PAR Laboratory Assignment

Lab4: Divide and Conquer parallelism with OpenMP: Sorting

Sergio Araguás Villas - par4107
David Pérez Barroso - par4121
29-11-2022
22-23 Q1

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Session 1: Task decomposition analysis for Merge Sort

In this session we will study the types of task decomposition for the code that implements a merge sort, a recursive algorithm that continuously splits the input array in half until it cannot be further divided, then merges the parts of the array in order that it is sorted using the function *merge*.

”Divide and conquer”

To study its behavior we start executing the file *multisort-seq.c* with the following command ***sbatch submit-seq.sh multisort-seq*** and observe its execution time so that it will be useful to compare with the future versions of it.

We obtain the next set of data once it is executed:

```
::::::::::::::
multisort-seq_seq_boada-11.times.txt
::::::::::::::
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
*****************************************************************************************
Initialization time in seconds: 0.682441
Multisort execution time: 5.191503
Check sorted data execution time: 0.011712
Multisort program finished
*****************************************************************************************
```

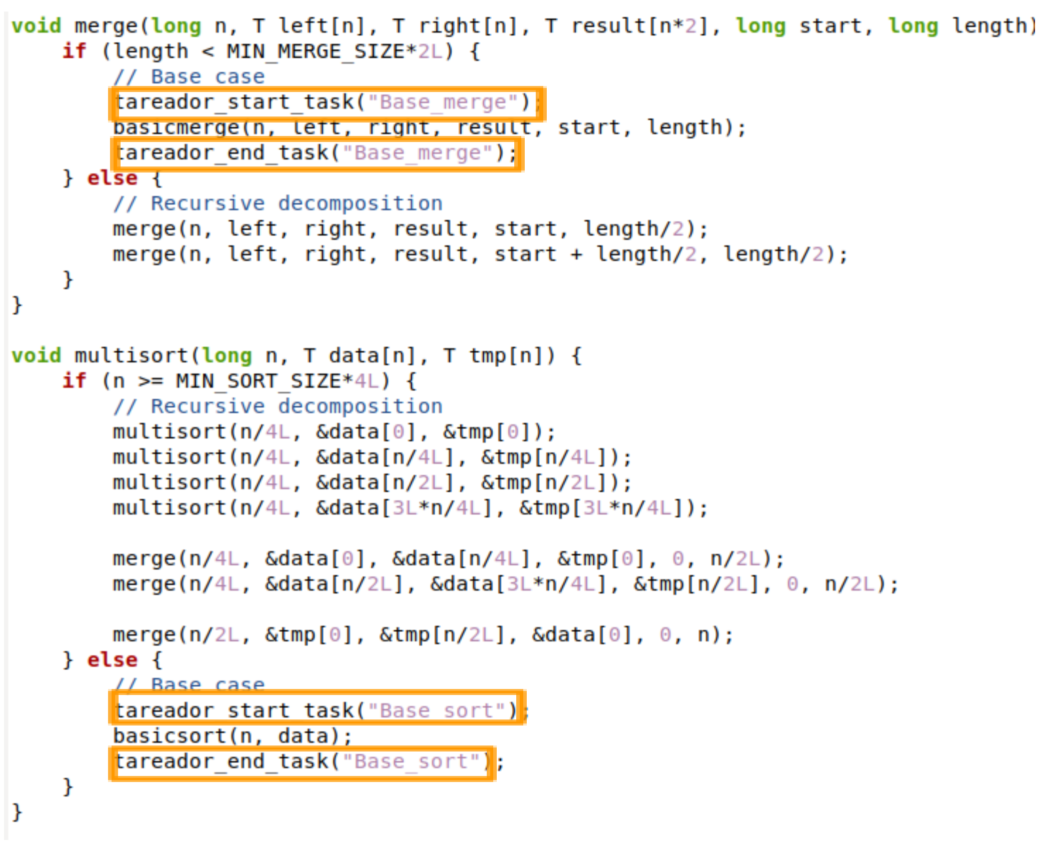
Task decomposition analysis with Tareador

In this section, we will investigate, using the *Tareador* tool, potential task decomposition strategies and their implications in terms of parallelism and task interactions required of the previous code.

Specifically, we will use two strategies for this code, Leaf strategy and Tree strategy.

Leaf strategy

To apply this strategy, we define the task in the section of code that invokes the function *basicsort* and the function *basicmerge*. The modified code that creates the tasks can be seen in the next figure. (Fig.1.)

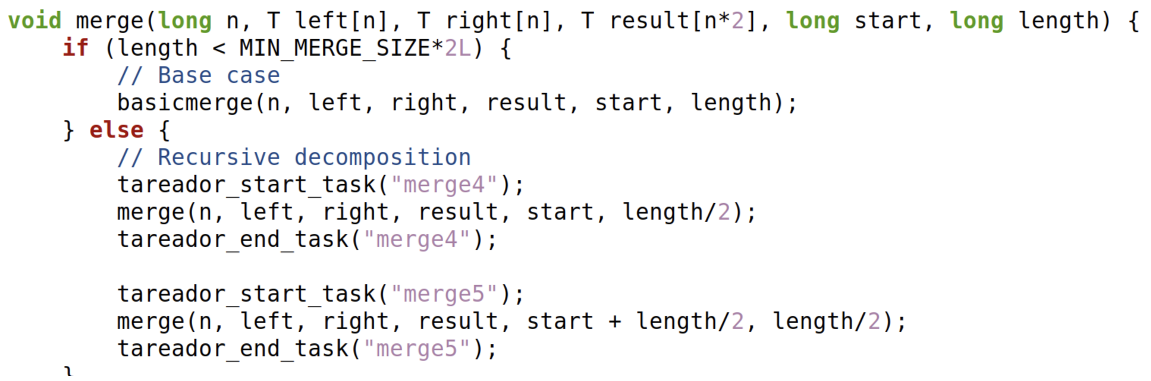


```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length)
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        tareador_start_task("Base_merge");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("Base_merge");
    } else {
        // Recursive decomposition
        merge(n, left, right, result, start, length/2);
        merge(n, left, right, result, start + length/2, length/2);
    }
}

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

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

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

Fig.1. Modified code for Leaf strategy

Tree strategy

In this strategy, we define the task in the section of code that invokes every function of the recursive decomposition part; *merge* and *multisort*. The modified code that creates the tasks can be seen in the next figure. (Fig.2.)



```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        tareador_start_task("merge4");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("merge4");

        tareador_start_task("merge5");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("merge5");
```

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("multisort1");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("multisort1");

        tareador_start_task("multisort2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("multisort2");

        tareador_start_task("multisort3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("multisort3");

        tareador_start_task("multisort4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("multisort4");

        tareador_start_task("merge1");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("merge1");

        tareador_start_task("merge2");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("merge2");

        tareador_start_task("merge3");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("merge3");
    } else {
        // Base case
        basicsort(n, data);
    }
```

Fig.2. Modified code for Tree strategy

The next step is to execute the two strategies with *Tareador*, to visualize the two tasks dependence graphs that the script generates. We obtain the next tasks dependence graphs:

For the Leaf Strategy we obtain the task dependence graph below (Fig.3.):

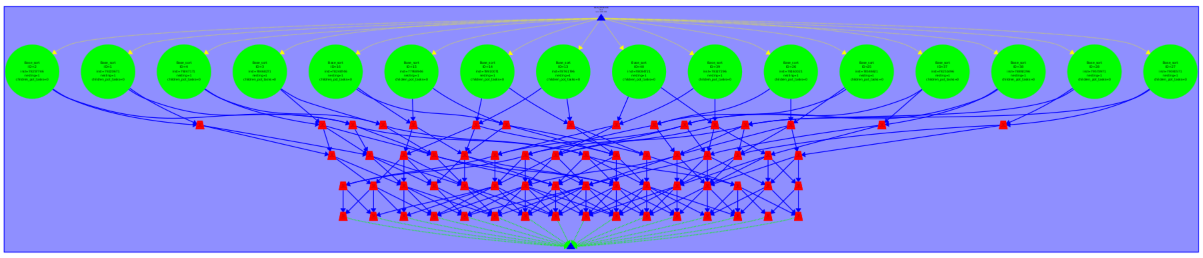


Fig.3. Task dependence graph Leaf strategy

For the Tree Strategy we obtain the task dependence graph below (Fig.4.):

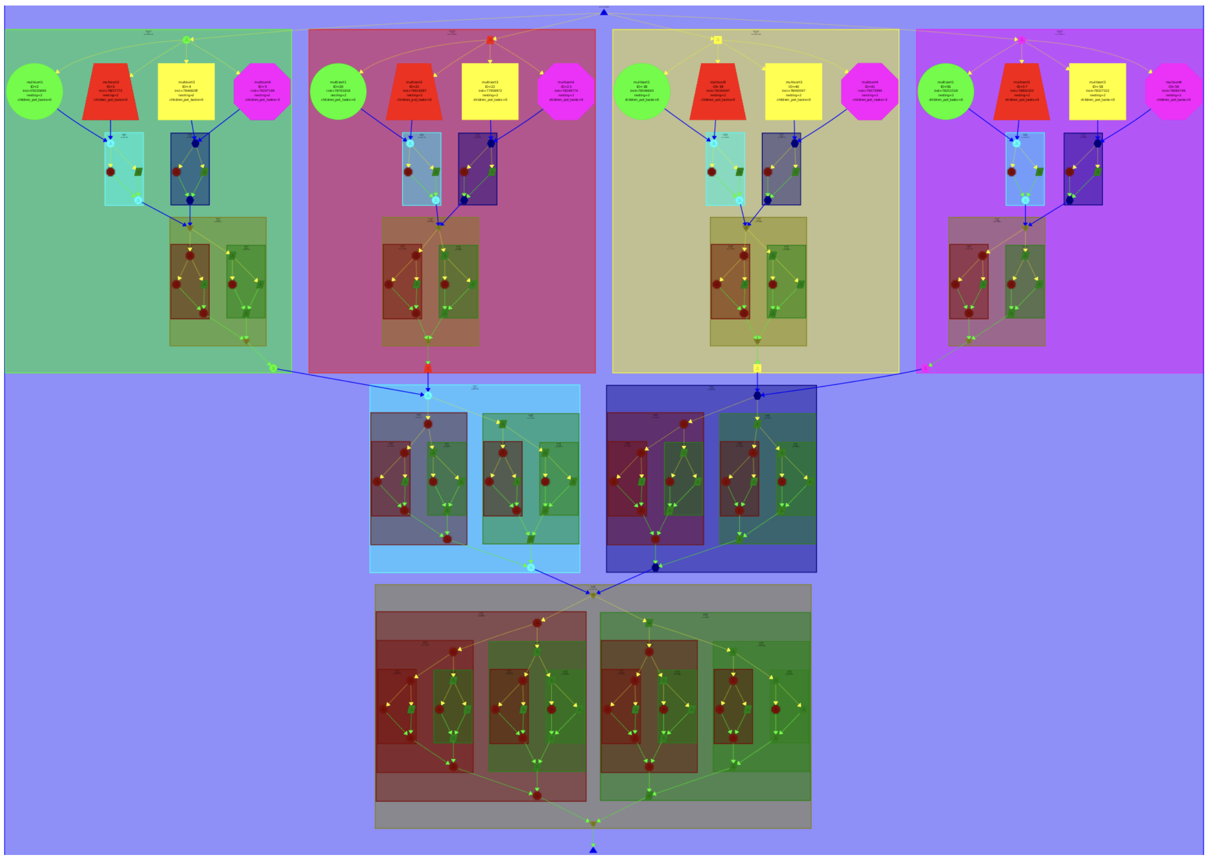


Fig.4. Task dependence graphs Tree strategy

As it can be observed from the two TDG, the differences are notables for each strategy. The first thing that it can be observed is that the Tree strategy has generated more tasks than the Leaf strategy, that is because there are more invocations to *merge* and *multisort* functions than to *basicsort* and *basicmerge* due the recursivity.

Also, the structure of the graphs are distinct, because in leaf strategy the tasks are generated only at leaf level, that is to say, that the tasks are only generated in the base cases of *merge* and *sort* functions, rather than the recursive case which is what the Tree strategy does.

We can also observe that the Tree strategy has more variety in the granularity of their tasks, the first tasks have a much larger granularity than the last ones causing an imbalance between tasks. This imbalance appears in the Leaf strategy too, where the *basicsort* tasks (green tasks) are much bigger than *basicmerge* tasks (red tasks).

In the Leaf strategy, we note that dependencies are created due the *basicmerge* function, which requires the two half of the arrays to be pre-sorted by the *basicsort* function. On the other hand, in the Tree strategy we observe the same dependencies; each recursive call to *merge* function needs its array parameter to be sorted first.

Analyzing the code, we conclude that in the Leaf decomposition strategy the need for synchronization is caused by the dependencies commented before, and it can be solved by adding the ***taskgroup*** directives, one containing all recursive calls of *multisort* function and one containing the first two *merge* calls.

In the case of Tree decomposition strategy the need for synchronization again is the *merge* functions as they need the array parameters sorted by *mergesort* functions and we can solve this by adding ***taskgroup*** directives as in Leaf strategy.

Finally, if we take a look at the simulation of the execution of the code with 20 processors on Paraver zoomed on the final part with *leaf strategy* (Fig.6.) and *tree strategy* (Fig.7.), we can note that in both cases the *merge* function is the one creating dependency, and that the multisort tasks are the ones that take the most time.

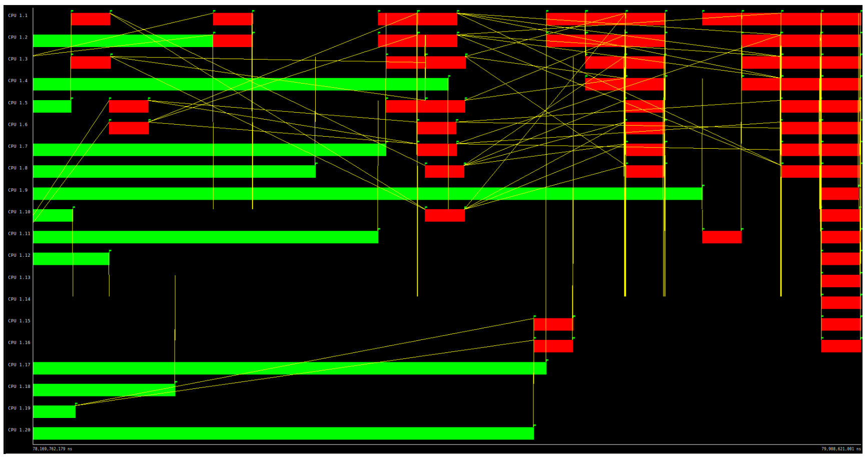


Fig.6. Paraver windows Leaf strategy

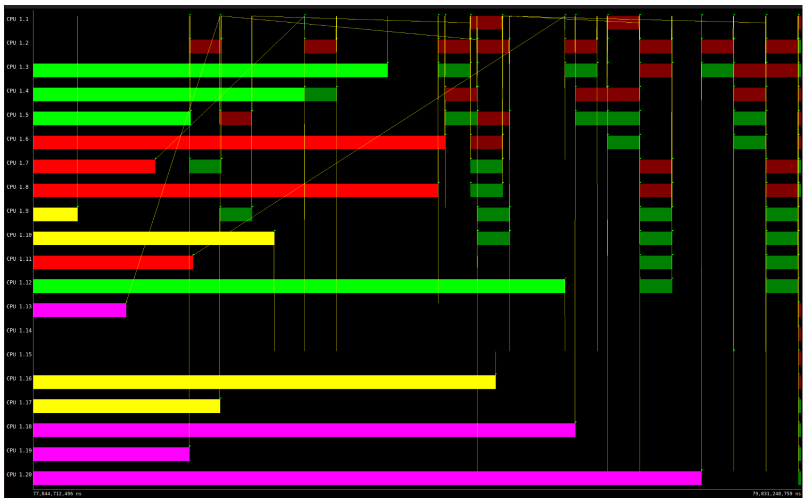


Fig.7. Paraver windows Tree strategy

Session 2: Shared-memory parallelisation with OpenMP tasks

In this session we will explore the two strategies that we saw in the last laboratory session, (Leaf strategy and Tree strategy) applied to the program multisort-omp.c and observe which results provokes these strategies to the execution of it.

Leaf strategy in OpenMP

To apply this strategy we add the directives *#pragma omp task* to the base case of the multisort function and to the base case of the merge function to create the leaf tasks. In order to avoid data race, we add a synchronization mechanism using the *#pragma omp taskwait* before the merge calls in the iterative part of multisort function.

Finally we also add the directives *#pragma omp parallel* and *#pragma omp single* before the call of *multisort* function in the *main* body to create the parallel region and to ensure that the tasks are created by one thread.

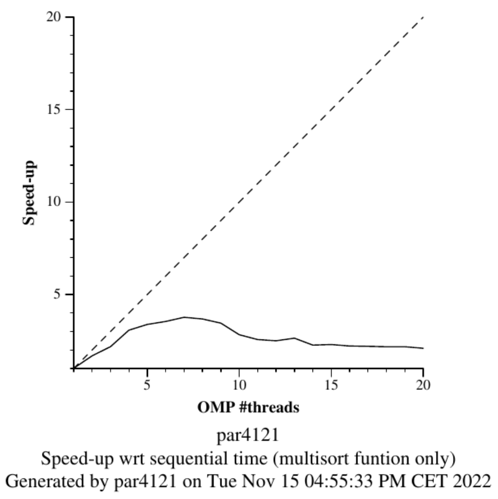
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        #pragma omp task
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        merge(n, left, right, result, start, length/2);
        merge(n, left, right, result, start + length/2, length/2);
    }
}

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

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

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

Fig.8. Code modification Leaf Strategy

Once we executed the program with *sbatch ./submit-omp.sh mulitsort-omp* 2 and default values, we obtain the next result:

```
******************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                CUTOFF=50
Number of threads in OpenMP:         OMP_NUM_THREADS=2
******************************************************************************
***********
Initialization time in seconds: 0.681686
Multisort execution time: 3.113451
Check sorted data execution time: 0.013375
Multisort program finished
******************************************************************************
```

We can observe in the output that the execution time has significantly decreased using two threads in comparison to the sequential version, obtaining 3.113451 *seconds* of the multisort execution time in contrast to the 5.191503 *seconds* that we had before.



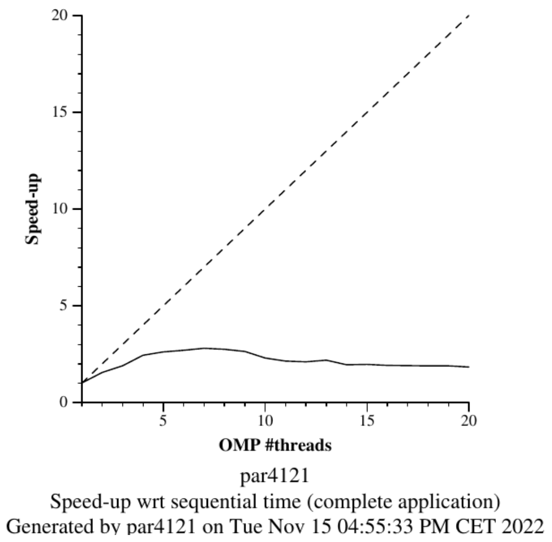


Fig.9. Plots of speed-up of Leaf strategy

We can observe on the plots above, that the speedups aren't as good as the expected speedups from parallel codes, this is because the leaf strategy makes the execution of a big percentage of tasks sequential.

Next, we execute the modelfactors analysis for this strategy obtaining the following tables:

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	0.21	0.21	0.22	0.33	0.24
Speedup	1.00	0.87	1.01	1.03	1.00	1.00	0.96	0.63	0.88
Efficiency	1.00	0.44	0.25	0.17	0.13	0.10	0.08	0.05	0.05

Overview of the Efficiency metrics in parallel fraction, ϕ=89.30%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	92.02%	39.73%	23.41%	15.96%	11.64%	9.26%	7.42%	4.02%	5.02%
Parallelization strategy efficiency	92.02%	52.56%	36.42%	27.38%	20.95%	16.95%	13.97%	9.81%	9.55%
Load balancing	100.00%	97.24%	90.41%	62.76%	37.49%	30.61%	24.06%	13.89%	16.68%
In execution efficiency	92.02%	54.05%	40.28%	43.62%	55.89%	55.39%	58.07%	70.64%	57.25%
Scalability for computation tasks	100.00%	75.59%	64.29%	58.28%	55.55%	54.63%	53.13%	40.95%	52.50%
IPC scalability	100.00%	68.02%	57.64%	54.79%	52.44%	52.95%	51.71%	40.21%	50.68%
Instruction scalability	100.00%	112.17%	113.13%	112.95%	111.68%	112.30%	111.73%	109.56%	111.66%
Frequency scalability	100.00%	99.08%	98.59%	94.18%	94.84%	91.86%	91.95%	92.95%	92.79%

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	0.71	0.75	0.78	0.78	0.8	0.82	0.82	0.82
LB (time executing explicit tasks)	1.0	0.79	0.81	0.78	0.77	0.8	0.81	0.81	0.83
Time per explicit task (average us)	2.74	3.54	4.01	4.18	3.99	4.11	4.1	3.93	3.98
Overhead per explicit task (synch %)	0.84	71.0	175.8	322.29	531.33	688.79	901.22	1840.02	1452.95
Overhead per explicit task (sched %)	9.5	39.44	45.93	33.62	25.92	25.8	24.72	28.25	30.56
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Fig.10. Tables of the modelfacotrs analysis of the leaf strategy

Observing the tables, we can see that the global efficiency and the parallelization strategy efficiency greatly decrease as more threads are added to the execution. We think that these low values are caused by the amount of overheads per explicit task due the threads synchronization that, opposite to efficiency, highly increase as more threads are added.

Finally, we execute the paraver analysis to deepen in the execution of the leaf strategy obtaining the following time lines and histograms:

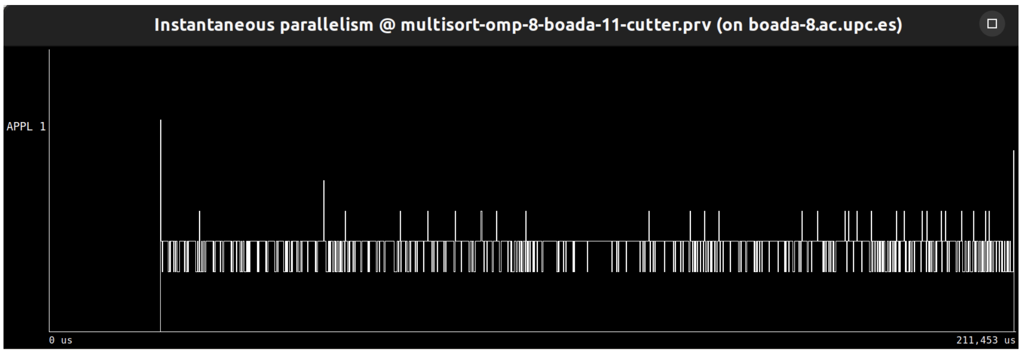
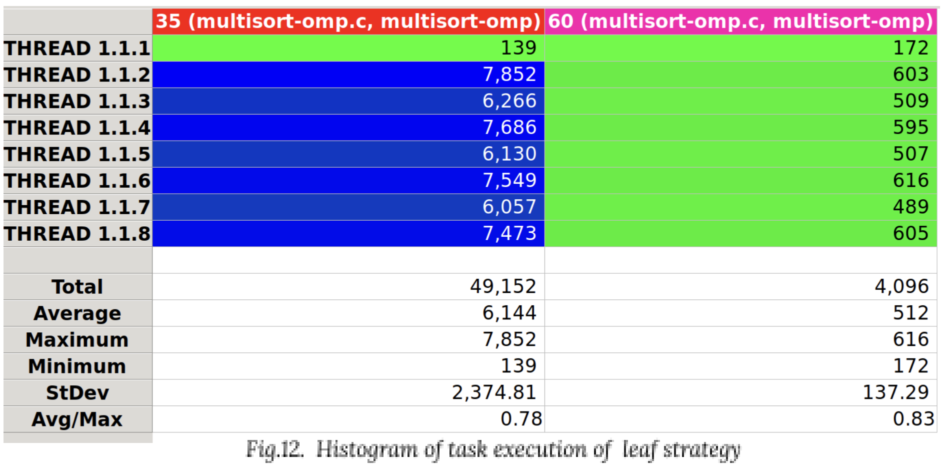

Instantaneous parallelism @ multisort-omp-8-boada-11-cutter.prv (on boada-8.ac.upc.es)
APPL 1
0 us
211,453 us

Fig.11. Trace of instantaneous parallelisation of leaf strategy



	35 (multisort-omp.c, multisort-omp)	60 (multisort-omp.c, multisort-omp)
THREAD 1.1.1	139	172
THREAD 1.1.2	7,852	603
THREAD 1.1.3	6,266	509
THREAD 1.1.4	7,686	595
THREAD 1.1.5	6,130	507
THREAD 1.1.6	7,549	616
THREAD 1.1.7	6,057	489
THREAD 1.1.8	7,473	605
Total	49,152	4,096
Average	6,144	512
Maximum	7,852	616
Minimum	139	172
StDev	2,374.81	137.29
Avg/Max	0.78	0.83

Fig.12. Histogram of task execution of leaf strategy

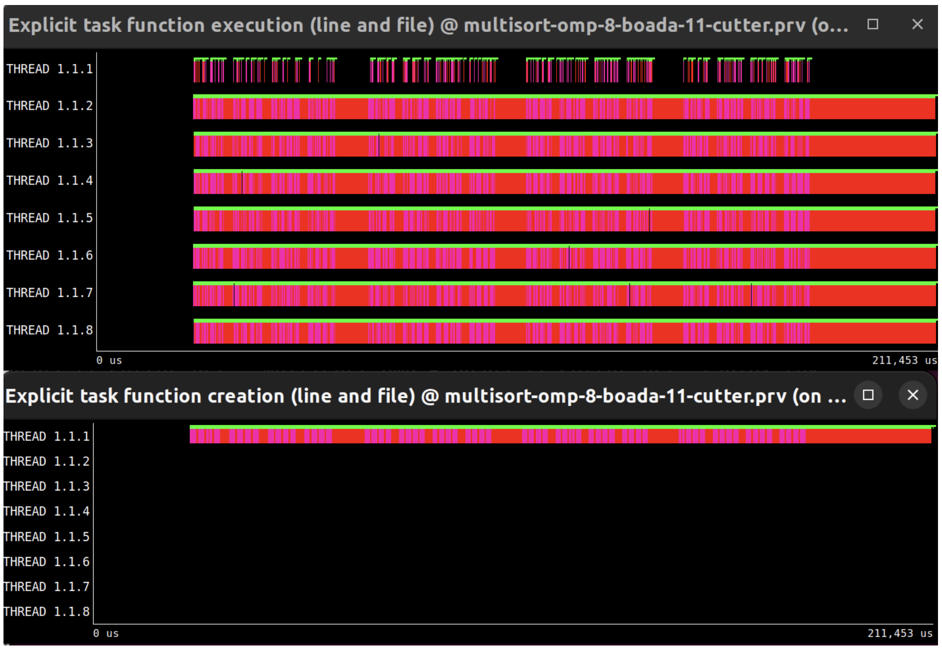


Fig.13. Trace of tasks creation and execution of leaf strategy

If we observe the histograms above we can conclude that thread one is the thread dedicated to the creation of tasks as we can note that it executes less tasks than the other threads which distribute the workload equally.

The program generates enough tasks for every thread except for the first one that is dedicated to the creation of tasks with an average simultaneous execution through the program of 7-8 tasks excluding the synchronization time which takes up an important percentage of the program.

Tree strategy in OpenMP

Now we repeat the process with the *Tree strategy* obtaining less than half the time of the time execution in *Leaf strategy* (1,48 in the case of *Tree strategy* and 3,11 in the case of *Leaf strategy*) Here we use the task group directive so the tasks have to wait for their ancestors.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        #pragma omp task
        merge(n, left, right, result, start, length/2);
        #pragma omp task
        merge(n, left, right, result, start + length/2, length/2);
    }
}

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



        #pragma omp taskgroup
        {
          #pragma omp task
          merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
          #pragma omp task
          merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        }

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

```
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                    CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=4
*****************************************************************************************
Initialization time in seconds: 0.681986
Multisort execution time: 1.484865
Check sorted data execution time: 0.014551
Multisort program finished
*****************************************************************************************
```

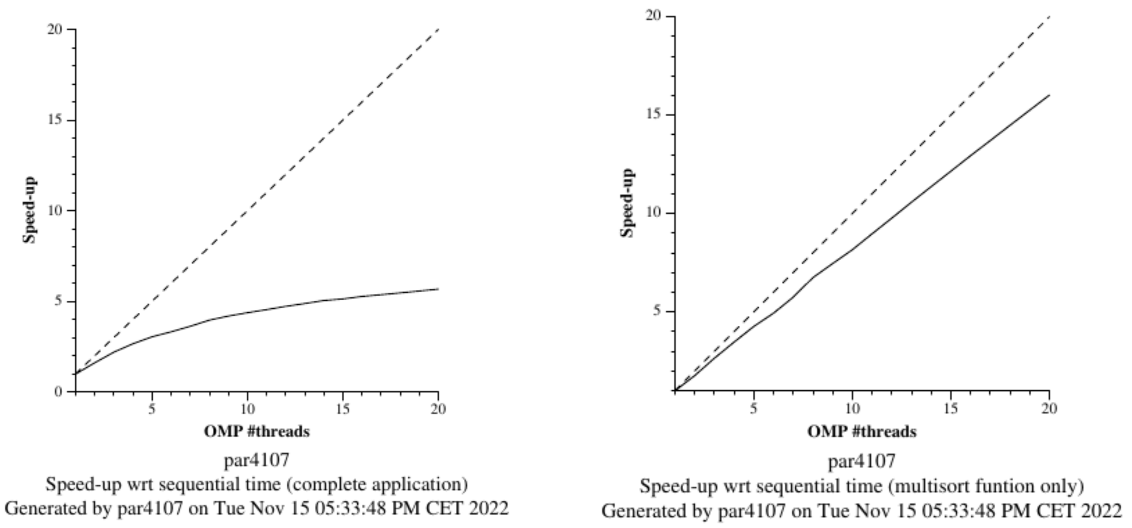
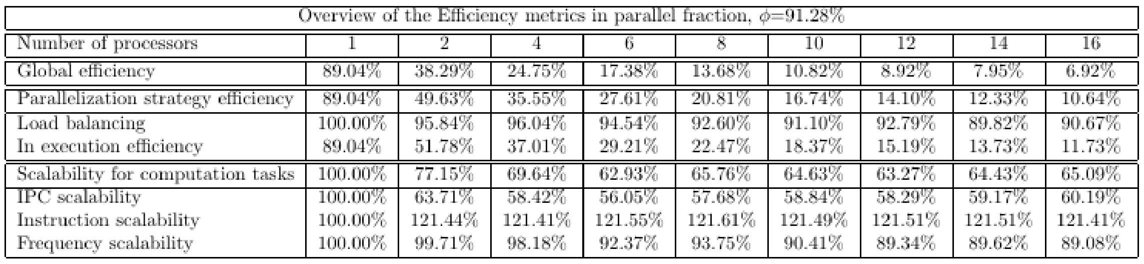


Fig.14. Plots of speed-up of Tree strategy

The plots generated for the *tree strategy* show us a good parallelization performance if we center our attention in the multisort function only (which is the parallel region), but on the other hand the strong scalability of the complete application is still not enough.

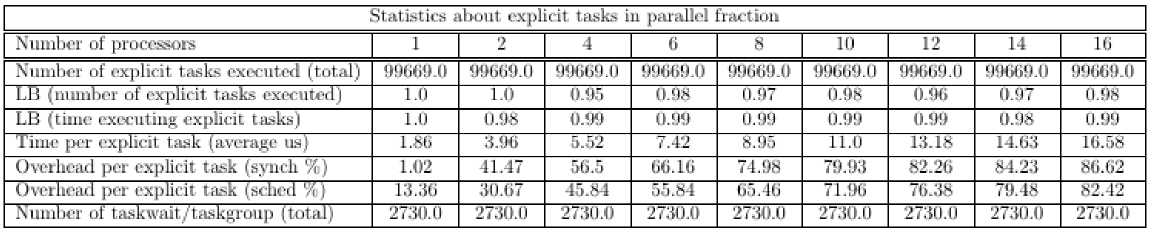
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.27	0.31	0.24	0.23	0.22	0.22	0.23	0.22	0.22
Speedup	1.00	0.87	1.10	1.15	1.20	1.18	1.17	1.21	1.21
Efficiency	1.00	0.44	0.28	0.19	0.15	0.12	0.10	0.09	0.08

Table 1: Analysis done on Mon Nov 21 02:54:55 PM CET 2022, par4107



Overview of the Efficiency metrics in parallel fraction, ϕ=91.28%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	89.04%	38.29%	24.75%	17.38%	13.68%	10.82%	8.92%	7.95%	6.92%
Parallelization strategy efficiency	89.04%	49.63%	35.55%	27.61%	20.81%	16.74%	14.10%	12.33%	10.64%
Load balancing	100.00%	95.84%	96.04%	94.54%	92.60%	91.10%	92.79%	89.82%	90.67%
In execution efficiency	89.04%	51.78%	37.01%	29.21%	22.47%	18.37%	15.19%	13.73%	11.73%
Scalability for computation tasks	100.00%	77.15%	69.64%	62.93%	65.76%	64.63%	63.27%	64.43%	65.09%
IPC scalability	100.00%	63.71%	58.42%	56.05%	57.68%	58.84%	58.29%	59.17%	60.19%
Instruction scalability	100.00%	121.44%	121.41%	121.55%	121.61%	121.49%	121.51%	121.51%	121.41%
Frequency scalability	100.00%	99.71%	98.18%	92.37%	93.75%	90.41%	89.34%	89.62%	89.08%

Table 2: Analysis done on Mon Nov 21 02:54:55 PM CET 2022, par4107



Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	1.0	0.95	0.98	0.97	0.98	0.96	0.97	0.98
LB (time executing explicit tasks)	1.0	0.98	0.99	0.99	0.99	0.99	0.99	0.98	0.99
Time per explicit task (average us)	1.86	3.96	5.52	7.42	8.95	11.0	13.18	14.63	16.58
Overhead per explicit task (synch %)	1.02	41.47	56.5	66.16	74.98	79.93	82.26	84.23	86.62
Overhead per explicit task (sched %)	13.36	30.67	45.84	55.84	65.46	71.96	76.38	79.48	82.42
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 Mon Nov 21 02:54:55 PM CET 2022, par4107

Fig.15. Tables of the modelfacotrs analysis of the tree strategy

Now if we take a look at the modelfactor tables we see that the global efficiency drastically decreases as we add more threads. We think this problem comes from the number of explicit tasks generated by the *tree strategy* as it doesn't stop generating tasks until it reaches the base case. This problem could be solved by adding a cut-off mechanism.

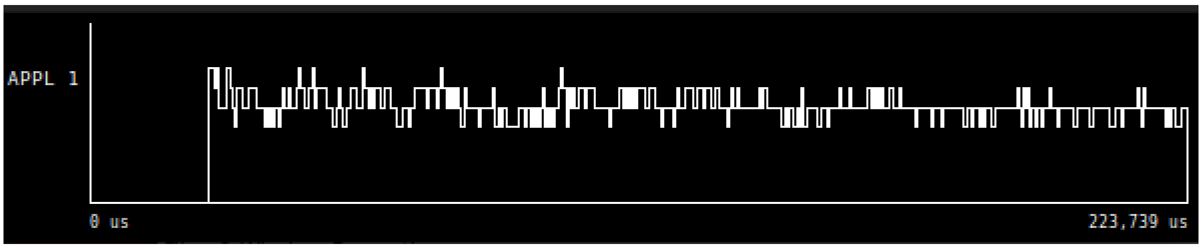


Fig.16. Trace of instantaneous parallelisation of tree strategy

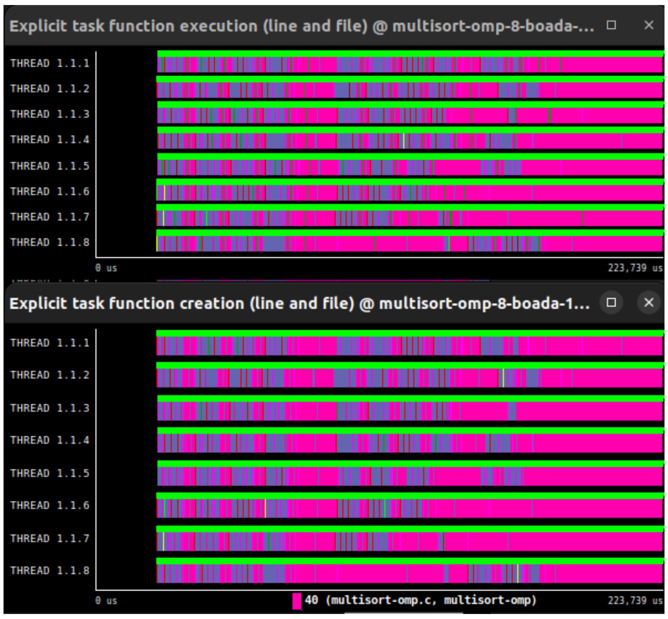
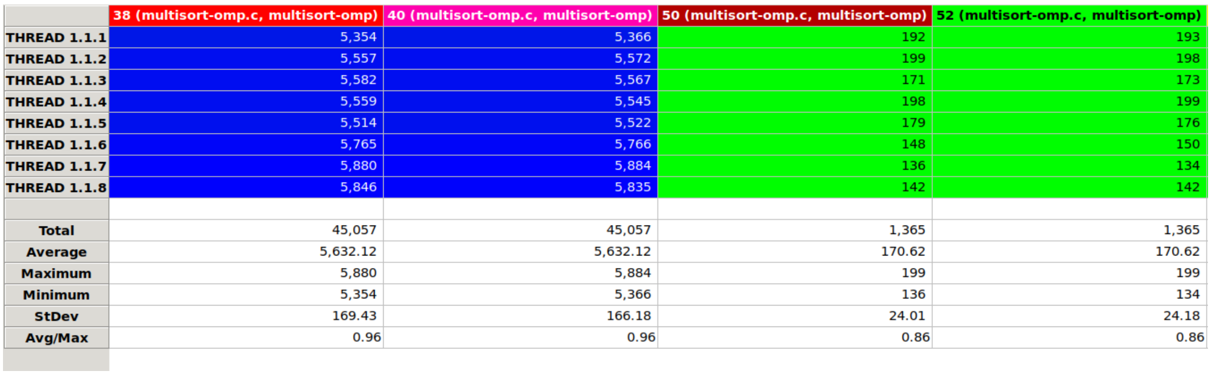
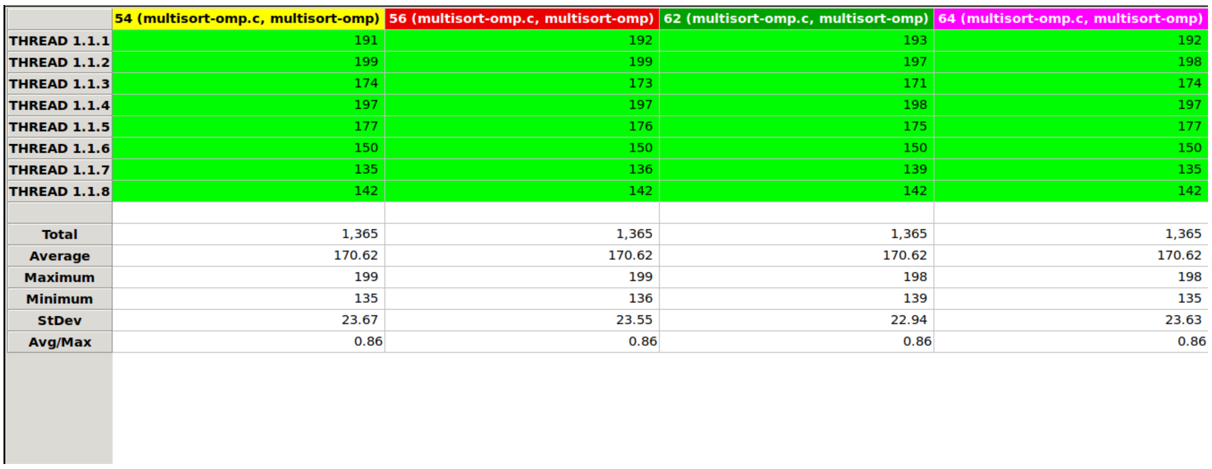


Fig.17. Trace of tasks creation and execution of tree strategy

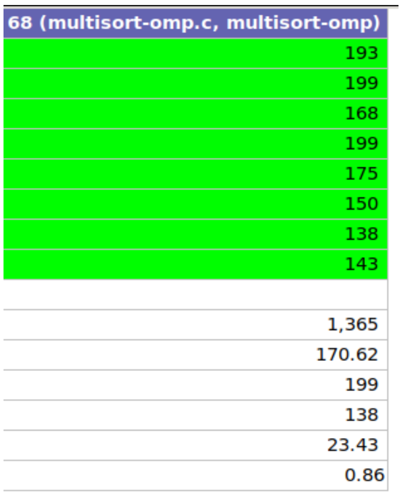
The Paraver histogram (Fig.18.) and (Fig.16) shows us that the number of tasks are equally distributed for all threads in time and that the creation of these tasks isn't exclusive to one thread as all threads generate them.



	38 (multisort-omp.c, multisort-omp)	40 (multisort-omp.c, multisort-omp)	50 (multisort-omp.c, multisort-omp)	52 (multisort-omp.c, multisort-omp)
THREAD 1.1.1	5,354	5,366	192	193
THREAD 1.1.2	5,557	5,572	199	198
THREAD 1.1.3	5,582	5,567	171	173
THREAD 1.1.4	5,559	5,545	198	199
THREAD 1.1.5	5,514	5,522	179	176
THREAD 1.1.6	5,765	5,766	148	150
THREAD 1.1.7	5,880	5,884	136	134
THREAD 1.1.8	5,846	5,835	142	142
Total	45,057	45,057	1,365	1,365
Average	5,632.12	5,632.12	170.62	170.62
Maximum	5,880	5,884	199	199
Minimum	5,354	5,366	136	134
StDev	169.43	166.18	24.01	24.18
Avg/Max	0.96	0.96	0.86	0.86



	54 (multisort-omp.c, multisort-omp)	56 (multisort-omp.c, multisort-omp)	62 (multisort-omp.c, multisort-omp)	64 (multisort-omp.c, multisort-omp)
THREAD 1.1.1	191	192	193	192
THREAD 1.1.2	199	199	197	198
THREAD 1.1.3	174	173	171	174
THREAD 1.1.4	197	197	198	197
THREAD 1.1.5	177	176	175	177
THREAD 1.1.6	150	150	150	150
THREAD 1.1.7	135	136	139	135
THREAD 1.1.8	142	142	142	142
Total	1,365	1,365	1,365	1,365
Average	170.62	170.62	170.62	170.62
Maximum	199	199	198	198
Minimum	135	136	139	135
StDev	23.67	23.55	22.94	23.63
Avg/Max	0.86	0.86	0.86	0.86



68 (multisort-omp.c, multisort-omp)
193
199
168
199
175
150
138
143
1,365
170.62
199
138
23.43
0.86

Fig.18. Histogram of task execution of tree strategy

Strategy comparison

As we observed, the *leaf strategy* creates the tasks in the base cases of the recursive implementation, that means that it only creates tasks at leaf level, meanwhile, the *tree strategy* creates the tasks before the calls to merge and multisort functions obtaining a bigger quantity execution of explicit tasks.

If we compare its execution times, we observe that *tree strategy* obtains a better execution time than the *leaf strategy* making the *tree strategy* a more efficient strategy.

Now if we observe their scalability plots, we found that both strategies have strong scalability problems, of these two, the tree strategy has a bit better strong scalability and the speed-up of the multisort region follows closely the line of the ideal strong scalability.

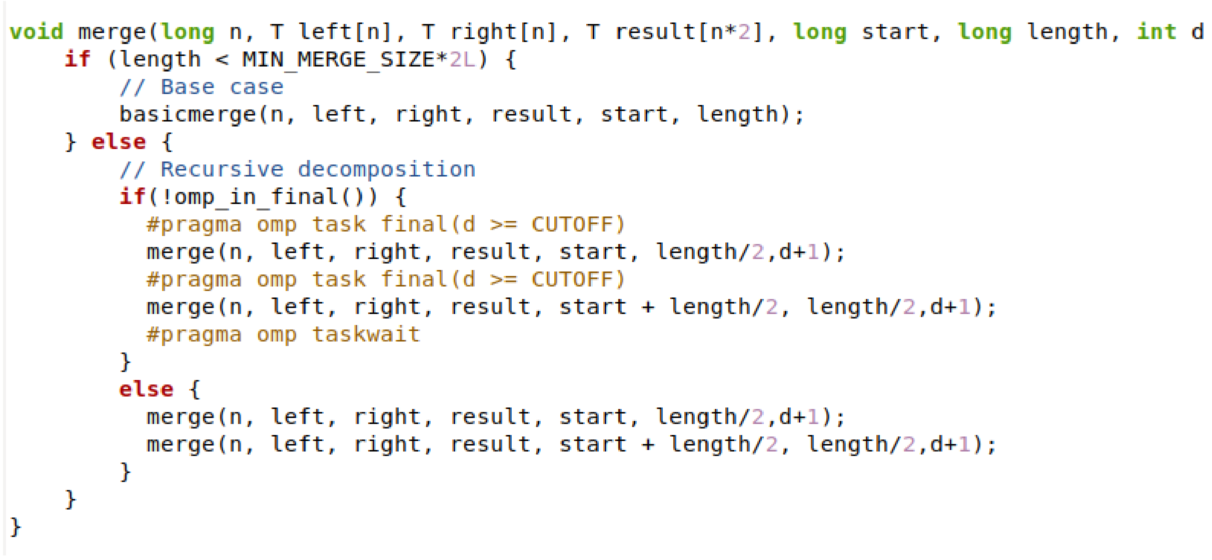
Finally, comparing their performance with modelfactors and paraver we saw that both strategies have efficiency problems, the *leaf strategy* executes less explicit tasks than the *tree strategy*, but it generates bigger overheads as threads are added, on the other hand the overheads of the *tree strategy* are quite minor but it executes much more explicit tasks affecting his performance. In both cases, all the threads have equally distributed the tasks, except by the thread that creates those.

Optimization: Task granularity control: the cut-off mechanism

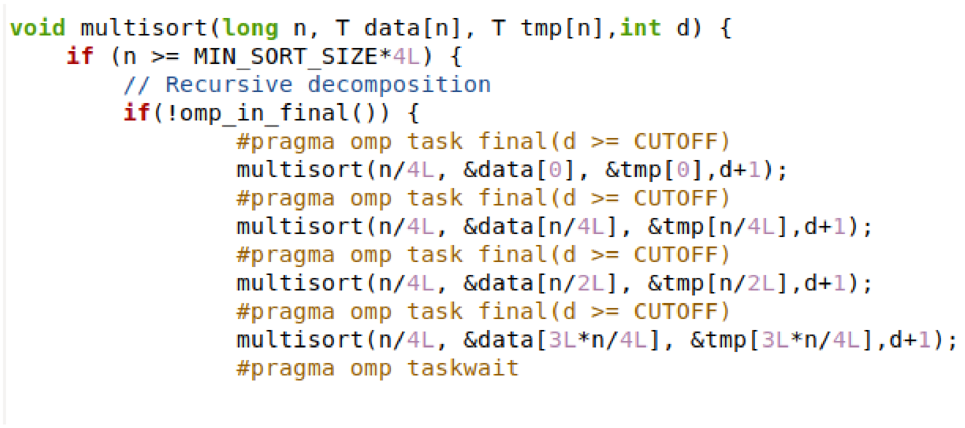
In this section we will control the task granularity of the *tree strategy* implementing a cut-off mechanism.

To implement that, we need to add the 'depth' of the tree as a variable to the *multisort* and *merge functions*, we call it '*d*'. In each recursive call we increment *d* by one, so we are going deeper in the tree. That allows us to know in which tree levels are and stop generating tasks if it passes the parameter specified with **-c** option.

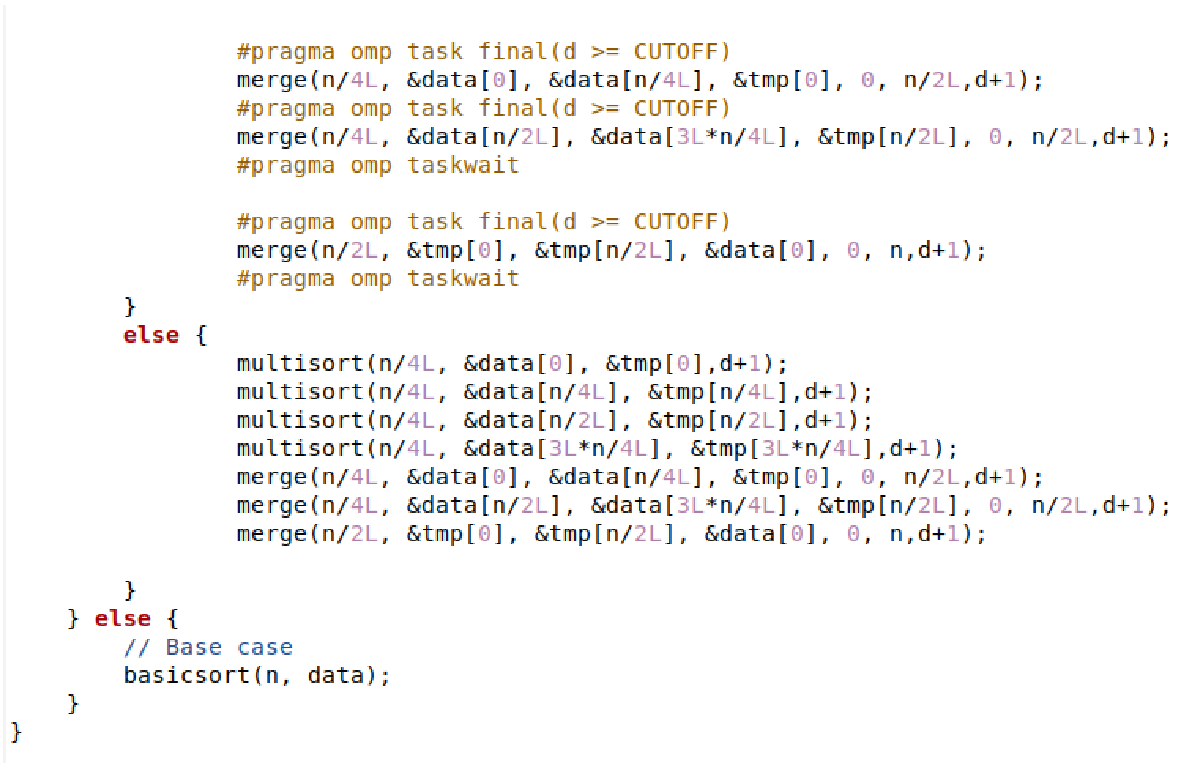
So to control that we added the *#pragma omp task final (d* >= CUTOFF) directive in *multisort* and merge functions and added two *else* for the sequential execution of the cutoff level functions , at Fig.19. it can be observed the implementation of the mechanism explained above.



```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int d
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        if(!omp_in_final()) {
          #pragma omp task final(d >= CUTOFF)
          merge(n, left, right, result, start, length/2,d+1);
          #pragma omp task final(d >= CUTOFF)
          merge(n, left, right, result, start + length/2, length/2,d+1);
          #pragma omp taskwait
        }
        else {
          merge(n, left, right, result, start, length/2,d+1);
          merge(n, left, right, result, start + length/2, length/2,d+1);
        }
    }
}
```



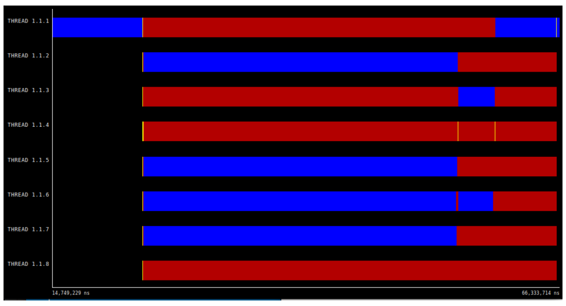
```
void multisort(long n, T data[n], T tmp[n],int d) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        if(!omp_in_final()) {
                #pragma omp task final(d >= CUTOFF)
                multisort(n/4L, &data[0], &tmp[0],d+1);
                #pragma omp task final(d >= CUTOFF)
                multisort(n/4L, &data[n/4L], &tmp[n/4L],d+1);
                #pragma omp task final(d >= CUTOFF)
                multisort(n/4L, &data[n/2L], &tmp[n/2L],d+1);
                #pragma omp task final(d >= CUTOFF)
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],d+1);
                #pragma omp taskwait
```



```
                #pragma omp task final(d >= CUTOFF)
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,d+1);
                #pragma omp task final(d >= CUTOFF)
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,d+1);
                #pragma omp taskwait

                #pragma omp task final(d >= CUTOFF)
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,d+1);
                #pragma omp taskwait
        }
        else {
                multisort(n/4L, &data[0], &tmp[0],d+1);
                multisort(n/4L, &data[n/4L], &tmp[n/4L],d+1);
                multisort(n/4L, &data[n/2L], &tmp[n/2L],d+1);
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],d+1);
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,d+1);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,d+1);
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,d+1);

        }
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Fig.19. Code implementation of cut-off mechanism

Now we modify the submit-strong.extrae.sh so that the cut off is 0 and then 1, then we execute the code with the command sbatch ./submit-strong-extrae.sh multisort-omp



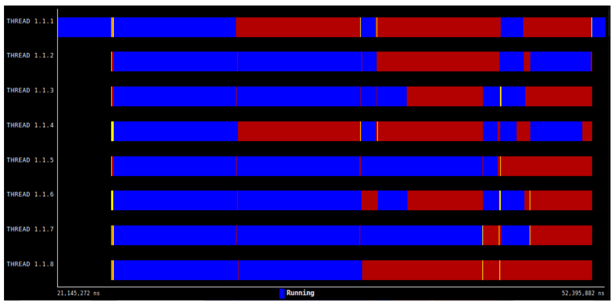
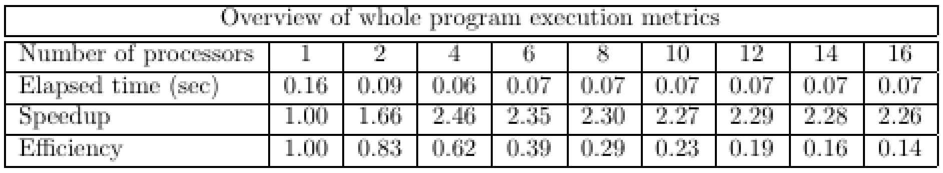


Fig.20. Traces cut-off version 1

Fig.21. Traces cut-off version 2

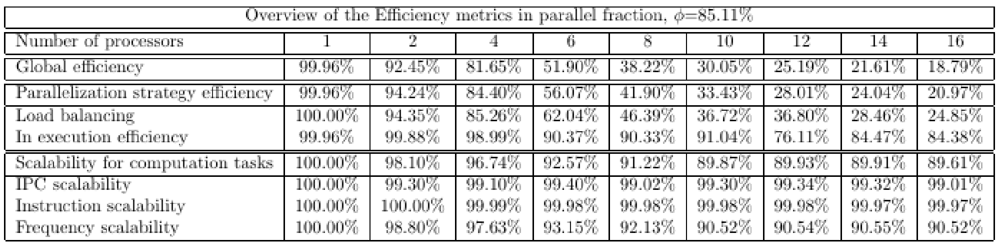
Comparing both versions we see there is a big difference in the time of synchronization. This difference is caused because tasks in the version two are bigger due to the cut-off and hence require less time of synchronization and less overheads.

We can also see differences in the modelfactor tables (Fig.22 and Fig.23).



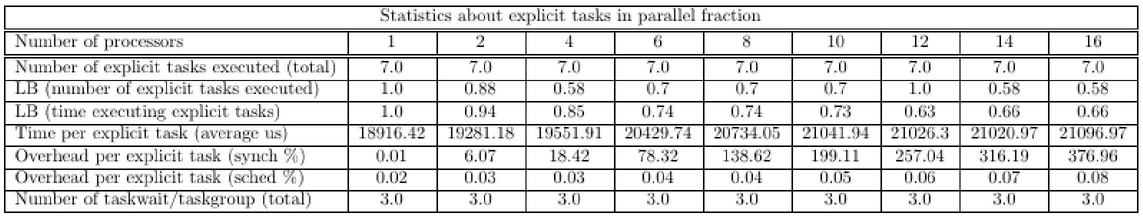
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.07	0.07	0.07	0.07	0.07	0.07
Speedup	1.00	1.66	2.46	2.35	2.30	2.27	2.29	2.28	2.26
Efficiency	1.00	0.83	0.62	0.39	0.29	0.23	0.19	0.16	0.14

Table 1: Analysis done on Mon Nov 28 03:07:20 PM CET 2022, par4107



Overview of the Efficiency metrics in parallel fraction, ϕ=85.11%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.96%	92.45%	81.65%	51.90%	38.22%	30.05%	25.19%	21.61%	18.79%
Parallelization strategy efficiency	99.96%	94.24%	84.40%	56.07%	41.90%	33.43%	28.01%	24.04%	20.97%
Load balancing	100.00%	94.35%	85.26%	62.04%	46.39%	36.72%	36.80%	28.46%	24.85%
In execution efficiency	99.96%	99.88%	98.99%	90.37%	90.33%	91.04%	76.11%	84.47%	84.38%
Scalability for computation tasks	100.00%	98.10%	96.74%	92.57%	91.22%	89.87%	89.93%	89.91%	89.61%
IPC scalability	100.00%	99.30%	99.10%	99.40%	99.02%	99.30%	99.34%	99.32%	99.01%
Instruction scalability	100.00%	100.00%	99.99%	99.98%	99.98%	99.98%	99.98%	99.97%	99.97%
Frequency scalability	100.00%	98.80%	97.63%	93.15%	92.13%	90.52%	90.54%	90.55%	90.52%

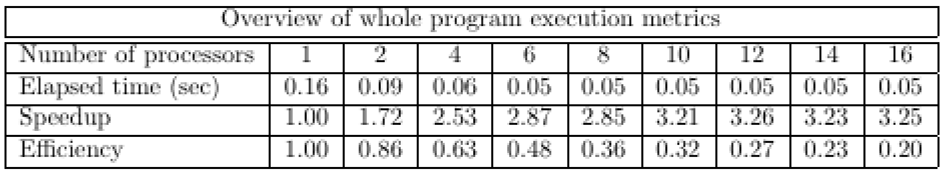
Table 2: Analysis done on Mon Nov 28 03:07:20 PM CET 2022, par4107



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	0.7	0.7	0.7	1.0	0.58	0.58
LB (time executing explicit tasks)	1.0	0.94	0.85	0.74	0.74	0.73	0.63	0.66	0.66
Time per explicit task (average us)	18916.42	19281.18	19551.91	20429.74	20734.05	21041.94	21026.3	21020.97	21096.97
Overhead per explicit task (synch %)	0.01	6.07	18.42	78.32	138.62	199.11	257.04	316.19	376.96
Overhead per explicit task (sched %)	0.02	0.03	0.03	0.04	0.04	0.05	0.06	0.07	0.08
Number of taskwait/taskgroup (total)	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0

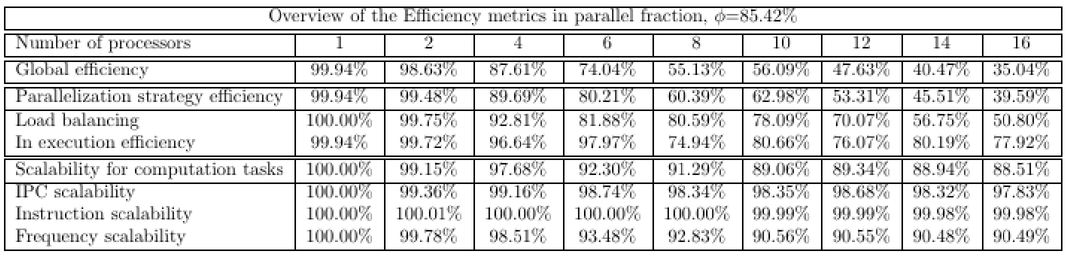
Table 3: Analysis done on Mon Nov 28 03:07:20 PM CET 2022, par4107

Fig.22. Modelfactor tables of version 1



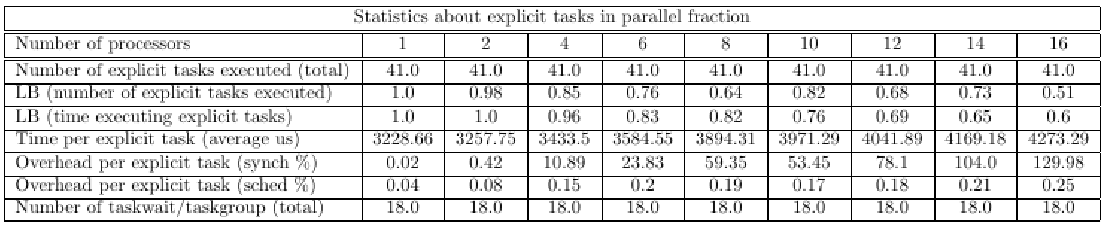
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.05	0.05	0.05	0.05	0.05	0.05
Speedup	1.00	1.72	2.53	2.87	2.85	3.21	3.26	3.23	3.25
Efficiency	1.00	0.86	0.63	0.48	0.36	0.32	0.27	0.23	0.20

Table 1: Analysis done on Mon Nov 28 03:09:56 PM CET 2022, par4107



Overview of the Efficiency metrics in parallel fraction, ϕ=85.42%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.94%	98.63%	87.61%	74.04%	55.13%	56.09%	47.63%	40.47%	35.04%
Parallelization strategy efficiency	99.94%	99.48%	89.69%	80.21%	60.39%	62.98%	53.31%	45.51%	39.59%
Load balancing	100.00%	99.75%	92.81%	81.88%	80.59%	78.09%	70.07%	56.75%	50.80%
In execution efficiency	99.94%	99.72%	96.64%	97.97%	74.94%	80.66%	76.07%	80.19%	77.92%
Scalability for computation tasks	100.00%	99.15%	97.68%	92.30%	91.29%	89.06%	89.34%	88.94%	88.51%
IPC scalability	100.00%	99.36%	99.16%	98.74%	98.34%	98.35%	98.68%	98.32%	97.83%
Instruction scalability	100.00%	100.01%	100.00%	100.00%	100.00%	99.99%	99.99%	99.98%	99.98%
Frequency scalability	100.00%	99.78%	98.51%	93.48%	92.83%	90.56%	90.55%	90.48%	90.49%

Table 2: Analysis done on Mon Nov 28 03:09:56 PM CET 2022, par4107



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	0.76	0.64	0.82	0.68	0.73	0.51
LB (time executing explicit tasks)	1.0	1.0	0.96	0.83	0.82	0.76	0.69	0.65	0.6
Time per explicit task (average us)	3228.66	3257.75	3433.5	3584.55	3894.31	3971.29	4041.89	4169.18	4273.29
Overhead per explicit task (synch %)	0.02	0.42	10.89	23.83	59.35	53.45	78.1	104.0	129.98
Overhead per explicit task (sched %)	0.04	0.08	0.15	0.2	0.19	0.17	0.18	0.21	0.25
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 Mon Nov 28 03:09:56 PM CET 2022, par4107

Fig.23. Modelfactor tables of version 2

If we compare both tables (Fig.22 and Fig.23) we can note that the elapsed time and the overheads have slightly reduced, on the other hand the load balancing has augmented from *version 1* to *version 2*.

We also notice that the number of explicit tasks has increased due to the cut-off level, that is because in the tree strategy we create tasks for each recursive call, that means that with a higher cut-off value we move deeper on the tree, generating more recursive calls and also tasks.

Once analyzed the impact of the cut-off, we can now explore what is the ideal value for that cut-off and if it depends on the number of threads the program has access to.

We can compare the execution time of different cut-offs using 8 and 16 threads by using the following commands:
sbatch ./submit-cutoff-omp.sh 8
sbatch ./submit-cutoff-omp.sh 16

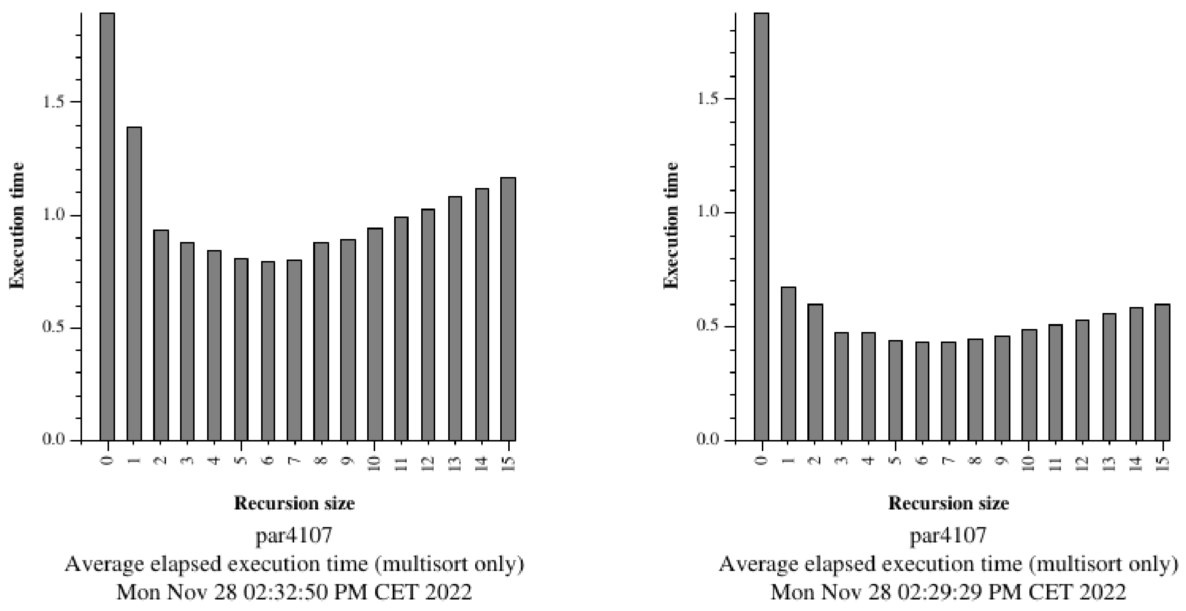


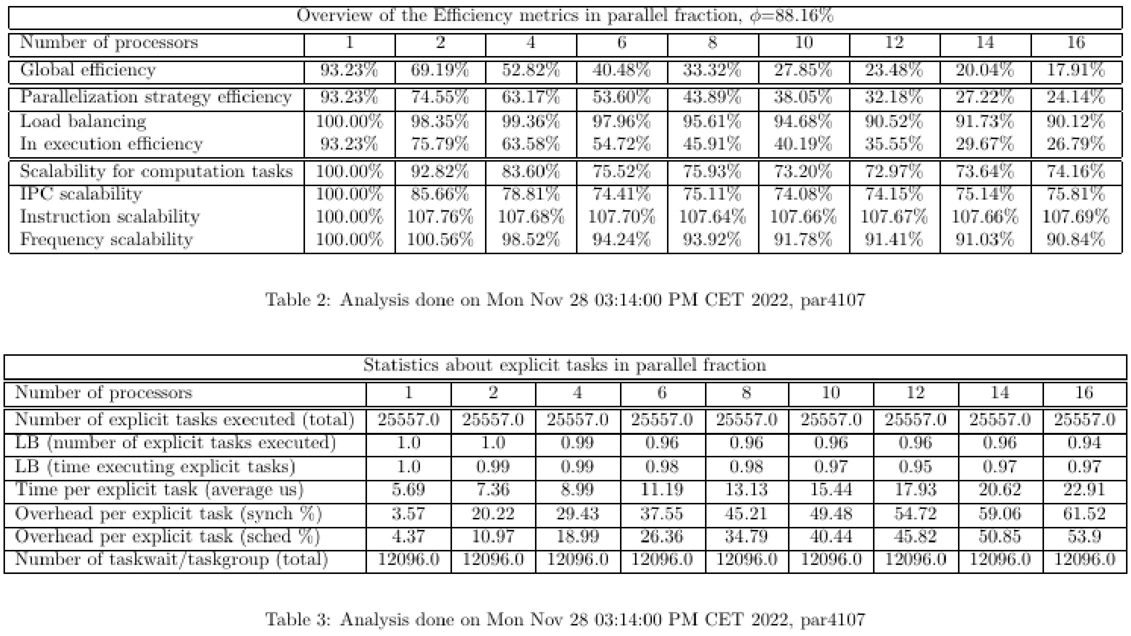
Fig.24. Execution time with different cut-offs (8 and 16 threads respectively)

If we observe the plots generated (Fig.24) we can see that the execution times in the version of 16 threads are significantly lower than the version of 8 threads (in exception to the cutoff 0). But in the other hand we also observe the same behavior in both tables as the execution time decreases to the optimal values for the cut off (5,6 or 7) and then increases as the cut-off value augments, so we can argue that the number of threads doesn't affect the optimal values of for the cut-off.

Here we can see the modelfactor tables of one of the optimal cut-off values (Fig.25), where the number of tasks has notably increased.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.19	0.14	0.10	0.09	0.08	0.08	0.08	0.08	0.08
Speedup	1.00	1.40	1.97	2.16	2.30	2.37	2.38	2.38	2.41
Efficiency	1.00	0.70	0.49	0.36	0.29	0.24	0.20	0.17	0.15

Table 1: Analysis done on Mon Nov 28 03:14:00 PM CET 2022, par4107



Overview of the Efficiency metrics in parallel fraction, ϕ=88.16%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	93.23%	69.19%	52.82%	40.48%	33.32%	27.85%	23.48%	20.04%	17.91%
Parallelization strategy efficiency	93.23%	74.55%	63.17%	53.60%	43.89%	38.05%	32.18%	27.22%	24.14%
Load balancing	100.00%	98.35%	99.36%	97.96%	95.61%	94.68%	90.52%	91.73%	90.12%
In execution efficiency	93.23%	75.79%	63.58%	54.72%	45.91%	40.19%	35.55%	29.67%	26.79%
Scalability for computation tasks	100.00%	92.82%	83.60%	75.52%	75.93%	73.20%	72.97%	73.64%	74.16%
IPC scalability	100.00%	85.66%	78.81%	74.41%	75.11%	74.08%	74.15%	75.14%	75.81%
Instruction scalability	100.00%	107.76%	107.68%	107.70%	107.64%	107.66%	107.67%	107.66%	107.69%
Frequency scalability	100.00%	100.56%	98.52%	94.24%	93.92%	91.78%	91.41%	91.03%	90.84%

Table 2: Analysis done on Mon Nov 28 03:14:00 PM CET 2022, par4107

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)	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0
LB (number of explicit tasks executed)	1.0	1.0	0.99	0.96	0.96	0.96	0.96	0.96	0.94
LB (time executing explicit tasks)	1.0	0.99	0.99	0.98	0.98	0.97	0.95	0.97	0.97
Time per explicit task (average us)	5.69	7.36	8.99	11.19	13.13	15.44	17.93	20.62	22.91
Overhead per explicit task (synch %)	3.57	20.22	29.43	37.55	45.21	49.48	54.72	59.06	61.52
Overhead per explicit task (sched %)	4.37	10.97	18.99	26.36	34.79	40.44	45.82	50.85	53.9
Number of taskwait/taskgroup (total)	12096.0	12096.0	12096.0	12096.0	12096.0	12096.0	12096.0	12096.0	12096.0

Table 3: Analysis done on Mon Nov 28 03:14:00 PM CET 2022, par4107

Fig.25. Modelfactor tables for 6 cut-off value

Optional 1

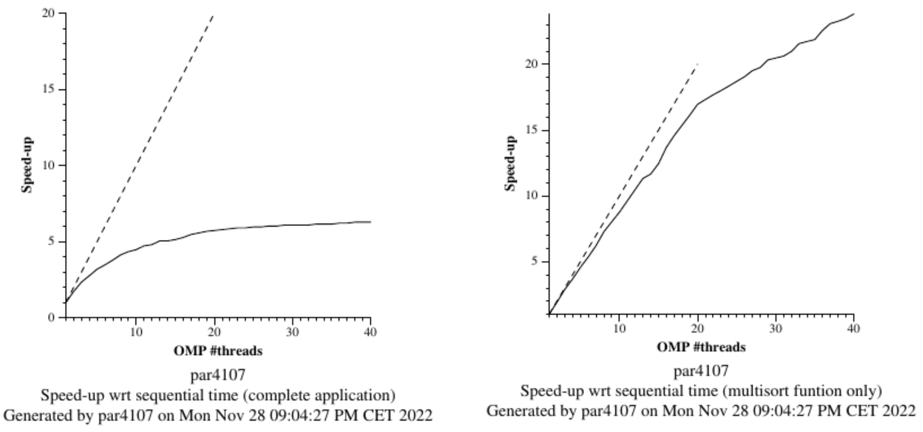
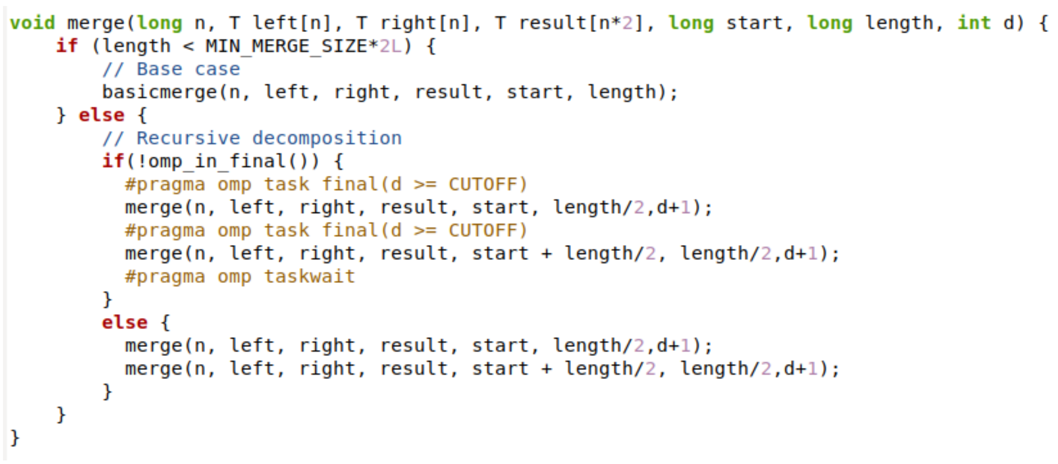


Fig.26. Speed-up for 6 cut-off with threads from 1 to 40

We executed the strong analisis using the cutoff that we obtained before (cut-off 6) and with 40 threads that exceed the number of physical threads that we have in boada. The performance is still growing even after exceeding the available physical threads, we think that this is caused by hyperthreading obtaining the better performance seen on the plots.

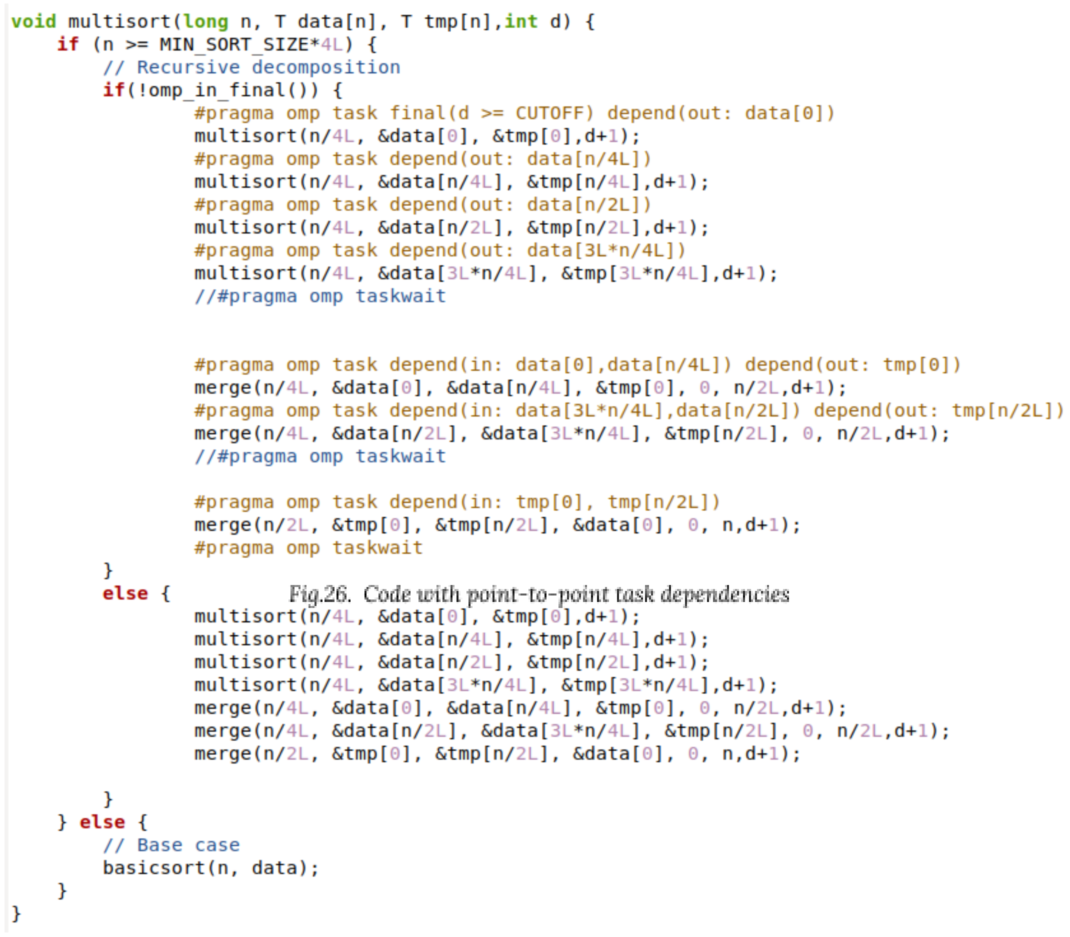
Session 3: Shared-memory parallelisation with OpenMP task using dependencies

We now modify our tree version strategy that has the cut-off mechanism, by replacing the redundant *taskwaits* synchronisations by point-to-point task dependencies, the next code fragment shows the changes of it.



```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int d) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        if(!omp_in_final()) {
          #pragma omp task final(d >= CUTOFF)
          merge(n, left, right, result, start, length/2,d+1);
          #pragma omp task final(d >= CUTOFF)
          merge(n, left, right, result, start + length/2, length/2,d+1);
          #pragma omp taskwait
        }
        else {
          merge(n, left, right, result, start, length/2,d+1);
          merge(n, left, right, result, start + length/2, length/2,d+1);
        }
    }
}
```

Fig.26. Code with point-to-point task dependencies



```
void multisort(long n, T data[n], T tmp[n],int d) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        if(!omp_in_final()) {
                #pragma omp task final(d >= CUTOFF) depend(out: data[0])
                multisort(n/4L, &data[0], &tmp[0],d+1);
                #pragma omp task depend(out: data[n/4L])
                multisort(n/4L, &data[n/4L], &tmp[n/4L],d+1);
                #pragma omp task depend(out: data[n/2L])
                multisort(n/4L, &data[n/2L], &tmp[n/2L],d+1);
                #pragma omp task depend(out: data[3L*n/4L])
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],d+1);
                //#pragma omp taskwait


                #pragma omp task depend(in: data[0],data[n/4L]) depend(out: tmp[0])
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,d+1);
                #pragma omp task depend(in: data[3L*n/4L],data[n/2L]) depend(out: tmp[n/2L])
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,d+1);
                //#pragma omp taskwait

                #pragma omp task depend(in: tmp[0], tmp[n/2L])
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,d+1);
                #pragma omp taskwait
        }
        else {
                multisort(n/4L, &data[0], &tmp[0],d+1);
                multisort(n/4L, &data[n/4L], &tmp[n/4L],d+1);
                multisort(n/4L, &data[n/2L], &tmp[n/2L],d+1);
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],d+1);
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,d+1);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,d+1);
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,d+1);

        }
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Fig.27. Code of the new dependencies version

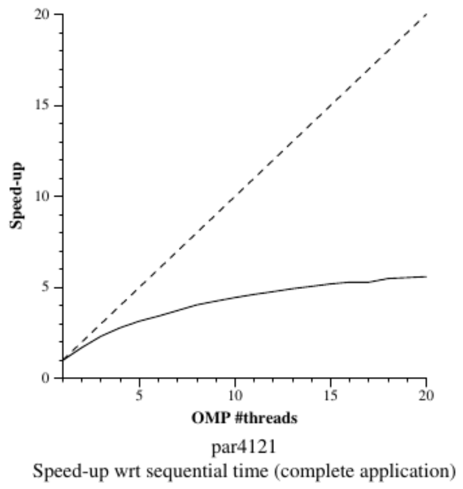
Once we have implemented the different dependencies into our cut-off tree strategy, we executed it to ensure that works correctly obtaining the next result that confirm it:

```
::::::::::::::
multisort-omp_8_boada-11.times.txt
::::::::::::::
*******************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                  CUTOFF=6
Number of threads in OpenMP:        OMP_NUM_THREADS=8
*******************************************************************************************
Initialization time in seconds: 0.688007
Multisort execution time: 0.704276
Check sorted data execution time: 0.012442
Multisort program finished
*******************************************************************************************
```

Comparing the execution times of the other versions that we implemented in the previous sessions, this strategy obtains the best execution time; it seems to be the one with the best performance due the times obtained.

If we take a look at its scalability, it can be seen in the plots of Fig.28., and comparing with the previous version, we can observe that their scalability is very similar, however the new version seems to approach more to the ideal scalability line. So in terms of performance it seems that both the tree strategy and the cut-off strategy are highly similar to the new version.

If we talk of the programmability of the new version, implementing the dependencies point to point helps to the programmer to have a better understanding of the data flow among the execution and where the task must be wait to other ones, however we think that it's easier to implement the task wait or task group version, because lets the programmer forget about it.



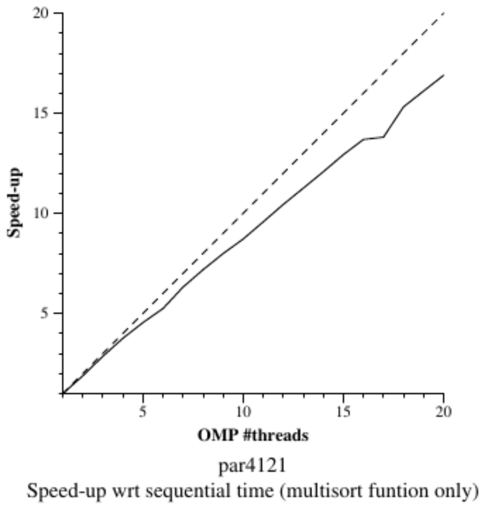
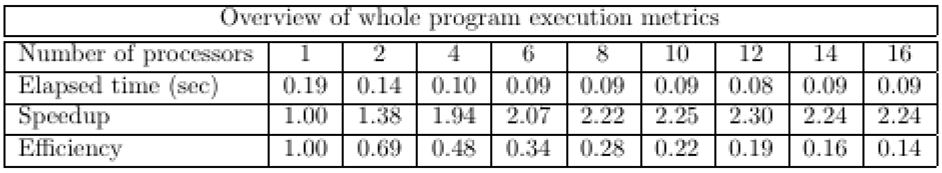


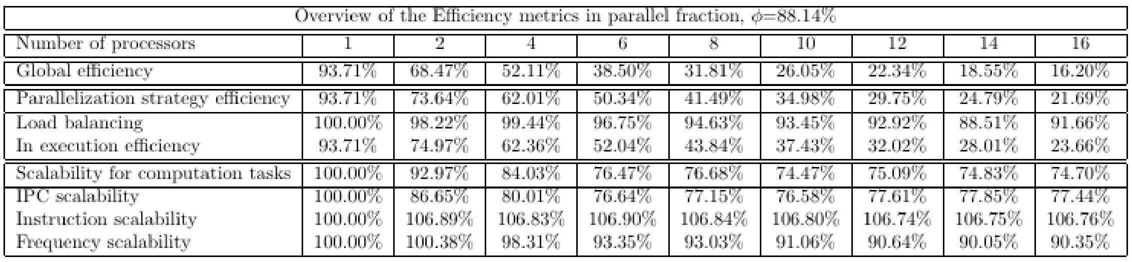
Fig.28. Scalability plots of the dependencies version

We executed the modelfactor analysis of the new version obtaining the next table results:



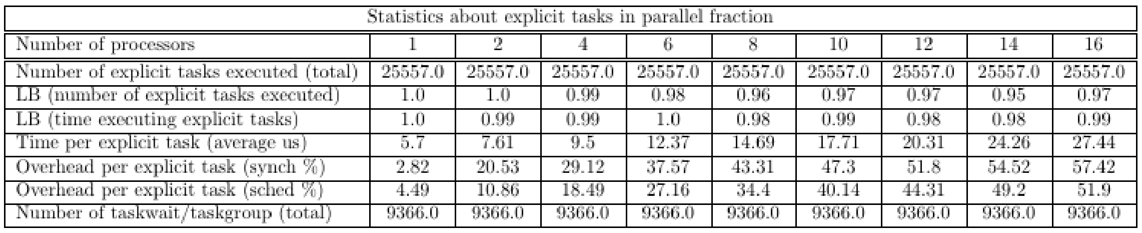
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.19	0.14	0.10	0.09	0.09	0.09	0.08	0.09	0.09
Speedup	1.00	1.38	1.94	2.07	2.22	2.25	2.30	2.24	2.24
Efficiency	1.00	0.69	0.48	0.34	0.28	0.22	0.19	0.16	0.14

Table 1: Analysis done on Tue Nov 29 10:49:30 AM CET 2022, par4107



Overview of the Efficiency metrics in parallel fraction, ϕ=88.14%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	93.71%	68.47%	52.11%	38.50%	31.81%	26.05%	22.34%	18.55%	16.20%
Parallelization strategy efficiency	93.71%	73.64%	62.01%	50.34%	41.49%	34.98%	29.75%	24.79%	21.69%
Load balancing	100.00%	98.22%	99.44%	96.75%	94.63%	93.45%	92.92%	88.51%	91.66%
In execution efficiency	93.71%	74.97%	62.36%	52.04%	43.84%	37.43%	32.02%	28.01%	23.66%
Scalability for computation tasks	100.00%	92.97%	84.03%	76.47%	76.68%	74.47%	75.09%	74.83%	74.70%
IPC scalability	100.00%	86.65%	80.01%	76.64%	77.15%	76.58%	77.61%	77.85%	77.44%
Instruction scalability	100.00%	106.89%	106.83%	106.90%	106.84%	106.80%	106.74%	106.75%	106.76%
Frequency scalability	100.00%	100.38%	98.31%	93.35%	93.03%	91.06%	90.64%	90.05%	90.35%

Table 2: Analysis done on Tue Nov 29 10:49:30 AM CET 2022, par4107



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)	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0	25557.0
LB (number of explicit tasks executed)	1.0	1.0	0.99	0.98	0.96	0.97	0.97	0.95	0.97
LB (time executing explicit tasks)	1.0	0.99	0.99	1.0	0.98	0.99	0.98	0.98	0.99
Time per explicit task (average us)	5.7	7.61	9.5	12.37	14.69	17.71	20.31	24.26	27.44
Overhead per explicit task (synch %)	2.82	20.53	29.12	37.57	43.31	47.3	51.8	54.52	57.42
Overhead per explicit task (sched %)	4.49	10.86	18.49	27.16	34.4	40.14	44.31	49.2	51.9
Number of taskwait/taskgroup (total)	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0	9366.0

Table 3: Analysis done on Tue Nov 29 10:49:30 AM CET 2022, par4107

Fig.29. Modelfactor tables of the new version

Comparing the results obtained from modelfactors analysis with the other versions, we have observed that the results are quite similar to the results obtained with the cut-off version with insignificant variations on their values, as expected the number of *taskwait/taskgruop* of the new version are lower than the cutt-off version.

If we compare it with the *tree version* the results are highly better as they are similar to the cut-off version.

Finally, we execute the Paraver analysis obtaining the different traces below:

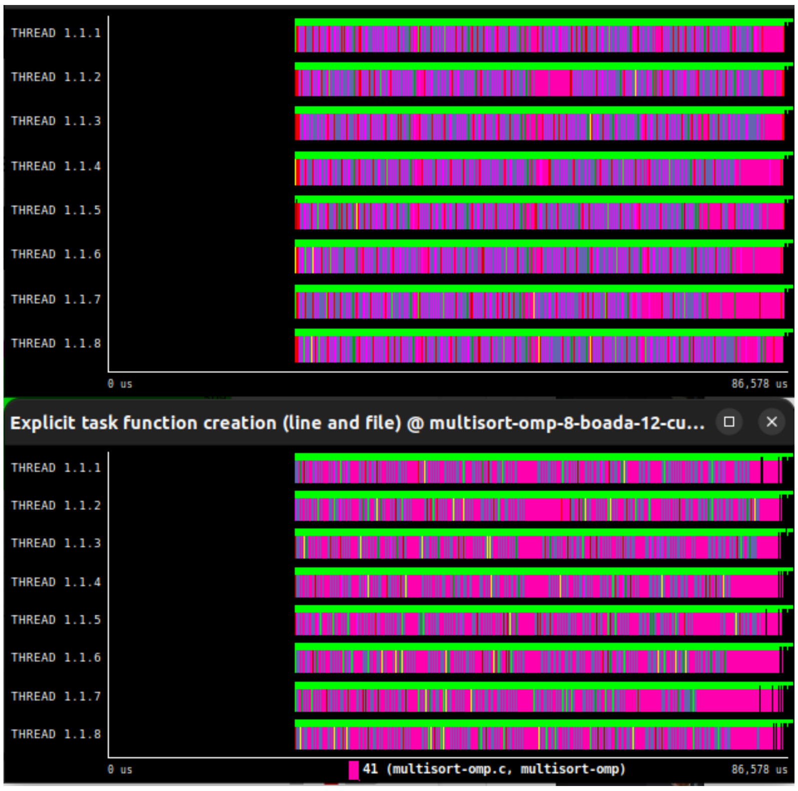


Fig.30. Trace of tasks creation and execution of tree strategy

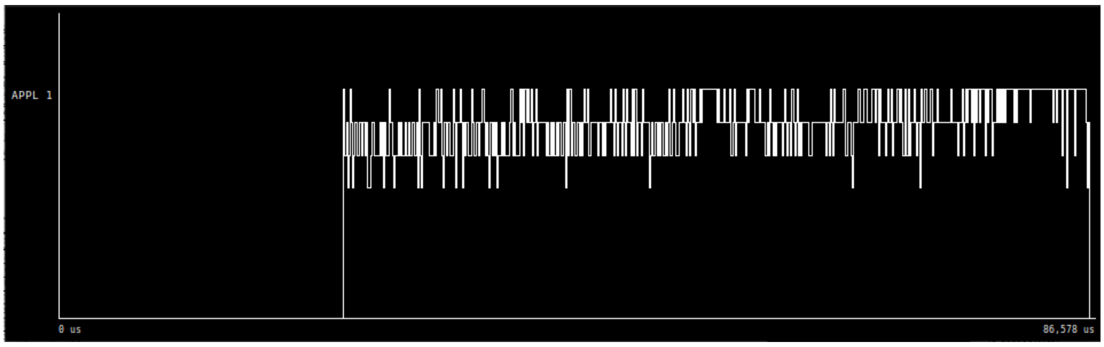
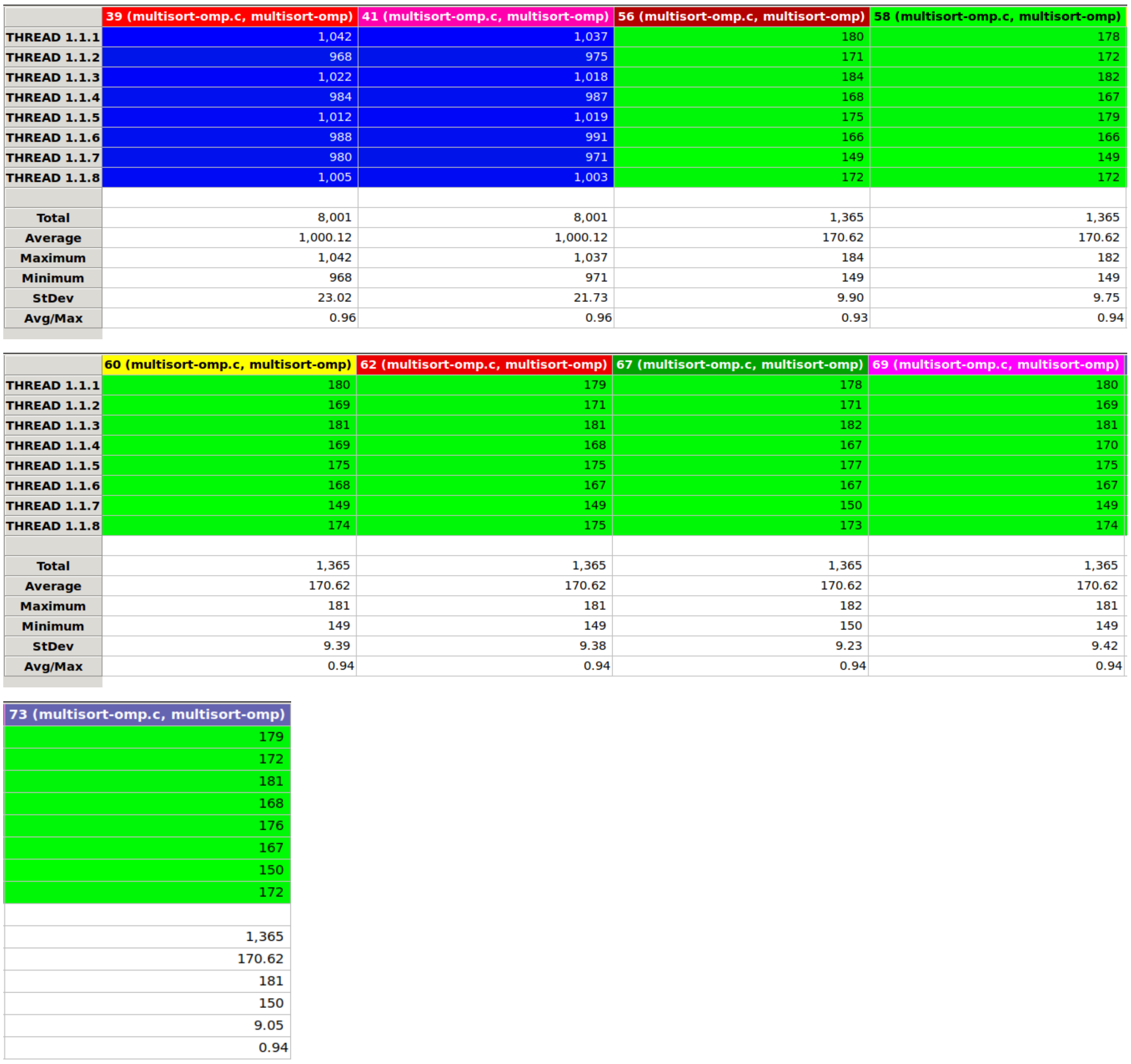


Fig.31. Trace of instantaneous parallelisation of tree strategy

As for the performance, the traces obtained are quite similar to the other versions but with some remarkable differences. The task creation and execution although being similar to the tree version (Fig.17.) the amount of time of the execution is highly lower due to the cut-off implementation.

We also can observe that there are more tasks executed at the same time than in the tree strategy version comparing the traces in Fig.16 and Fig.31.



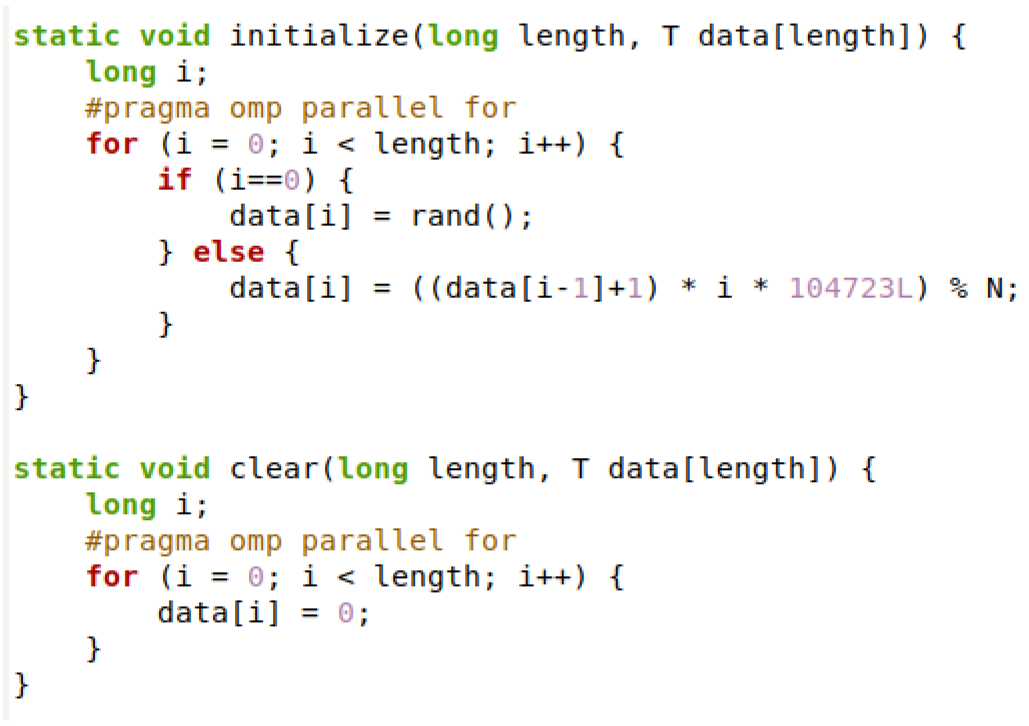
	39 (multisort-omp.c, multisort-omp)	41 (multisort-omp.c, multisort-omp)	56 (multisort-omp.c, multisort-omp)	58 (multisort-omp.c, multisort-omp)	60 (multisort-omp.c, multisort-omp)	62 (multisort-omp.c, multisort-omp)	67 (multisort-omp.c, multisort-omp)	69 (multisort-omp.c, multisort-omp)	73 (multisort-omp.c, multisort-omp)
THREAD 1.1.1	1,042	1,037	180	178	180	179	178	180	179
THREAD 1.1.2	968	975	171	172	169	171	171	169	172
THREAD 1.1.3	1,022	1,018	184	182	181	181	182	181	181
THREAD 1.1.4	984	987	168	167	169	168	167	170	168
THREAD 1.1.5	1,012	1,019	175	179	175	175	177	175	176
THREAD 1.1.6	988	991	166	166	168	167	167	167	167
THREAD 1.1.7	980	971	149	149	149	149	150	149	150
THREAD 1.1.8	1,005	1,003	172	172	174	175	173	174	172
Total	8,001	8,001	1,365	1,365	1,365	1,365	1,365	1,365	1,365
Average	1,000.12	1,000.12	170.62	170.62	170.62	170.62	170.62	170.62	170.62
Maximum	1,042	1,037	184	182	181	181	182	181	181
Minimum	968	971	149	149	149	149	150	149	150
StDev	23.02	21.73	9.90	9.75	9.39	9.38	9.23	9.42	9.05
Avg/Max	0.96	0.96	0.93	0.94	0.94	0.94	0.94	0.94	0.94

Fig.32. Histogram of task execution of dependencies version

Observing the histogram of the task execution of the new version, we can conclude that the tasks are well balanced among threads, so the executed number of tasks are highly equal on all threads.

Optional 2

In this part we will parallelize the functions that initialize the data and the tmp vectors, by adding the *#pragma omp parallel for* directive.



```
static void initialize(long length, T data[length]) {
    long i;
    #pragma omp parallel for
    for (i = 0; i < length; i++) {
        if (i==0) {
            data[i] = rand();
        } else {
            data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
    }
}

static void clear(long length, T data[length]) {
    long i;
    #pragma omp parallel for
    for (i = 0; i < length; i++) {
        data[i] = 0;
    }
}
```

Fig.33. Code with parallel directives

If we take a look at the scalability plots of this version (Fig.34) and compare them with the last ones (Fig.28) we see an enormous difference in the plot that reflects the complete application performance and not that much in the one that reflects the multisort performance.

That's because with this version we have augmented the performance of the data processing functions that don't interact with the *multisort* part of the application and were worsening the performance of the complete application.

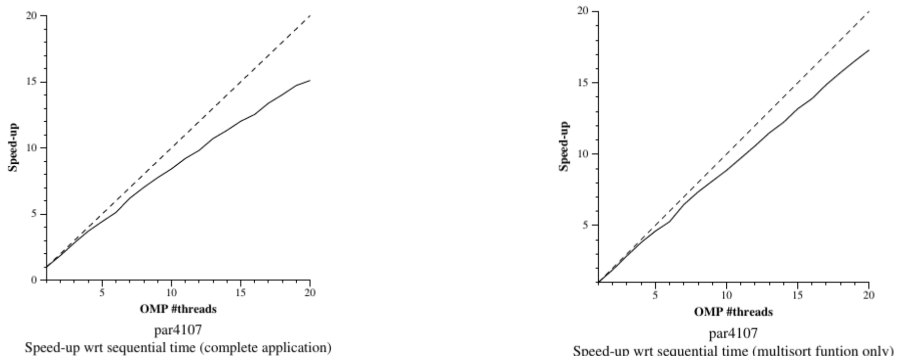


Fig.34. Scalability plots

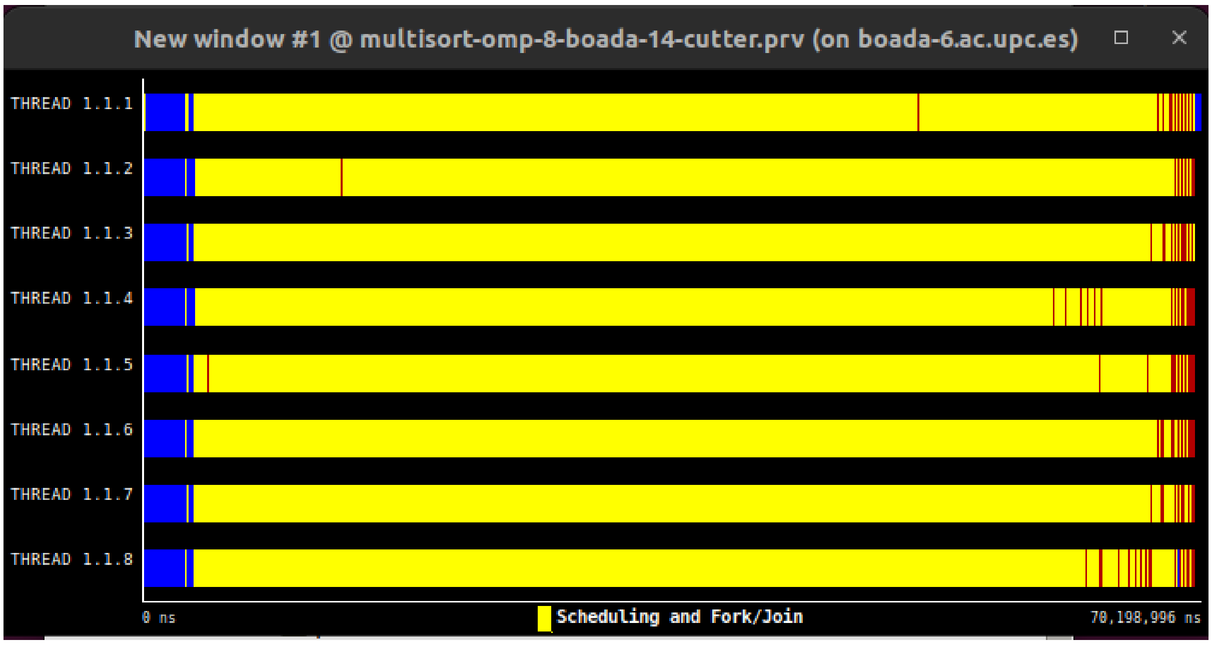


Fig.35. Paraver timelines

Finally if we take a look at the complete timeline of the program, we can see that there are blue lines (running regions) in all threads meaning that the initialization of data is distributed among all threads making the code more parallel and efficient.