PAR Laboratory Assignment 4

Index

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

In this laboratory assignment we explore the use of parallelism in recursive algorithms. In this case, we work with the *Merge Sort* algorithm which we have seen in previous courses. This algorithm divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves using the function `merge`. We are provided with the file *multisort.c* and the file *multisort-tareador.c* to work on the next three sessions.

Task decomposition analysis for Merge Sort

In this first session we work with the *multisort-tareador.c* file and we investigate, using the *Tareador* tool, the potential parallelism problems that the code has. These problems will be dealt with using OMP directives that we will have to take into account in the following sessions.

"Divide and conquer"

First of all, we analyze the provided code on *multisort.c* and we make sure we understand the `merge` and `multisort` functions that form the *Merge Sort* algorithm. Then, to see an example of an execution of this code, we compile and execute a sequential version of the code (which is provided in the executable file *multisort-seq*). Executing using `./multisort-seq -n 32768 -s 1024 -m 1024` (these values are the default values when it is unspecified in each of the options respectively), we get the following output:

```
********************************************************************************Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024

********************************************************************************Initialization time in seconds: 0.855622

Multisort execution time: 6.262741

Check sorted data execution time: 0.016487

Multisort program finished

********************************************************************************
```

We will have the times that we got in the previous output in mind when we analyze the scalability of the parallel versions in the following sections.

Task decomposition analysis with Tareador

Now, we investigate the *multisort-tareador.c* code on the potential task decomposition strategies and their implications in terms of parallelism and task interaction required, using the *Tareador* tool.
We modify the code to define tasks for each of the recursive task decomposition strategies:

- *Leaf* : In this strategy we define the tasks when the `basicsort` and `basicmerge` functions are invoked.

```
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("BasicMerge");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("BasicMerge");
    } 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("BasicSort");
        basicsort(n, data);
        tareador_end_task("BasicSort");
    }
}
```

➢ *Tree* : In this strategy we define a task for each of the invocations of `multisort` and `merge`.

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

The next step is to compile and execute each of the strategies with *Tareador*. We compile using `make multisort-tareador`, which is the target for the file that we modified, and we execute it using the *run-tareador.sh* script.
The following task dependence graph belongs to the *leaf* strategy:

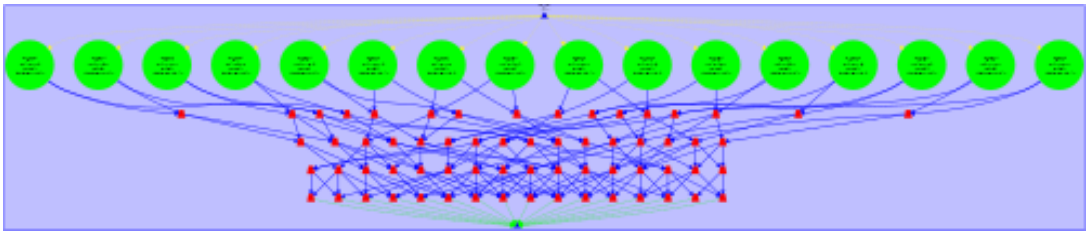


fig. 1: Task dependence graph *leaf* strategy

The next one is the *tree* strategy task dependence graph:

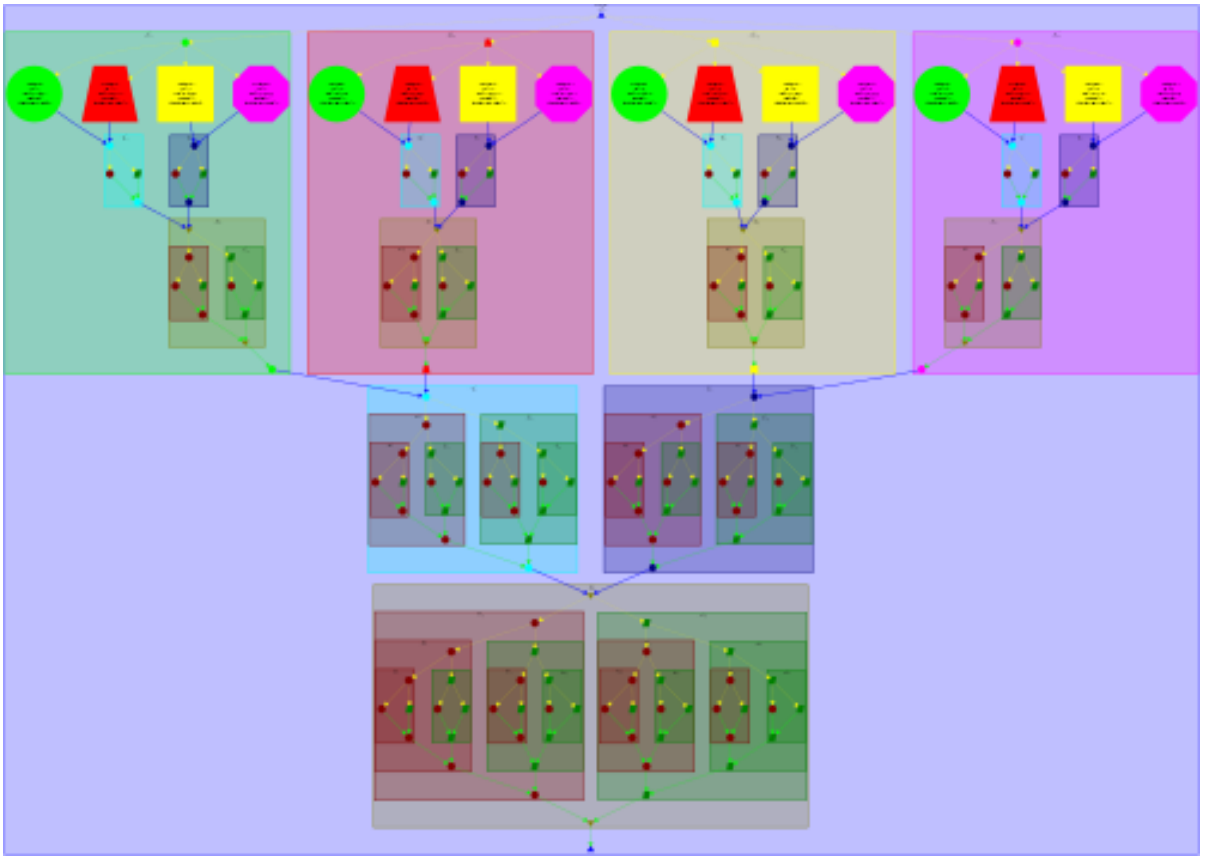


fig. 2: Task dependence graph *tree* strategy

As we can see the 2 task dependence graphs are very different for each strategy. Obviously, the *tree* strategy has a lot more tasks generated because there are a lot more invocations for the `merge` and `multisort` functions than the `basicsort` and `basicmerge`. The structure of the graph varies a bit because in the *leaf* strategy we only generate tasks in the base case of the functions `merge` and `multisort` rather than the recursive case which is what we do in the *tree* strategy. In terms of granularity, in the *leaf* strategy we see a big imbalance between the task that represents `basicsort` (shown in green, see fig. 1) and `basicmerge` (shown in red, see fig. 1). On the other hand, in the *tree* strategy we also see a big difference between task granularities, specifically between the first `multisort` tasks (the 4 of them, one for invocation.

Shown in bright green, red, yellow and magenta on the top of fig. 2) and the `merge` and `multisort` tasks that are done below these 4 for each branch. We can see the tasks representing `multisort` don't create dependencies because they don't rely on the main vector as `merge` does. As we analyze the code we can see the parts that we can parallelize using different omp directives for both of the strategies that we have seen. This is something that we will see further explained in the following sessions when modifying the *multisort.c* code, but we can already see where we will have to create the explicit tasks (*#pragma omp task*) and where we will have to synchronize the tasks (using *#pragma omp taskwait* in this case) for the *leaf* and *tree* strategies.

Finally, we simulate the execution of each strategy with 16 processors using *Paraver*. The following images are zoomed parts of the traces of the *leaf* and *tree* strategies respectively:

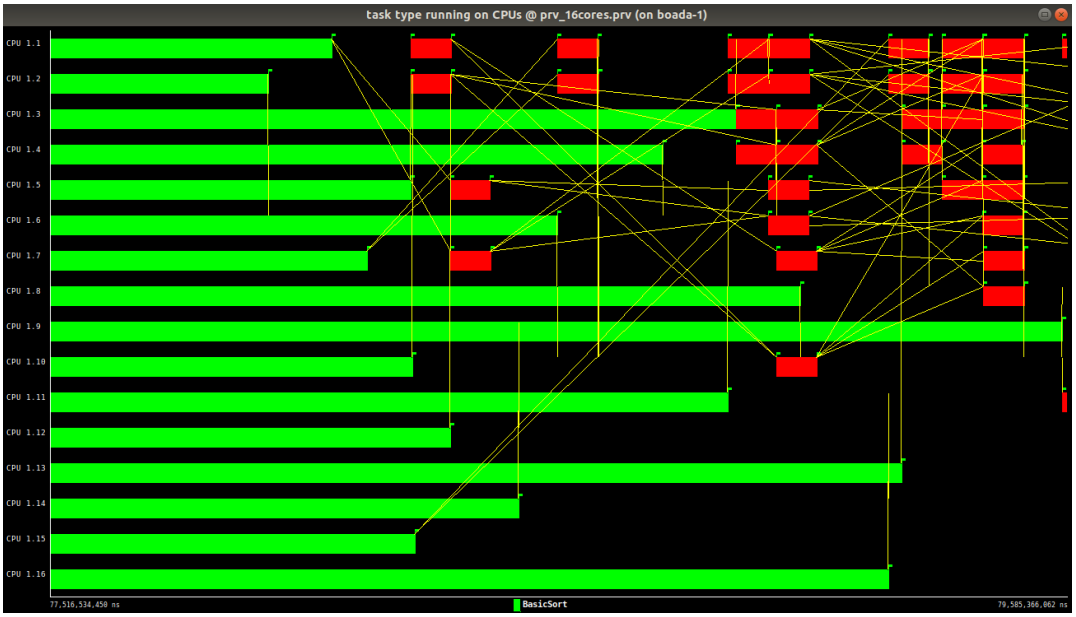


fig. 3: *Paraver* trace *leaf* strategy

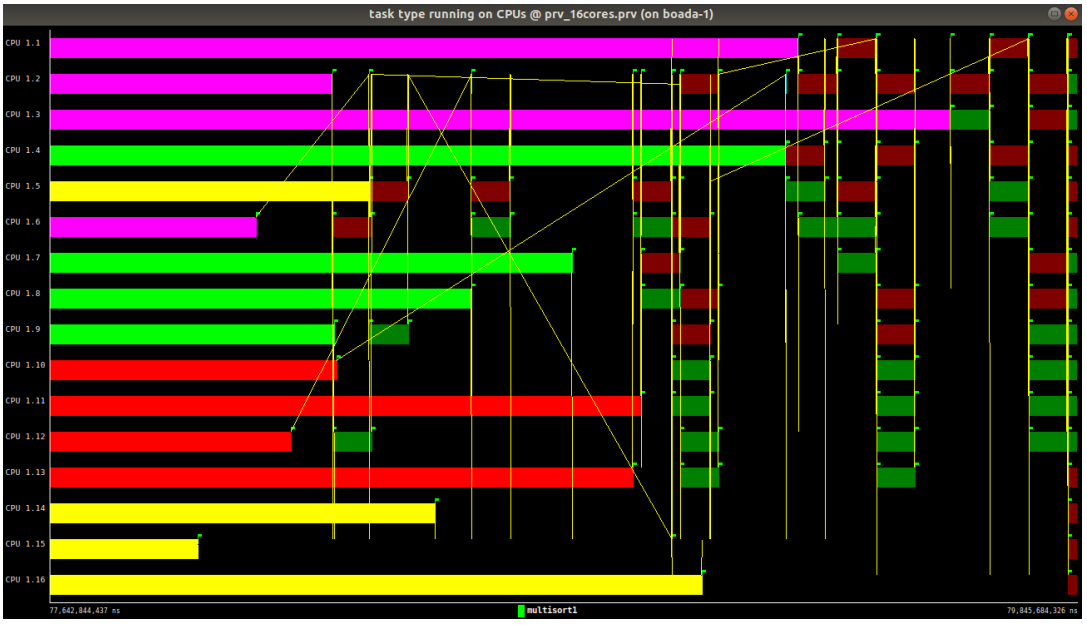


fig. 4: *Paraver* trace *tree* strategy

As we can see in the plots and the graphs, in both cases, `merge` is the function that creates dependencies because it is the one that gathers the sorted halves. On the other hand, `multisort`, since the function doesn't depend on the divisions of the original vector, does not create dependencies. We can clearly see in the previous plots (see fig. 3 and fig. 4) that the `multisort` function needs a lot more time of the trace because a lot more tasks are created.

Shared-memory parallelisation with OpenMP tasks

In this section we work with the *multisort.c* file and we investigate two parallelization approaches. We will start with the leaf strategy and afterwards we will implement the tree strategy.

Leaf strategy in OpenMP

- *Leaf:* In order to correctly implement the leaf strategy, we create, in the base case, the tasks that execute the *basicsort* and the *basicmerge* and using the Open MP directive *#pragma omp taskwait*, and with it, we synchronize the different threads, after making the calls recursive to the multisort function and to execute the two calls to the merge function.

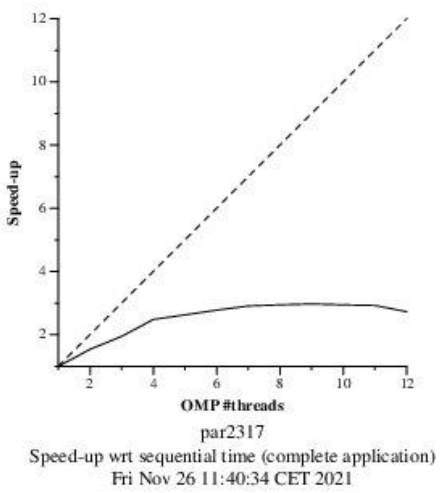
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long
length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        #pragma omp task
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        merge(n, left,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

With the code already modified, using the sbatch command, we execute it several times to check there are no errors in the execution and we obtain the following output:

multisort-omp_2_boada-3.times.txt

```
************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************
Initialization time in seconds: 0.855672
Multisort execution time: 3.747875
Check sorted data execution time: 0.016509
Multisort program finished
************************************************************************
```

In the output we can observe a great improvement in the execution time, going from `6.262741s` to `3.747875s`.



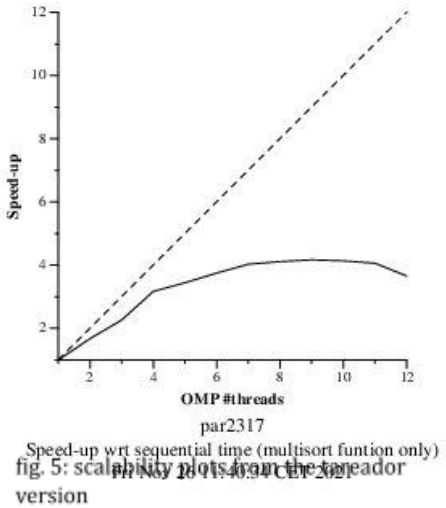


fig. 5: scalability plots from the tareador version

Regarding the scalability of this version, we can see in the plots (see fig. 5), how the speedup is far from being ideal as the number of threads increases. We find a sharp change from thread 4 in the improvement slope, and it drops drastically and even decreases after 11.

Since this solution does not scale satisfactorily, we analyze the execution using the tool *paraver*. We will use the submit-extrae.sh script for 8 processors, which will trace the execution of the parallel execution.

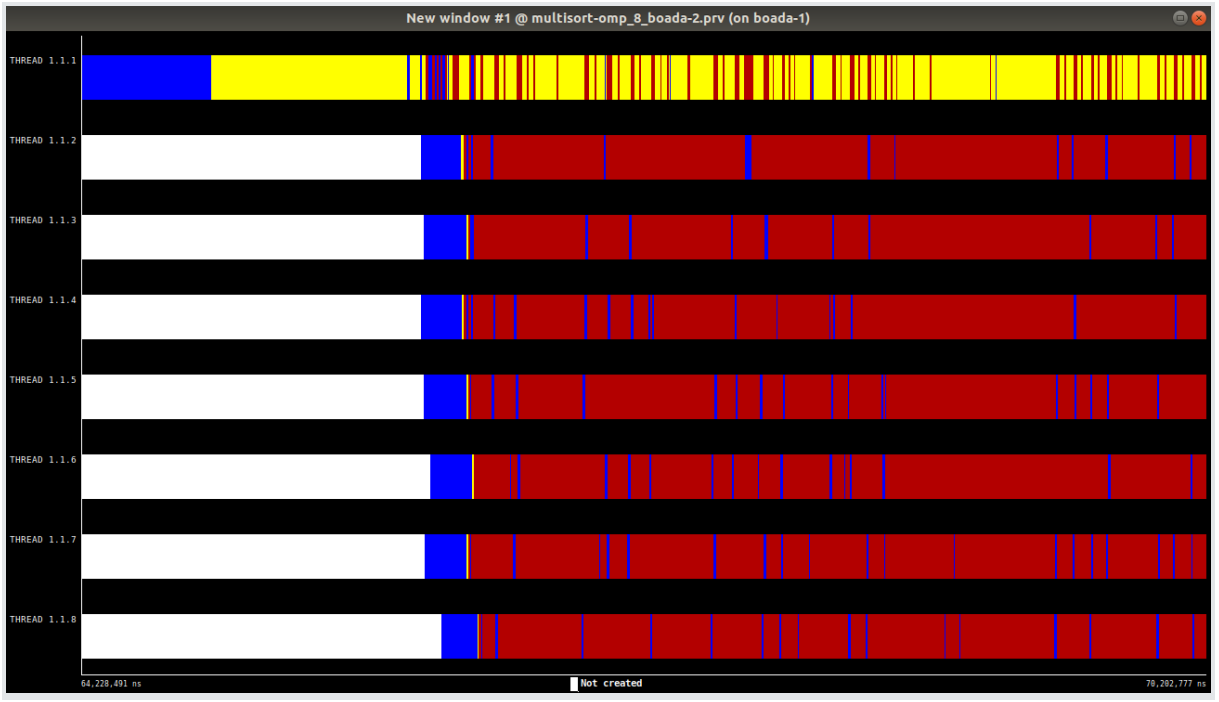


fig. 6: trace plots from the leaf version

As we can see in figure 6, in this version, everything indicates that thread 1 is mainly dedicated to instantiating the tasks that the rest of the threads execute. From this graph, we are also struck by the fact that much of the time is spent synchronizing the threads.

	35 (multisort.c, multisort-omp)	62 (multisort.c, multisort-omp)
THREAD 1.1.1	5	6
THREAD 1.1.2	1,330	150
THREAD 1.1.3	1,528	137
THREAD 1.1.4	1,408	135
THREAD 1.1.5	1,593	157
THREAD 1.1.6	1,402	146
THREAD 1.1.7	1,539	155
THREAD 1.1.8	1,435	138
Total	10,240	1,024
Average	1,280	128
Maximum	1,593	157
Minimum	5	6
StDev	488.62	46.76
Avg/Max	0.80	0.82

fig. 7: histogram from the tree version

If we look at the histogram we can verify that thread 1 is the one who instantiates the tasks, because it has executed less tasks than the rest. We can also see how the workload is distributed equally among the different threads except for thread 1, which, as we have already mentioned, is dedicated to generating the tasks.

To have a clearer vision of the behavior of this strategy, we analyze the table of execution times per thread. Given that the execution times are quite similar between the threads and that the number of tasks executed by each thread are similar too, we deduce that both the number of tasks and the workload per task are evenly distributed among the threads (except thread 1) and tasks.

Tree strategy in OpenMP

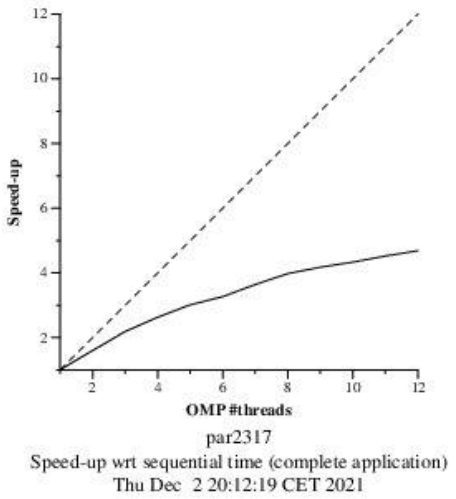
➢ *Tree:* In order to correctly implement the *tree strategy*, we create tasks in the calls to the *merge* and *multisort* functions and using the *OpenMP* directive *#pragma omp taskgroup*, we manage to group the tasks and synchronize the different threads.

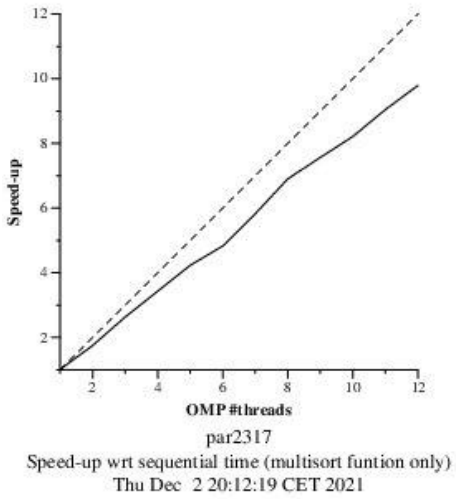
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long
length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
              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

Once the *tree strategy* is implemented, using the *sbatch* command, we execute it several times to check that there are no errors in the execution and we obtain the following output:

```
************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024

Cut-off level:                        CUTOFF=16

Number of threads in OpenMP:          OMP_NUM_THREADS=2

************************************************************************************
Initialization time in seconds: 0.854817

Multisort execution time: 3.660293

Check sorted data execution time: 0.017639

Multisort program finished

************************************************************************************
```

Although we see an improvement in the output, it is not as significant as when comparing the *leaf* strategy with the *sequential* version. With this new implementation, the execution time has been reduced from `3.747875s` to `3.660293`.





Regarding the scalability of this version, we see, in the plots, how the strong scalability of the whole program does not scale especially well, since as the number of threads increases it moves away from the ideal.
On the other hand, the strong scalability, in just the case of the multisort function, behaves in a quite desirable way, avoiding deviating too much from the path of the ideal line. This is because instead of the responsibility of creating tasks in a single thread, it is distributed among all those available.

To get a more complete view of how the script works, we now analyze the execution with *Paraver* and study the histograms to better understand the operation of the tasks in this version.

fig. 8: scalability plot tree version

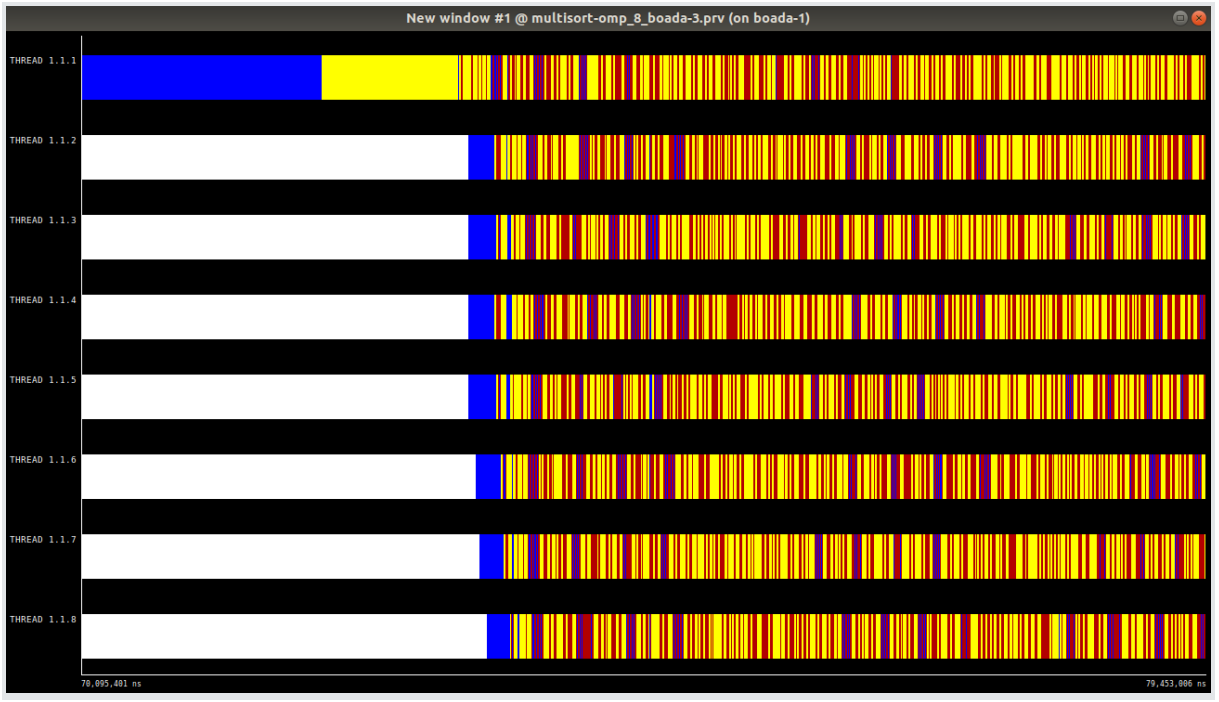


fig. 9: trace *tree* strategy

Thanks to this graph (see fig. 9) we can see that the tasks are distributed evenly between the threads and that the biggest problem with this version is the complexity of distributing the work between the threads, and as a consequence much of the time is invested in scheduling.

	38 (multisort.c, multisort-omp)	40 (multisort.c, multisort-omp)	50 (multisort.c, multisort-omp)	52 (multisort.c, multisort-omp)
THREAD 1.1.1	1,270	1,281	10	10
THREAD 1.1.2	1,144	1,149	43	42
THREAD 1.1.3	1,123	1,134	51	52
THREAD 1.1.4	1,156	1,146	45	44
THREAD 1.1.5	1,174	1,170	42	42
THREAD 1.1.6	1,078	1,085	54	54
THREAD 1.1.7	1,154	1,132	52	49
THREAD 1.1.8	1,118	1,120	44	48
Total	9,217	9,217	341	341
Average	1,152.12	1,152.12	42.62	42.62
Maximum	1,270	1,281	54	54
Minimum	1,078	1,085	10	10
StDev	52.40	53.91	13.04	13.01
Avg/Max	0.91	0.90	0.79	0.79

54 (multisort.c, multisort-omp)	56 (multisort.c, multisort-omp)	62 (multisort.c, multisort-omp)	64 (multisort.c, multisort-omp)
11	13	17	12
43	43	43	44
49	50	51	50
44	43	43	43
44	43	42	43
52	52	52	52
51	50	48	50
47	47	45	47
341	341	341	341
42.62	42.62	42.62	42.62
52	52	52	52
11	13	17	12
12.36	11.69	10.31	12.02
0.82	0.82	0.82	0.82

67 (multisort.c, multisort-omp)
13
44
50
43
43
51
49
48
341
42.62
51
13
11.59
0.84

fig. 10: histograms *tree* version

If we interpret the histograms, we come to the conclusion that the threads execute approximately the same number of tasks and therefore, the workload is distributed equally between the different threads. In addition, we realize that, unlike in the leaf strategy, in this one there's not any specific thread that is exclusively dedicated to generating tasks, but all the threads generate them.

Version comparison

While the *leaf* version is implemented by creating tasks in the base cases, the *tree* does so by creating them before the calls to the *multisort* and *merge* functions.
Regarding the execution time, there's not a big difference, but the tree strategy is more efficient.

Finally, we observe that although both versions have strong scalability problems, the *tree* version manages to considerably improve the speedup of the multisort function, and is able to follow closely the strong ideal scalability line. And as we have seen in both cases, the threads distribute the tasks evenly, except thread 1, in the leaf strategy.

Task granularity control: the cut–off mechanism

```
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);
        }
}
```

In order to increase the granularity and since we don't have any method to know the number of leaves, we will implement a cut-off mechanism.
To do this, we add the parameter *'d'* (as in depth) to the *multisort* and *merge* functions, and we increment it by one for each recursive call. In this way, we know what level we are at and we can stop generating tasks when -c parameter specifies.

When the program receives the -c parameter, it collects its value in the CUTOFF variable, that is why using the OpenMp directive *#pragma omp task final (d <= CUTOFF)* we can create a final type task, in the case the condition is met, and we can stop generating tasks as specified by the user with the new parameter.

We execute the code with the command *sbatch -p execution submit-extrae.sh multisort-omp 8 0* to obtain the graphs of the execution that allow us to analyze it and we obtain the following:

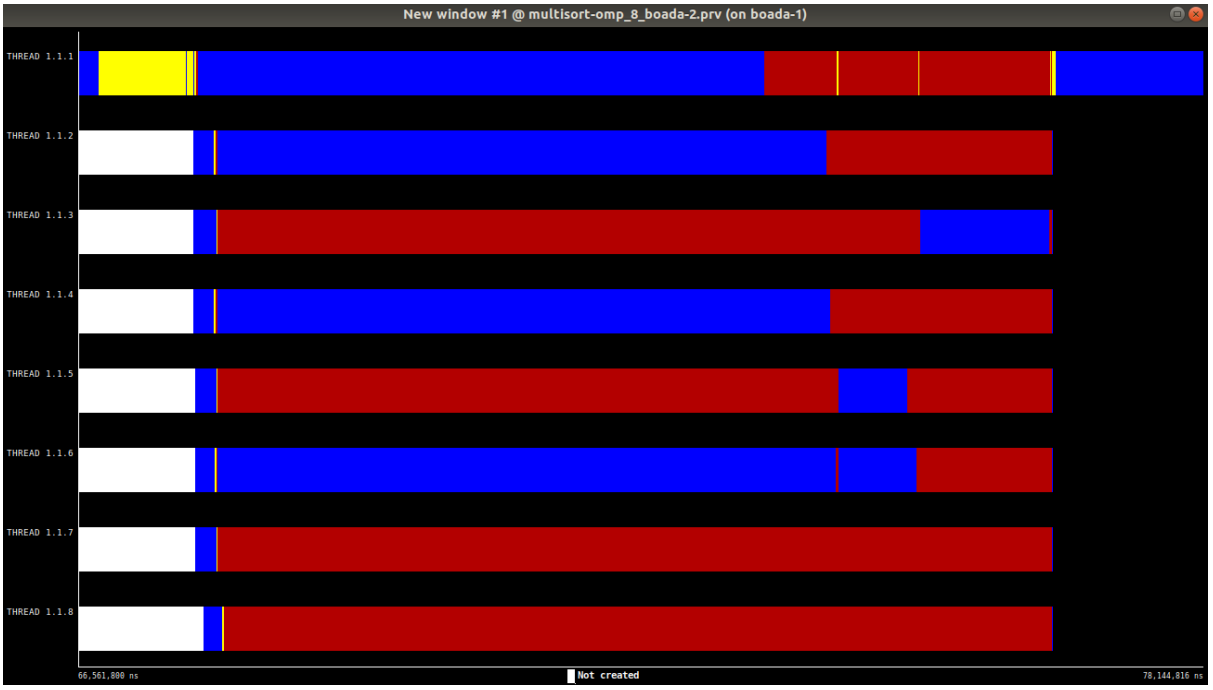


fig. 11: trace cutoff version 1

We execute the code with the command sbatch -p execution submit-extrae.sh multisort-omp 8 1 and we obtain the following:

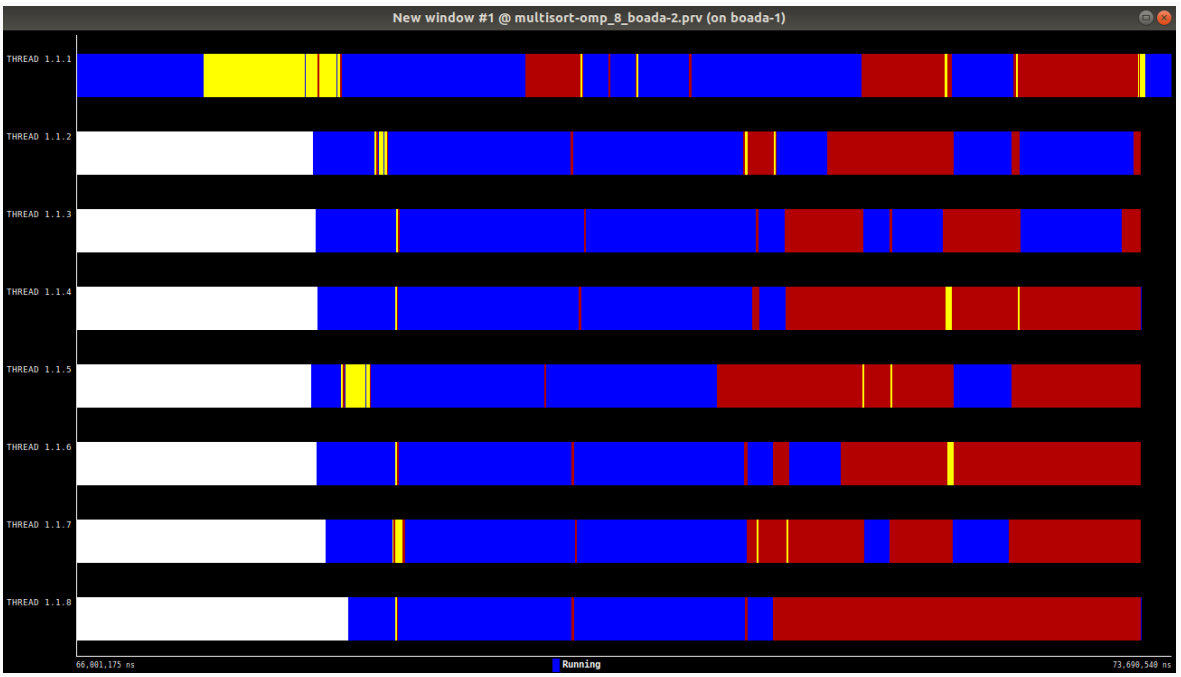


fig. 12: trace cutoff version 2

Compared to the initial version, we see a big change in terms of granularity. Whereas in the original, the tasks were very short and this caused overheads due to scheduling. In the case of the cut-off versions we find bigger tasks which cause synchronization overheads but reduce the time invested in scheduling.

Regarding the differences that we find between the cut-off versions, if we compare the traces of the two cut-off versions, we see that the figures 11 and 12 are more or less similar, the difference is that in the first one it spends less time scheduling and running and more time in synchronization.

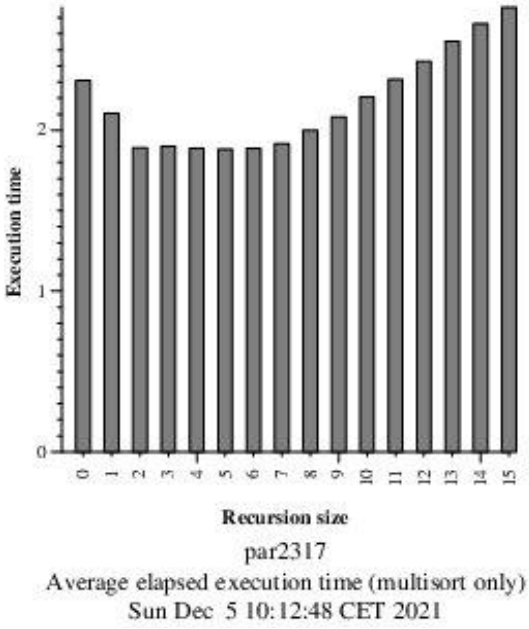


fig. 13: elapsed time executing the mutlisort function with 8,4 and 2 threads respectively

After having tested the execution with different numbers of threads, we have come to the conclusion that the best option is to use 8 threads, since with other values such as 4 or 2 the average execution time is longer. We can see in the graph (see fig. 13) how the best results are obtained with the cut-off equal to 5 or 6, although we also obtain similar results with 1,2,3,4 and 7.

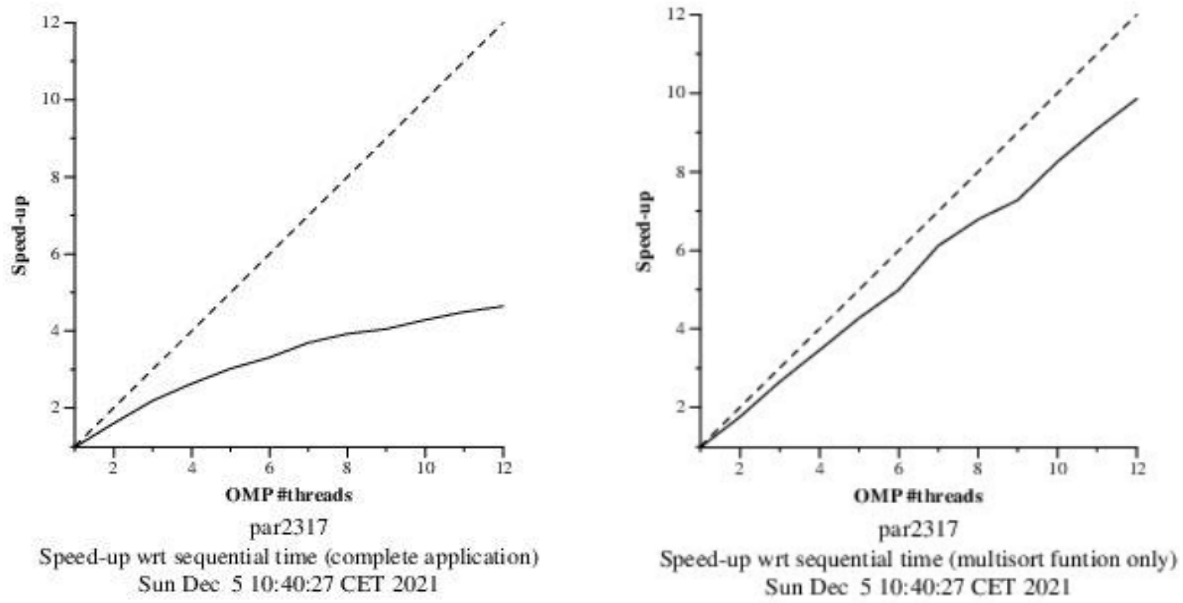


fig. 14: scalability plot tree strategy using *cut-off*

Finally, we analyze the scalability of this version by executing the binary along with the *submit-strong-omp.sh* script. This version using the maximum value of *cut-off* (see fig. 14), as expected, behaves very similarly to the tree strategy in terms of scalability. The same pattern can be found, while the whole program loses speed very quickly, the multisort function manages to closely follow the optimal scalability line. Then, if we use our own values, in this case we chose that *cut-off* is 6 (which is one of the optimum values that we found in the previous analysis), we see that the scalability is even better. In other words, the scalability line is even closer to the optimal than using the maximum value of *cut-off*.

Optional 1

In this optional section, we set the maximum number of cores to 24. In order to do that, we change the variable np_NMAX inside the *submit-strong-omp.sh* script.

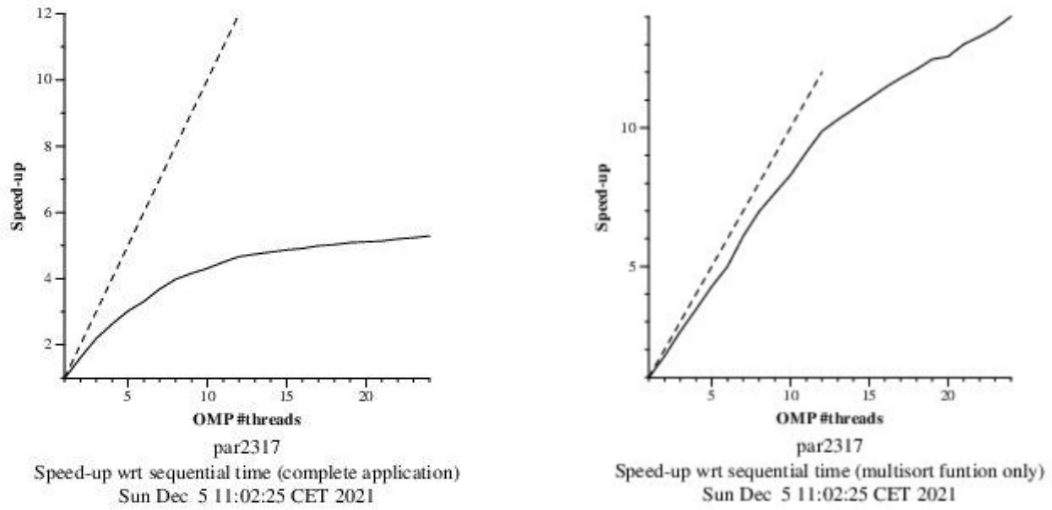


fig. 15: scalability plots *tree* strategy until 24 cores

In this case, the scalability graphs continue to improve efficiency by increasing the number of threads because boada has 12 cores and, since each core has 2 threads, it can execute at the same time 24 tasks, giving room for even more improvement than before.

Using OpenMP task dependencies

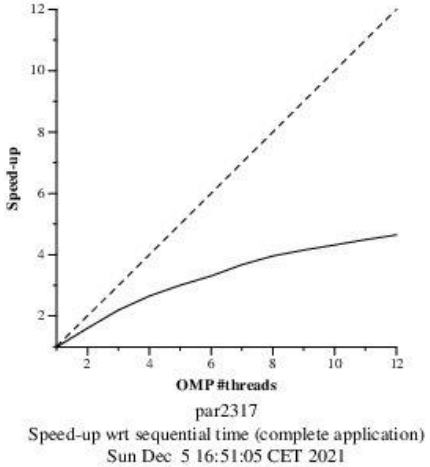
In this final session, we modify the previous *tree* strategy code that uses the *cut-off* mechanism in order to enforce task dependencies. We avoid the *taskwait* and *taskgroup* clauses that we applied before so we can express dependencies among tasks. Instead, in this case we use the clauses `#pragma omp task depend(in: ...)` and `#pragma omp task depend(out: ...)` in order to set the values for the next or the other dependent operations respectively.
In the following code we see the modification that we did replacing some of the *taskwait* and *taskgroup* clauses that are redundant with the *task depend* clauses, so we can specify the dependencies among tasks (see `multisort` function specifically):

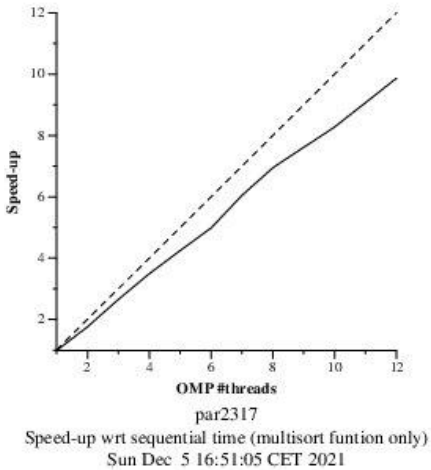
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long
length) {
        if (length < MIN_MERGE_SIZE*2L) {
            // Base case
            basicmerge(n, left, right, result, start, length);
        } else {
            // Recursive decomposition
            #pragma omp task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

void multisort(long n, T data[n], T tmp[n], int depth) {

  if (n >= MIN_SORT_SIZE*4L) {
      // Recursive decomposition
    #pragma omp task depend(out: data[0])
    multisort(n/4L, &data[0], &tmp[0], depth +1);
    #pragma omp task depend(out: data[n/4L])
    multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
    #pragma omp task depend(out: data[n/2L])
    multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
    #pragma omp task depend(out: data[3L*n/4L])
    multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+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);
    #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);
    //#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);
    #pragma omp taskwait

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

Next, we compile it and execute it using 8 threads and we make sure that the output does not return errors. The following output is an example of one of our executions:

```
************************************************************************

Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024

Cut-off level:                        CUTOFF=16

Number of threads in OpenMP:          OMP_NUM_THREADS=8

************************************************************************

Initialization time in seconds: 0.855207

Multisort execution time: 0.898116

Check sorted data execution time: 0.015405

Multisort program finished

************************************************************************
```

If we compare it with the previous versions that we did in the other sessions, this one, if we look at the execution time, seems to be the best one and most efficient out of all of them. We see this better graphically with the scalability plots:





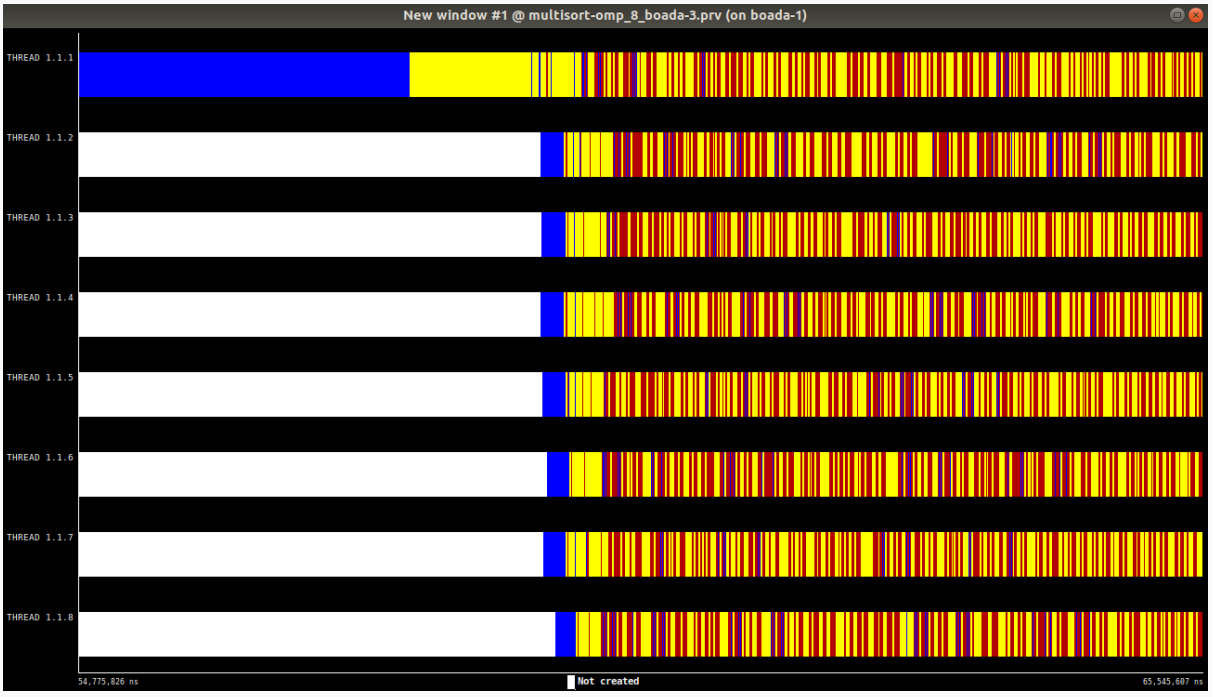
As we can see the plots (see fig. 16) are very similar to the previous *tree* versions. Neither the *cut-off* nor the *tree* strategy with *OpenMP* are very distinct to the performance of this version. This is surprising because we thought that when applying the *task depend* clauses the execution only has to wait for the specific values that are indicated in the code, instead of waiting for all the tasks to be finished and suspending the tasks that wait.

Regarding the programmability, this version seems more logical for the programmer to code because he/she knows at all times what tasks depend on others and can clearly imagine the task dependence graph. Though, this version is more tedious and difficult to code than the previous ones that use *taskgroup* and *taskwait* clauses since it is longer and more detailed.

If we trace the parallel executions we get the following trace plot and histograms to understand how the

fig. 16: Scalability plots of the version with task dependencies

performance of this version is achieved:



As we can see in fig. 17, the traces are similar to the first *tree* strategy (see fig. 9) with running time and synchronization, but very different from the *cut-off* traces that have a lot more running time.

Finally, we generate the table that gives us a profile of the task creation and execution of the program (see fig. 18). We can see that the execution of the code is correctly parallelized because every thread does almost the same amount of tasks for each one of the columns:

	38 (multisort.c, multisort-omp)	40 (multisort.c, multisort-omp)	50 (multisort.c, multisort-omp)	52 (multisort.c, multisort-omp)
THREAD 1.1.1	1,195	1,214	38	37
THREAD 1.1.2	1,141	1,139	40	40
THREAD 1.1.3	1,192	1,174	45	42
THREAD 1.1.4	1,157	1,157	34	35
THREAD 1.1.5	1,179	1,182	46	46
THREAD 1.1.6	1,069	1,068	51	49
THREAD 1.1.7	1,199	1,195	42	42
THREAD 1.1.8	1,085	1,088	45	50
Total	9,217	9,217	341	341
Average	1,152.12	1,152.12	42.62	42.62
Maximum	1,199	1,214	51	50
Minimum	1,069	1,068	34	35
StDev	47.32	47.99	4.95	5.05
Avg/Max	0.96	0.95	0.84	0.85

54 (multisort.c, multisort-omp)	56 (multisort.c, multisort-omp)	60 (multisort.c, multisort-omp)	62 (multisort.c, multisort-omp)	66 (multisort.c, multisort-omp)
37	38	38	37	38
40	40	40	40	40
42	42	46	43	42
35	35	33	35	35
48	47	46	47	48
50	49	50	50	48
43	44	42	43	44
46	46	46	46	46
341	341	341	341	341
42.62	42.62	42.62	42.62	42.62
50	49	50	50	48
35	35	33	35	35
4.90	4.47	5.12	4.77	4.44
0.85	0.87	0.85	0.85	0.89

fig. 18: Profile of explicit tasks creation and execution (version using task dependencies)

Optional 2

In this final optional section we will take the previous code using task dependencies and we will parallelize the 2 functions that initialize the vectors `data` and `tmp`. If we take a look at the code we see that this is done in the `initialize` and `clear` functions. Since the vectors are initialized using a *for* loop, we parallelize the loop using the *#pragma omp parallel for* clause as the following example:

```
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;
    }
}
```

After doing these changes we compile and execute code using the *submit-strong-omp.sh* script so we can analyze the scalability of the parallelized code. This execution generates the following speed-up plots:

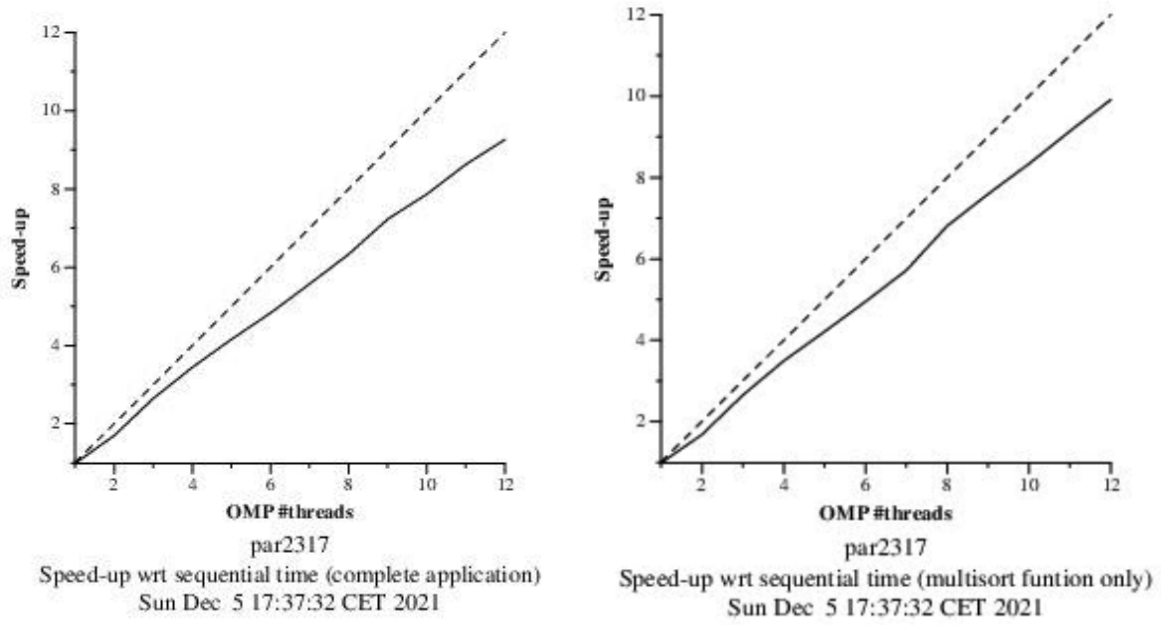


fig. 19: Scalability plots (Optional 2)

If we look at the plots (see fig. 19) and compare it with the previous *tree* versions, we see that the performance is better. That is because it is parallelized even more than before as we parallelize the functions that are first called in the main function.

If we generate the *Paraver* traces we see that threads other than the first one start running before the other versions. That is, they have less *not created* regions (see white part in fig. 20) and more *running* regions (see blue part) at the beginning of the execution. So now the execution is even more parallelized than before. Here's the traces that we generated (we emphasize the beginning of the execution so we can see better the difference with previous traces):

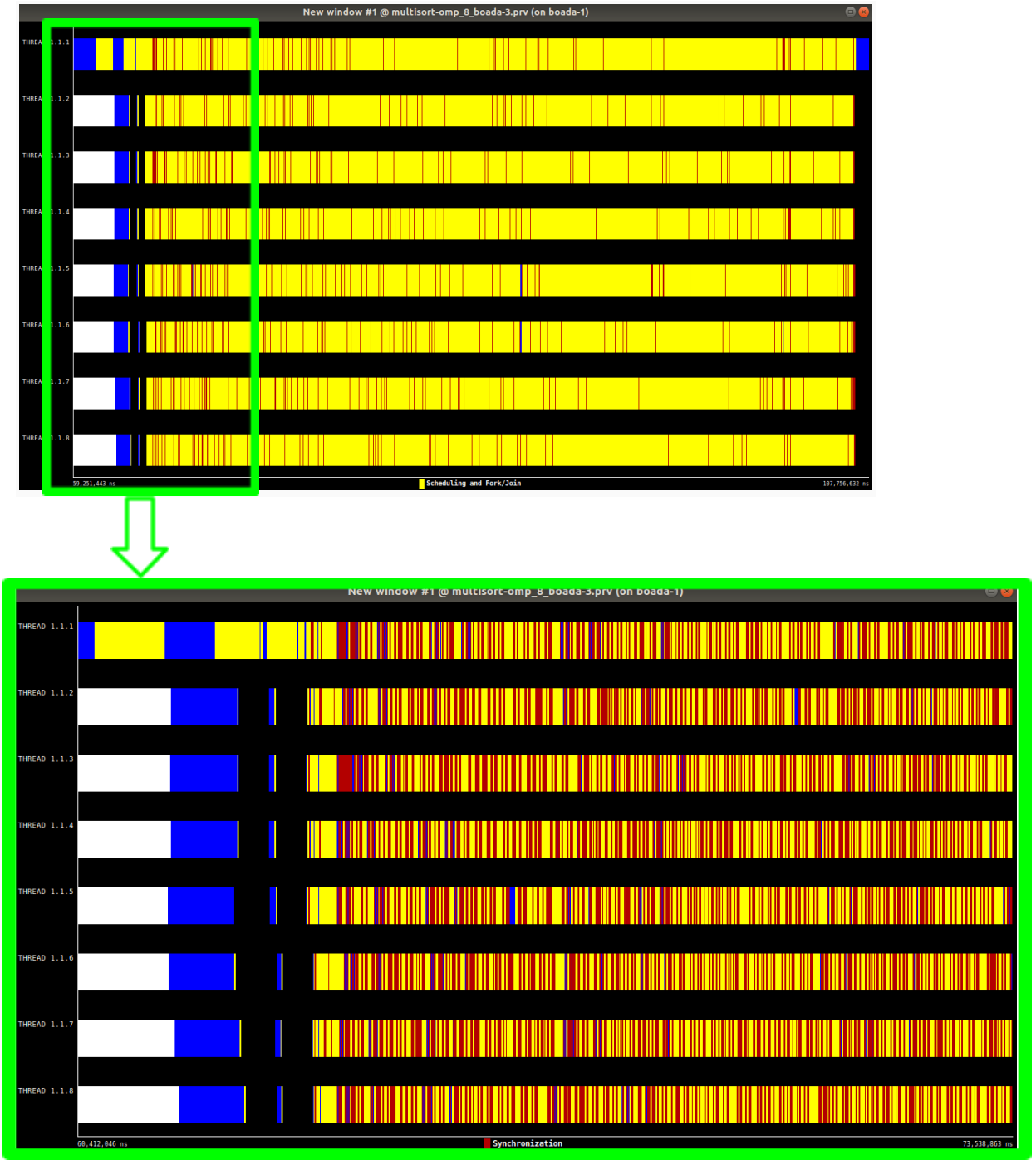


fig. 20 : *Paraver* traces (Optional 2). The image at the bottom is the zoomed part of the beginning of the execution

Conclusion

In the first part of this practical assignment, apart from learning how to deal with recursive functions we saw two different strategies to parallelize the code we were given: the *leaf* strategy and the *tree* strategy.
Once we studied these possibilities and we knew it worked, we proceeded to implement them in the code. Then, we compiled and executed it using the corresponding script so we could visualize it with *Paraver*.

In the second part, we continued working on the study of strategies, but this time focusing on parallelizing the script. In this section we tested the different strategies to parallelize the code.

In conclusion, during these lab sessions we have learned how to parallelize codes that include recursive functions and we have been able to check the efficiency of the different implementations depending on the case.