfactors and the scalability pots, what we can extract is that compared to the previous section, with cut-off = 5 the performance is almost the same. The time used for synchronization has increased a little bit because there isn't a taskgroup that stops the thread while waits to finish the other tasks.


2D thread state profile @ multisort-omp-8-boada-2-cutter.prv
Default

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	82.04 %	11.54 %	6.42 %
THREAD 1.1.2	69.08 %	19.84 %	11.08 %
THREAD 1.1.3	67.43 %	21.63 %	10.93 %
THREAD 1.1.4	69.91 %	19.71 %	10.38 %
THREAD 1.1.5	68.96 %	20.25 %	10.79 %
THREAD 1.1.6	69.94 %	19.42 %	10.64 %
THREAD 1.1.7	70.07 %	19.56 %	10.37 %
THREAD 1.1.8	68.12 %	20.91 %	10.97 %
Total	565.56 %	152.87 %	81.57 %
Average	70.70 %	19.11 %	10.20 %
Maximum	82.04 %	21.63 %	11.08 %
Minimum	67.43 %	11.54 %	6.42 %
StDev	4.38 %	2.94 %	1.45 %
Avg/Max	0.86	0.88	0.92

Figure 34: 2D state profile cut-off = 5, using task dependencies

In the version with dependencies, threads are in running state more time than waiting, however there are overheads with synchronization, this is different in the cut-off without dependencies because the thread are less time running but more waiting by the taskgroup statement.



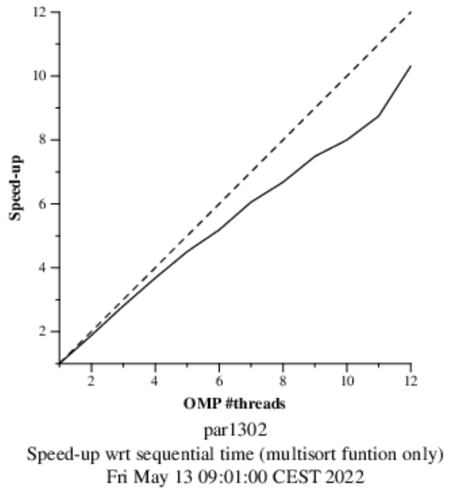
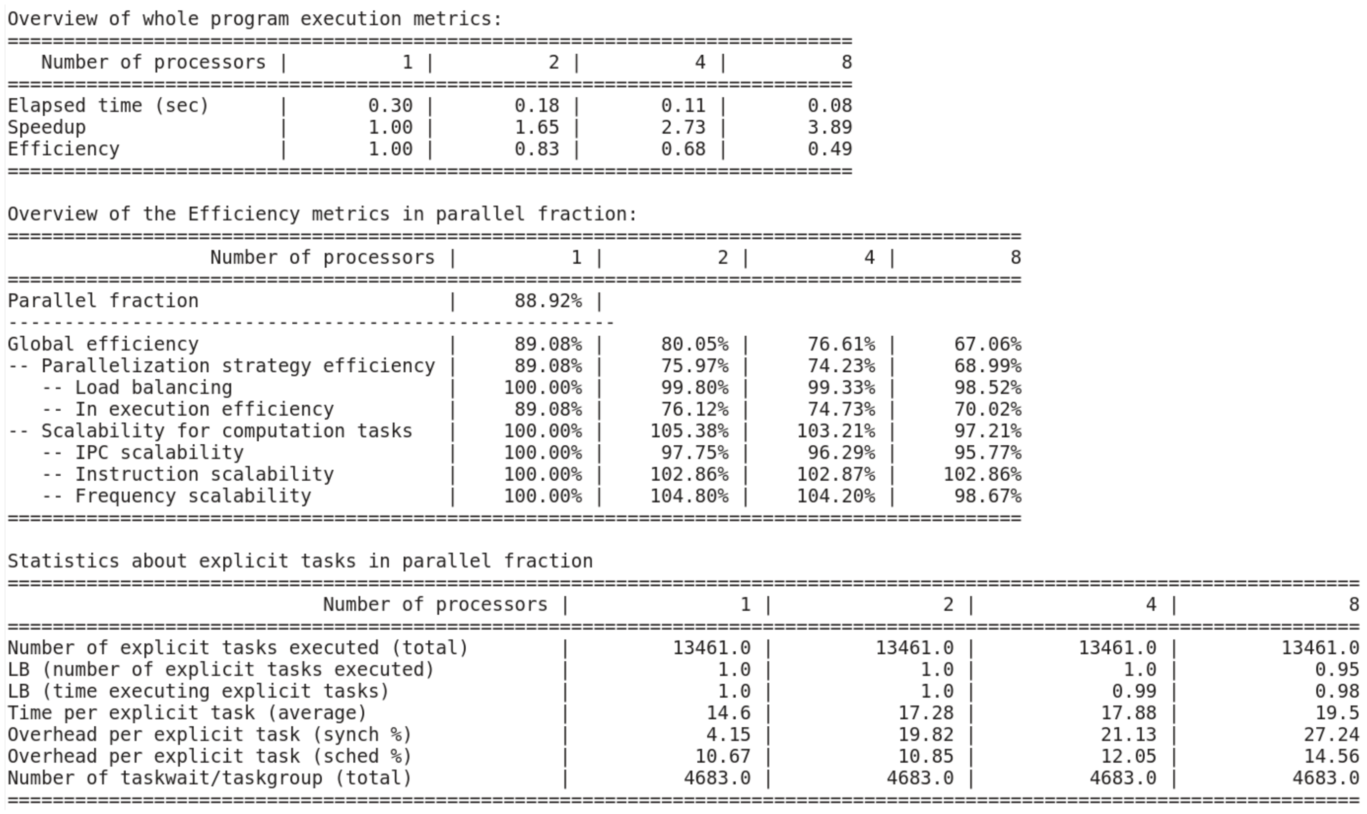


Figure 35: Scalability and speed-up plot, cut-off = 5, using tasks dependencies



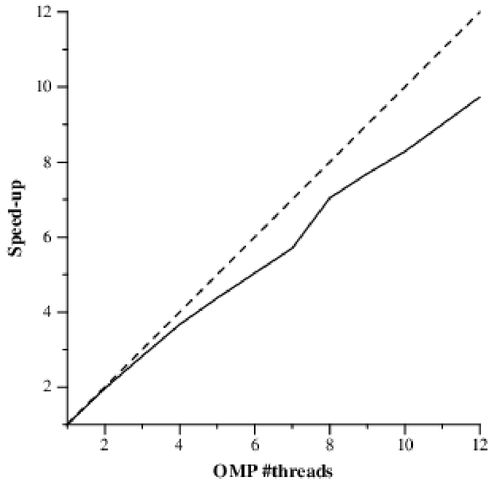
```
Overview of whole program execution metrics:
===========================================================================
   Number of processors |          1 |          2 |          4 |          8
===========================================================================
Elapsed time (sec)      |       0.30 |       0.18 |       0.11 |       0.08
Speedup                 |       1.00 |       1.65 |       2.73 |       3.89
Efficiency              |       1.00 |       0.83 |       0.68 |       0.49
===========================================================================

Overview of the Efficiency metrics in parallel fraction:
==========================================================================================
                  Number of processors |          1 |          2 |          4 |          8
==========================================================================================
Parallel fraction                      |     88.92% |
------------------------------------------------------
Global efficiency                      |     89.08% |     80.05% |     76.61% |     67.06%
-- Parallelization strategy efficiency |     89.08% |     75.97% |     74.23% |     68.99%
   -- Load balancing                   |    100.00% |     99.80% |     99.33% |     98.52%
   -- In execution efficiency          |     89.08% |     76.12% |     74.73% |     70.02%
-- Scalability for computation tasks   |    100.00% |    105.38% |    103.21% |     97.21%
   -- IPC scalability                  |    100.00% |     97.75% |     96.29% |     95.77%
   -- Instruction scalability          |    100.00% |    102.86% |    102.87% |    102.86%
   -- Frequency scalability            |    100.00% |    104.80% |    104.20% |     98.67%
==========================================================================================

Statistics about explicit tasks in parallel fraction
========================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8
========================================================================================================================
Number of explicit tasks executed (total)        |           13461.0 |           13461.0 |           13461.0 |           13461.0
LB (number of explicit tasks executed)           |               1.0 |               1.0 |               1.0 |              0.95
LB (time executing explicit tasks)               |               1.0 |               1.0 |              0.99 |              0.98
Time per explicit task (average)                 |              14.6 |             17.28 |             17.88 |              19.5
Overhead per explicit task (synch %)             |              4.15 |             19.82 |             21.13 |             27.24
Overhead per explicit task (sched %)             |             10.67 |             10.85 |             12.05 |             14.56
Number of taskwait/taskgroup (total)             |            4683.0 |            4683.0 |            4683.0 |            4683.0
========================================================================================================================
```

Figure 36: Data extracted from modelfactors.out, Cut-off = 5, using tasks dependecies

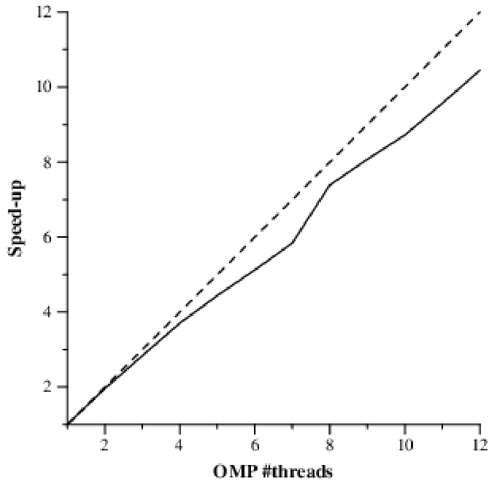
3.3.1 Optional 2

To parallelize the the two functions that initialise the *data* and *tmp* vectors we added the respective *#pragma omp parallel* and *#pragma omp single* to *initialize(N, data)* and to *clear(N, tmp)* in main calls. In to the functions we've just added a taskloop clause.

The results are awesome. We obtained a scalability much better than the other times we parallelized the program and the speed-up looks fine too.



par1302
Speed-up wrt sequential time (complete application)
Mon May 16 11:45:28 CEST 2022



par1302
Speed-up wrt sequential time (multisort funtion only)
Mon May 16 11:45:28 CEST 2022

Figure 37: Scalability and speed-up plot for optional 2

We can see that the initialization code is near linear parallelism. This fact reduce our execution time close to Time/Thread. This may be explained by the way we create tasks with the task loop clauses and is shown in the chronogram in Figure 38 and 39.

At the Figure 38 we see the near linear parallelization of initialize function and if we zoom a bit more into orange part, we see the same with the clear function at Figure 39.

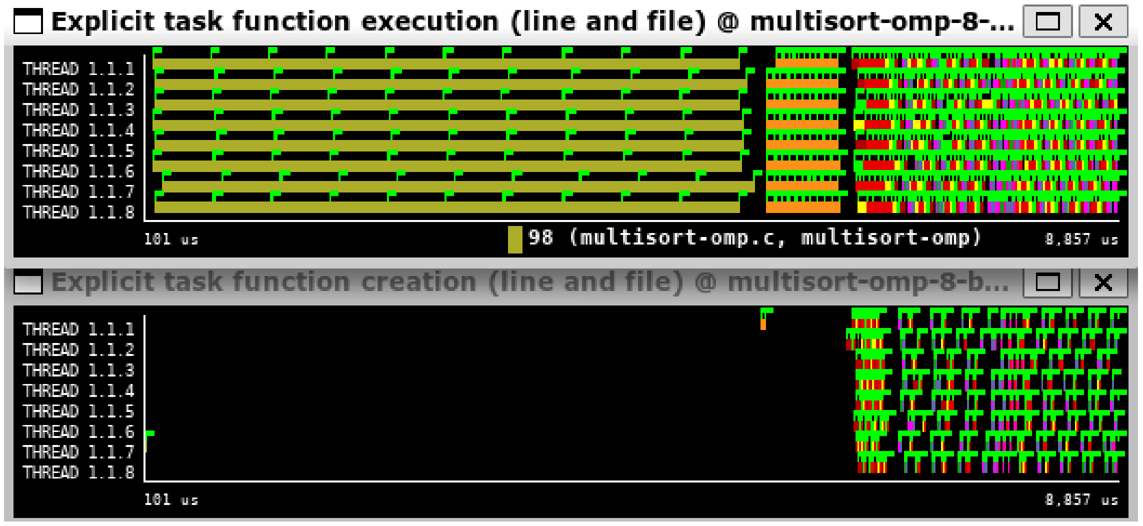

Explicit task function execution (line and file) @ multisort-omp-8-...
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
101 us
98 (multisort-omp.c, multisort-omp)
8,857 us
Explicit task function creation (line and file) @ multisort-omp-8-b...
101 us
8,857 us

Figure 38: Explicit task execution and creation, Optional 2 (zoomed in)

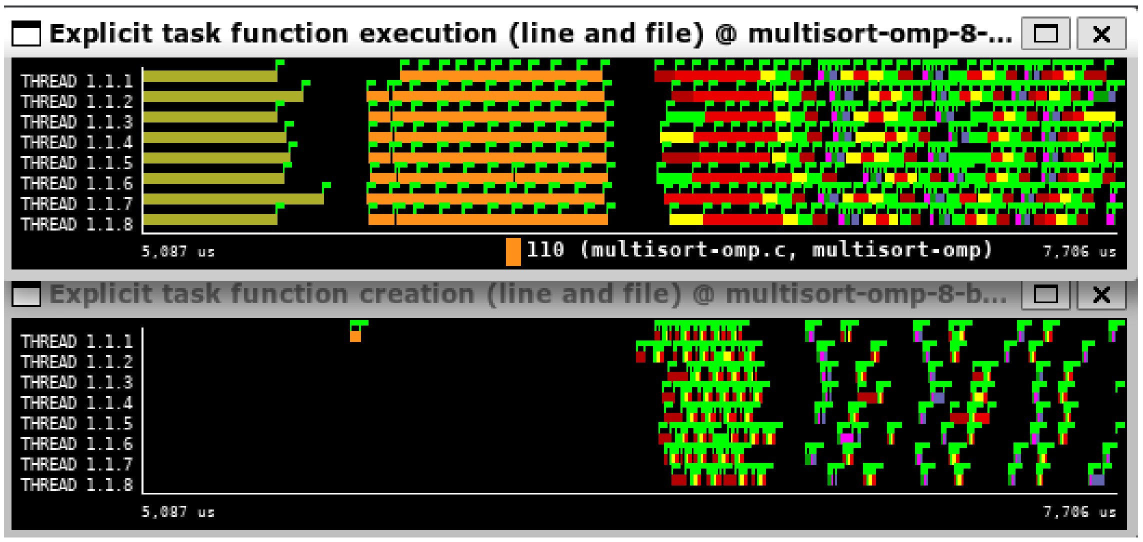
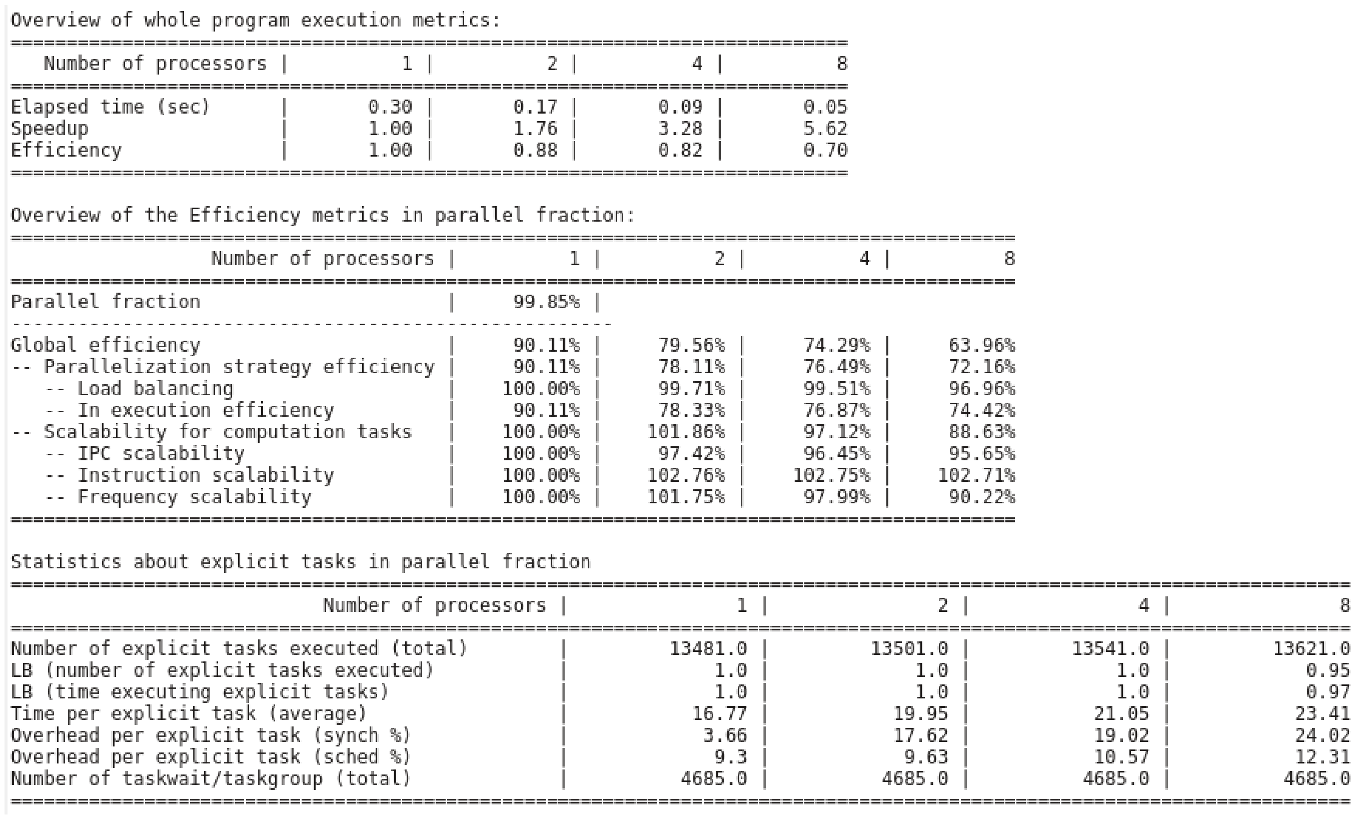

Explicit task function execution (line and file) @ multisort-omp-8-...
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
5,087 us
110 (multisort-omp.c, multisort-omp)
7,706 us
Explicit task function creation (line and file) @ multisort-omp-8-b...
5,087 us
7,706 us

Figure 39: Explicit task execution and creation, Optional 2 (more zoomed in)

When the taskloop clause ensure the end of all tasks we proceed to execute mutlisort function. So the first strange brown part is from initialize function, de second orange one from clear and finally multisort seen before.

Our complete application runtime has evolved from a maximum speed-up of around 3.8 to 5.62 (Figure 40). This is because we parallelized 99.85% of the code while in other executions the fact of not parallelizing the initialization code caused this value to drop between 83 and 94%.



```
Overview of whole program execution metrics:
============================================================================
   Number of processors |         1 |         2 |         4 |         8
============================================================================
Elapsed time (sec)      |      0.30 |      0.17 |      0.09 |      0.05
Speedup                 |      1.00 |      1.76 |      3.28 |      5.62
Efficiency              |      1.00 |      0.88 |      0.82 |      0.70
============================================================================

Overview of the Efficiency metrics in parallel fraction:
===========================================================================================
                  Number of processors |         1 |         2 |         4 |         8
===========================================================================================
Parallel fraction                      |    99.85% |
-------------------------------------------------------
Global efficiency                      |    90.11% |    79.56% |    74.29% |    63.96%
-- Parallelization strategy efficiency |    90.11% |    78.11% |    76.49% |    72.16%
   -- Load balancing                   |   100.00% |    99.71% |    99.51% |    96.96%
   -- In execution efficiency          |    90.11% |    78.33% |    76.87% |    74.42%
-- Scalability for computation tasks   |   100.00% |   101.86% |    97.12% |    88.63%
   -- IPC scalability                  |   100.00% |    97.42% |    96.45% |    95.65%
   -- Instruction scalability          |   100.00% |   102.76% |   102.75% |   102.71%
   -- Frequency scalability            |   100.00% |   101.75% |    97.99% |    90.22%
===========================================================================================

Statistics about explicit tasks in parallel fraction
=========================================================================================================================
                            Number of processors |                 1 |                 2 |                 4 |                 8
=========================================================================================================================
Number of explicit tasks executed (total)        |           13481.0 |           13501.0 |           13541.0 |           13621.0
LB (number of explicit tasks executed)           |               1.0 |               1.0 |               1.0 |              0.95
LB (time executing explicit tasks)               |               1.0 |               1.0 |               1.0 |              0.97
Time per explicit task (average)                 |             16.77 |             19.95 |             21.05 |             23.41
Overhead per explicit task (synch %)             |              3.66 |             17.62 |             19.02 |             24.02
Overhead per explicit task (sched %)             |               9.3 |              9.63 |             10.57 |             12.31
Number of taskwait/taskgroup (total)             |            4685.0 |            4685.0 |            4685.0 |            4685.0
=========================================================================================================================
```

Figure 40: Data extracted from modelfactors.out, Optional 2 - Initialize parallelization

4 Final Conclusions

To conclude, in this laboratory assignment, we've worked with different OpenMP parallelization strategies of the Mergesort algorithm. This recursive program had the goal of sorting using a divide and conquer plan.

We have seen the leaf strategy and the tree strategy. We can say that the tree strategy is better for this kind of sorting problems. Then, for the tree strategy, we have analysed two mechanisms: the cut-off system and the dependencies between tasks. These two had a very similar performance. The first one was less time in running state, whereas the version with dependences was more time on it.