PARALLELISM

Laboratory 4: Divide and Conquer parallelism with OpenMP: Sorting

PAR1105

Albert Borrellas Maldonado

Víctor Martínez Murillo

04-12-2019

Fall 2019-2020

Introduction

In this laboratory assignment we will use Mergesort, a sort algorithm that combines "divide and conquer" strategy dividing the initial list recursively, ordering sublists with quicksort and then merging them back into a sorted list. It will be implemented combining the call of functions *multisort* and *merge*, provided in multisort.c.

Session 1: Task decomposition analysis for Mergesort

In this first session, we will work with the code multisort-tareador.c which is a version modified by including the *Tareador* calls to analyze the task decomposition. Basically, the calls are introduced in every recursive call in functions *merge* and *multisort* that are the ones we are going to parallelise.

```
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("merge1");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("merge1");
                tareador_start_task("merge2");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("merge2");
        }
}

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

                tareador_start_task("multisort3");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("multisort3");

                tareador_start_task("merge1m");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("merge1m");
                tareador_start_task("merge2m");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("merge2m");
                tareador_start_task("merge3m");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("merge3m");
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Image 1: Code used in the *Tareador* version.

Image 2 is the task dependency graph generated by *Paraver*. *Multisort* does not create dependencies as it does not depend on other subdivisions of the original vector. However, *merge* does it, because it reunites two subdivisions of the vector and they must have finished the *multisort* stage.

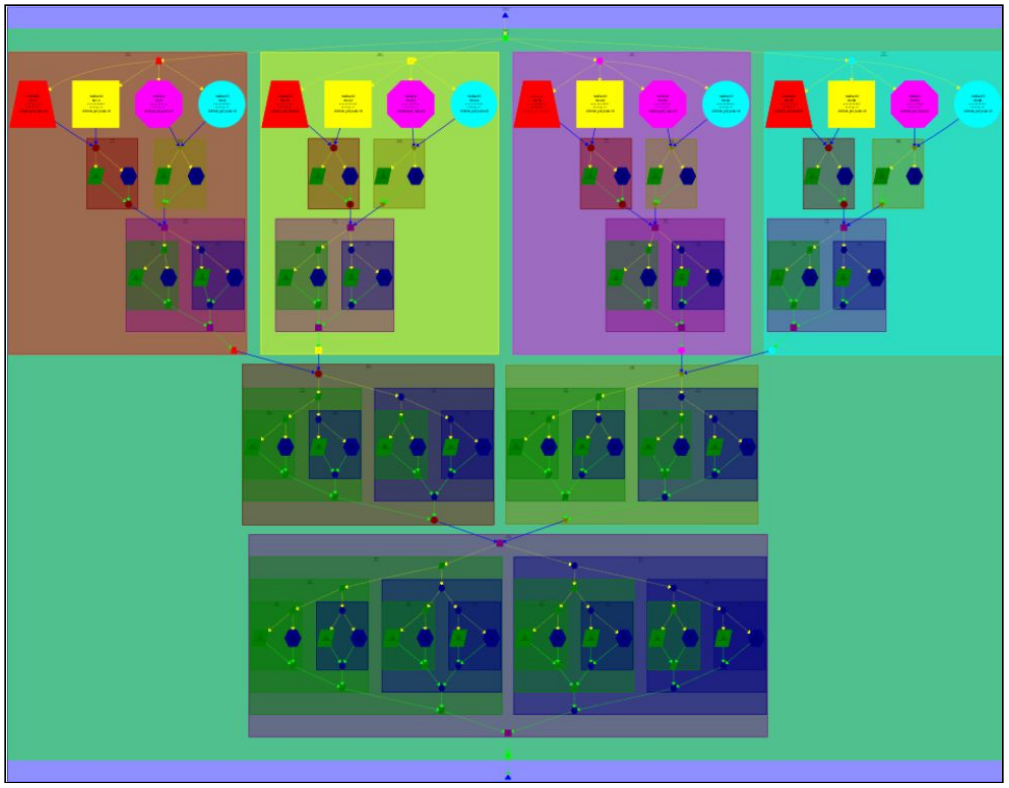

Image 2: Tasks dependency graph on the tareador version.

In *Paraver* timelines, we can see that *Tareado* spends most of its time waiting in the critical zone (showed in red) than running the code (showed in blue), the yellow segments show us that the first thread is the only one creating descendants.

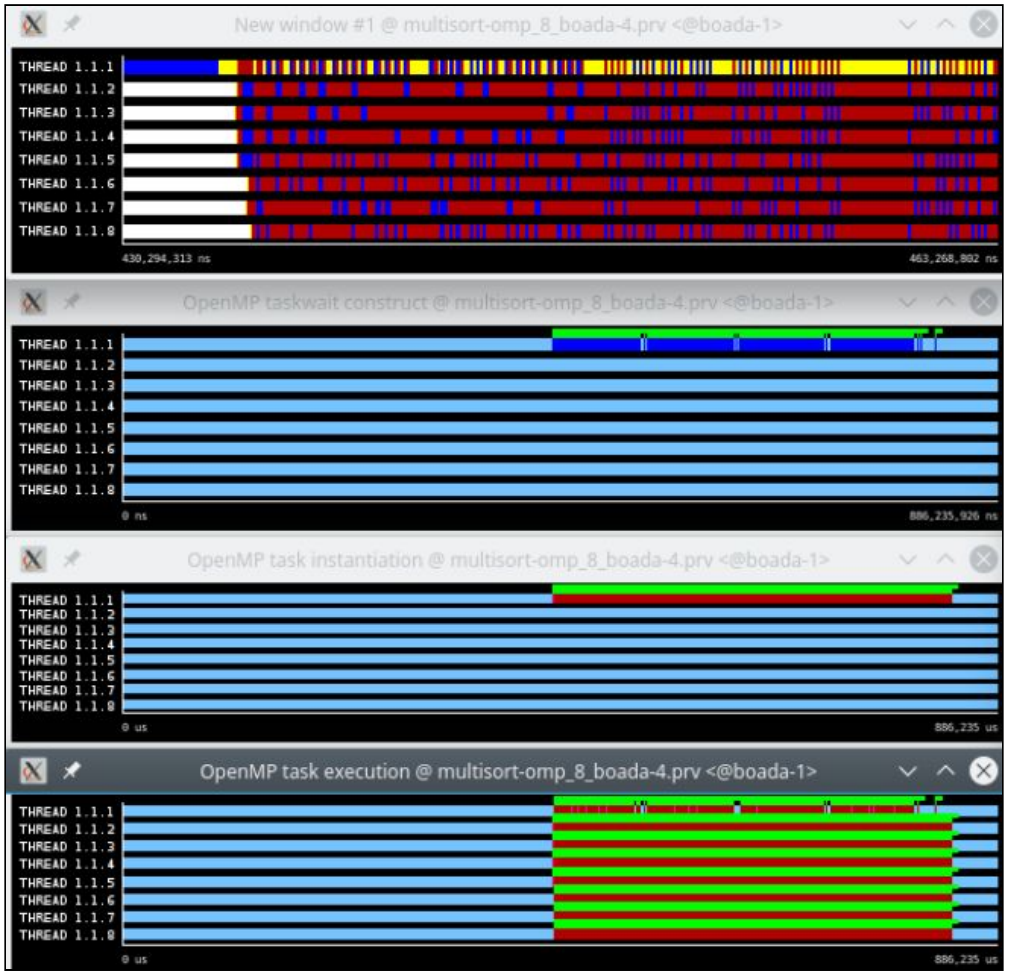

Image 3: *Paraver* tasks configurations from *Tareador* version.

We obtained the scalability plot from two different sources: image 4, the script *submit-strong-omp.sh*, used in previous lab sessions, and, image 5 is based on predictions made by *Tareador*. Although, the first one do not represent as many processors as the second one, there are two similarities: the speed-up is far from being the ideal case, and there is somewhere between 12 and 16 processors where the speed-up gets constant, as seen at Image 2, it is only possible to parallelize 16 tasks.

Disclaimer: all scalability plots in this assignment show two plots, the upper one from the complete application (initialization + multisort execution) and the lower one, which only contemplate the multisort function. This is why the upper one is always similarly bad as the initialization is not parallelized **until optional 2**.

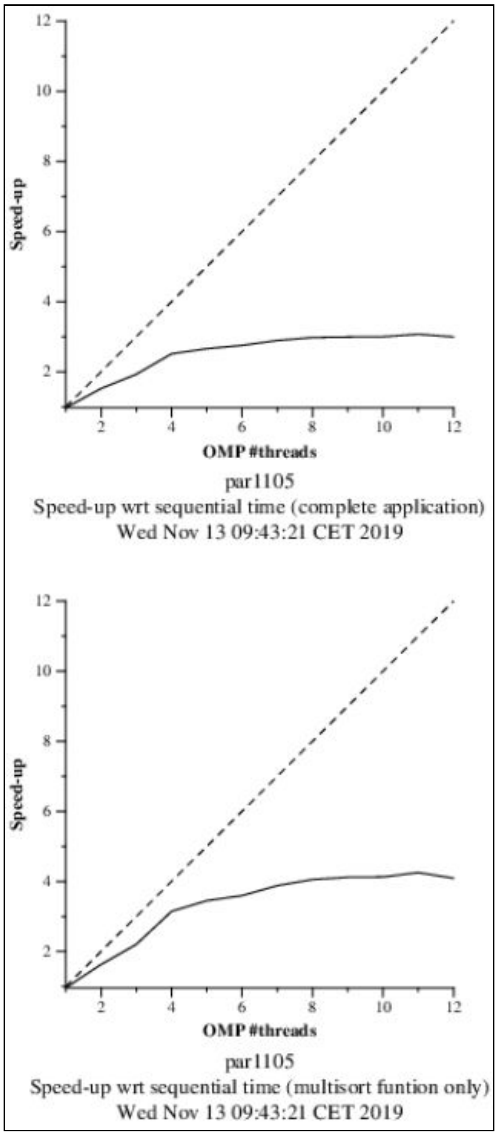

Image 4: Scalability plots from *Tareador* version.

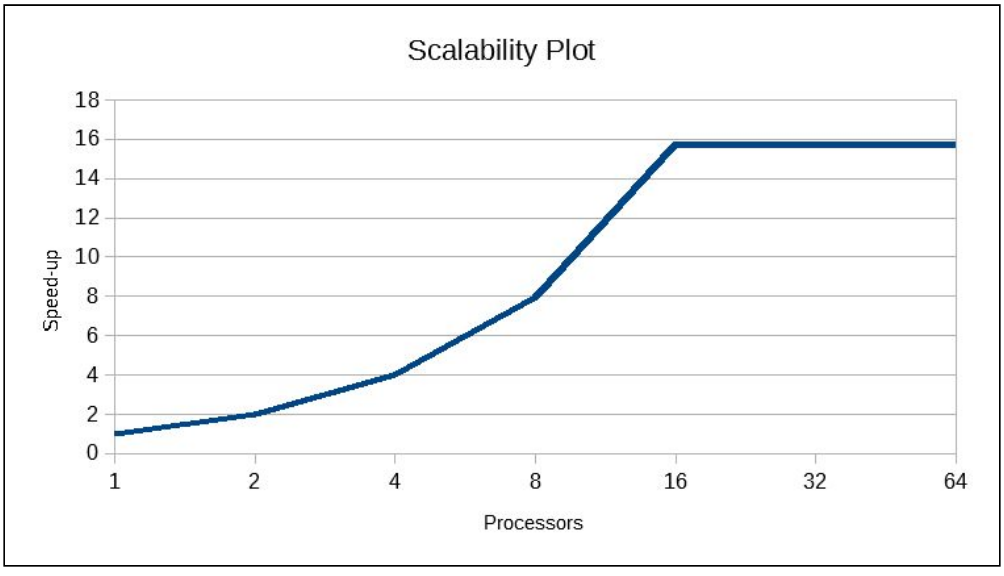

Image 5: Scalability plot based on the predictions from *Tareador*.

In addition, we made a table that shows the time execution (showed in μs) with different number of processors, and the speedup. We can see that if we duplicate the processors we reduce the time in a half, until the number reaches 16 processors.

#Processors	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Time Execution (μs)	20.334.411	10.173.716	5.086.725	2.550.595	1.289.922	1.289.909	1.289.909
Speed-up	1	1,9987	3,9975	7,9724	15,7640	15,7642	15,7642

Table 1: Execution time and speed-up predicted by *Tareador* using different number of processors.

Session 2: Shared-memory parallelization with OpenMP tasks

In this lab session we will use the leaf and tree strategies to parallelize the original sequential code (*multisort.c*).

The leaf version creates the tasks once the recursion is finished and the program's flow is at the leaves. Specifically, the task directives from OpenMP will be called at *basicsort* and *basicmerge*.

```
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);
        }
}
```

Image 6 : The leaf version code.

The tree version does the opposite, it creates the tasks during the recursive decomposition. Specifically, the directives will be called at 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
                #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]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #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 taskwait
                #pragma omp task
                #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 taskwait
                #pragma omp task
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                #pragma omp taskwait
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Image 7: The tree version code.

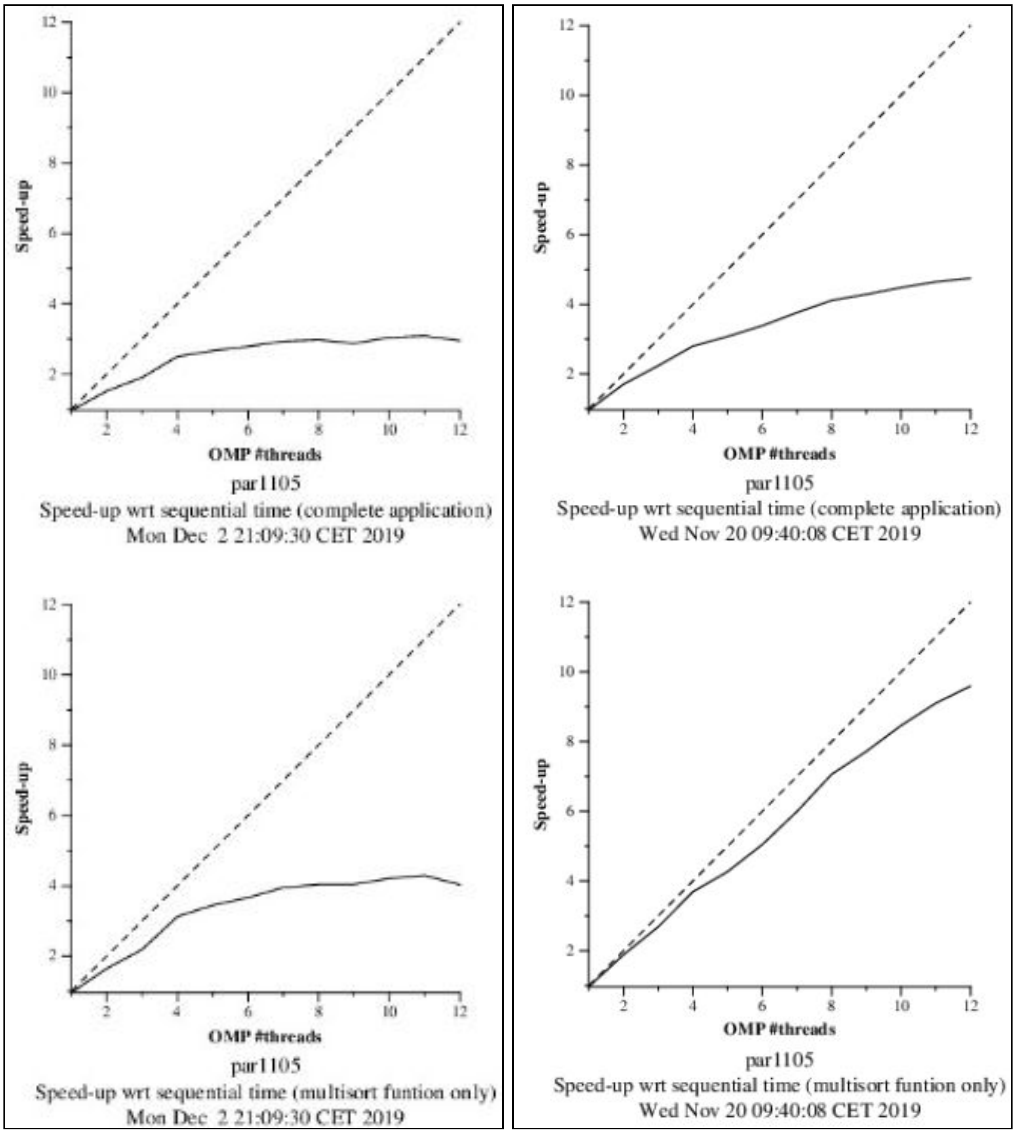

Image 8: Scalability plots from multisort-omp with the leaf version (left) and the tree version (right).

Once scalability plots from both versions are seen, we can clearly determine which version is the best. As imagined, the tree version is better by far compared to the leaf version, this is because of the fact that the leaf version is a strategy in which we only use parallelism at the end of the program flow, otherwise the tree version code does the opposite and takes in advantage the benefits of parallelism and creates tasks in the recursive calls to *multisort* and *merge* functions, so the calls will be distributed more often and it is obvious that with the increment of threads the code has a better execution time.

To finish this lab session we will use the cut-off mechanism in our tree version code to control the maximum recursion level for the task generation.

```
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);
        }
```

Image 9: Cut-off mechanism in the tree version code.

The recursion level in which we set the cut-off is done manually. By default, it is set to 4, but it can be modified as pleased. Such is that, a comparison between the different recursion level cut-off is strongly recommended. As seen in image 10, values near the median, are the ones in which the execution time is lower because the number of tasks created is well controlled.

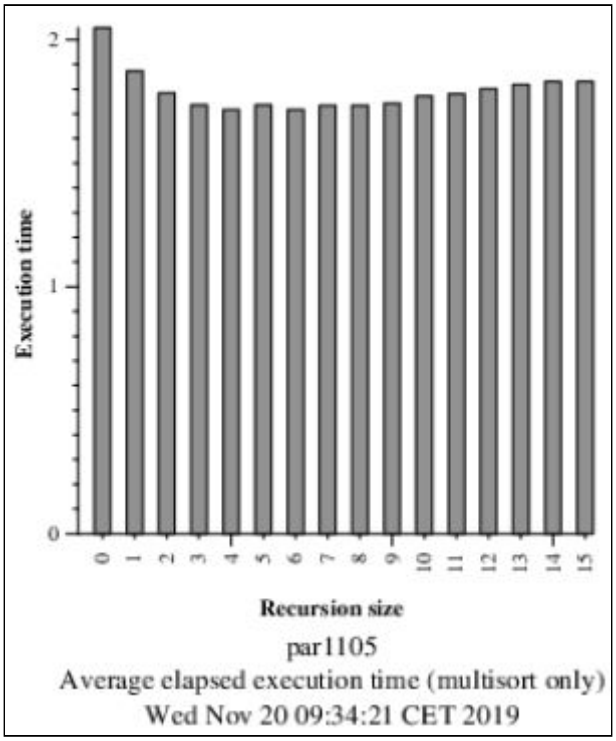

Image 10: Cut-off mechanism level comparison in execution time.

Tasks are created until the cut-off is reached [image 11].

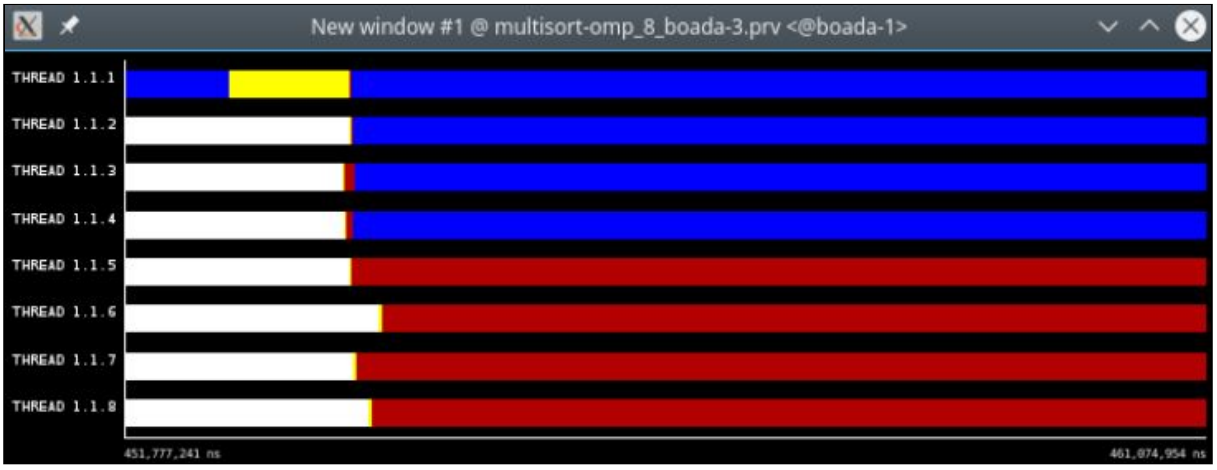

Image 11: *Paraver* timeline.

Optional 1:

In this first optional we want to execute the multisort-omp tree version code in different *Boada*'s. We obtained the plot from the *boada* 1-4 before [image 8, right] so we will show the plots obtained from *boada* 5 and *boada* 6-8. For the execution in *boada* 6-8 we modified the np_NMAX = 16 (compared to the 12 from *boada* 1-4 and 5, the default value) in the submit-omp-strong.sh file as this *boada* architecture dispose of more *cores*.

Both scalability plots show a really good speed-up, pretty close to the ideal case, but as imagined, the one from *boada* 6-8 goes further because of the number of cores.

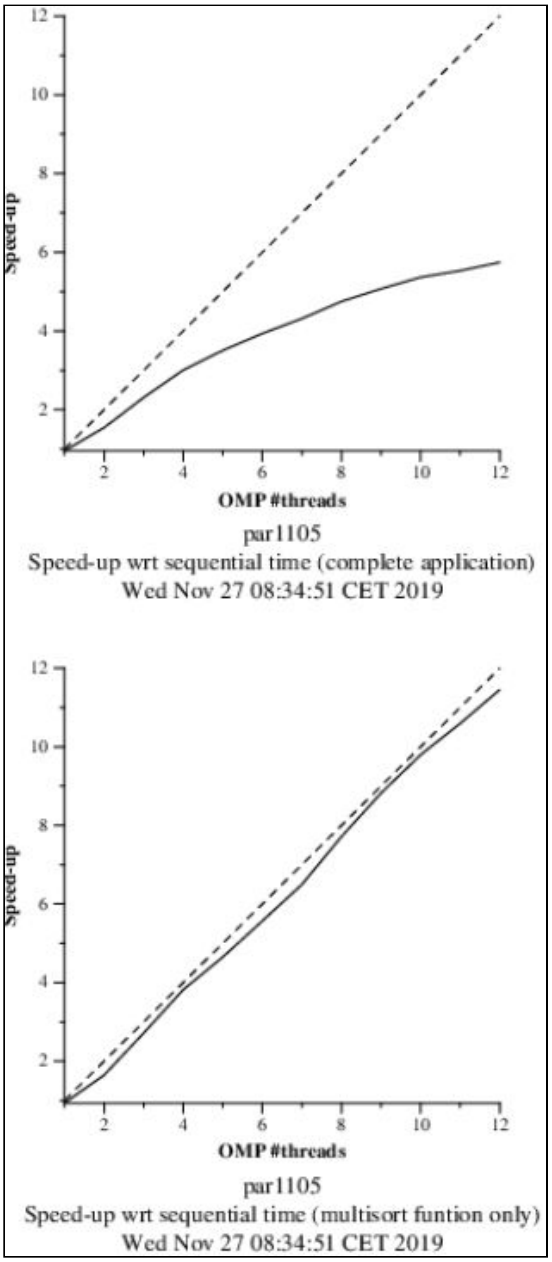

Image 12: Scalability plots from multisort-omp from the tree version in boada-5.

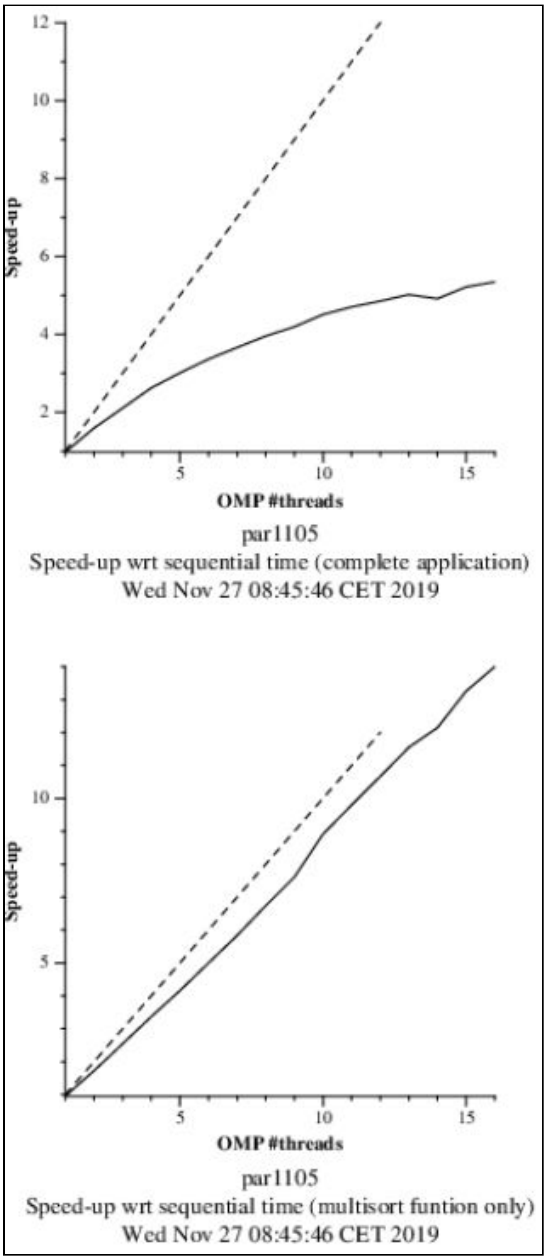

Image 13: Scalability plots from multisort-omp from the tree version in boada-6.

Optional 2:

In this second optional we will modify our code from the tree version completing the parallelization by parallelizing the two functions that initialize the *data* and *tmp* vectors. As commented in the previous disclaimer, now the upper scalability plot is also determined by the application of parallelism to the initialization, such is that the upper plot is close to the ideal case. [code included in multisort-omp-treetwait-op2.c]

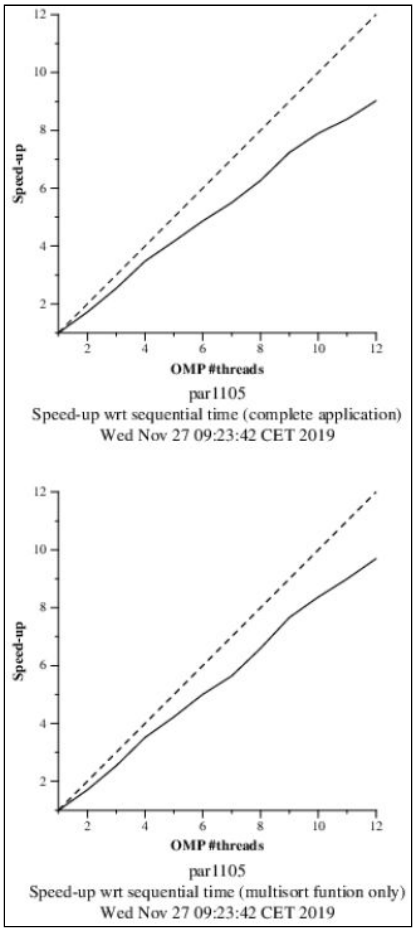

Image 14: Scalability plots from multisort-omp-twait-op2 from boada-4.

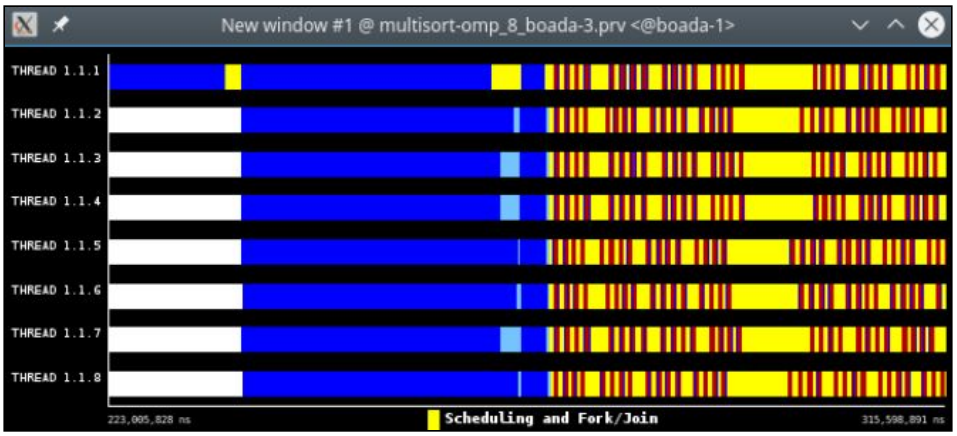

Image 15: Paraver capture from multisort-omp-twait-op2 from boada-4.

Session 3: Using OpenMP tasks dependencies

In this laboratory session we will change the Tree parallelization version code that we made on the previous session and express some dependencies changing the clause *taskwait* to *depend*. Is not always possible for the program to work as desired, that is why some *taskwait*'s are still in this code.

```
void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long length);

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

                #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[n/2L], data[3L*n/4L]) depend(out: tmp[n/2L])
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                #pragma omp task depend(in: tmp[0], tmp[n/2L]) //depend(out: data[0])
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                #pragma omp taskwait
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Image 16: Tree version code with tasks dependencies.

The dependences in the code are the part of the vector that we sent to sort with both multisort and merge functions. In addition, in the merge call functions we need the data that we sent to have been sort before by the multisort function. In the void merge we only call #pragma omp task because the are no dependencies.

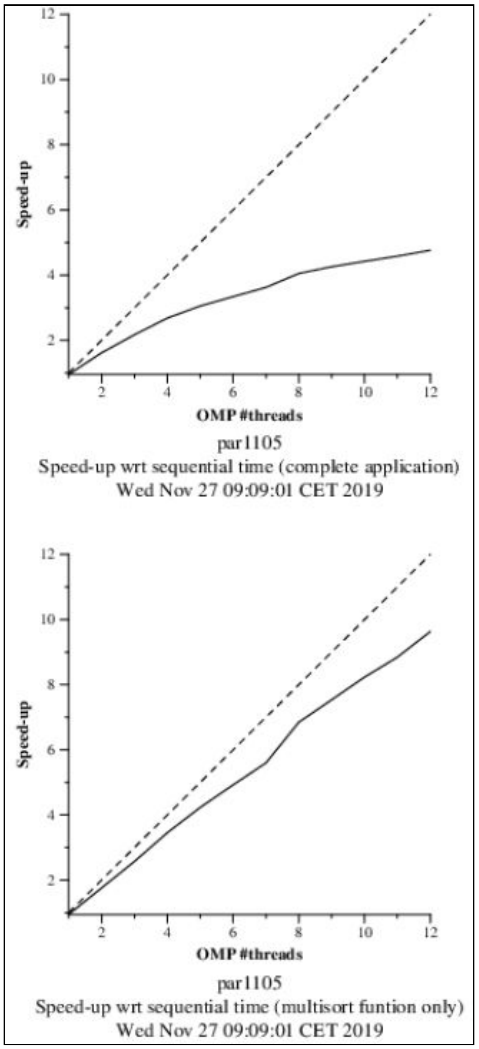

Image 17 :Scalability plot from the tree version code with tasks dependencies.

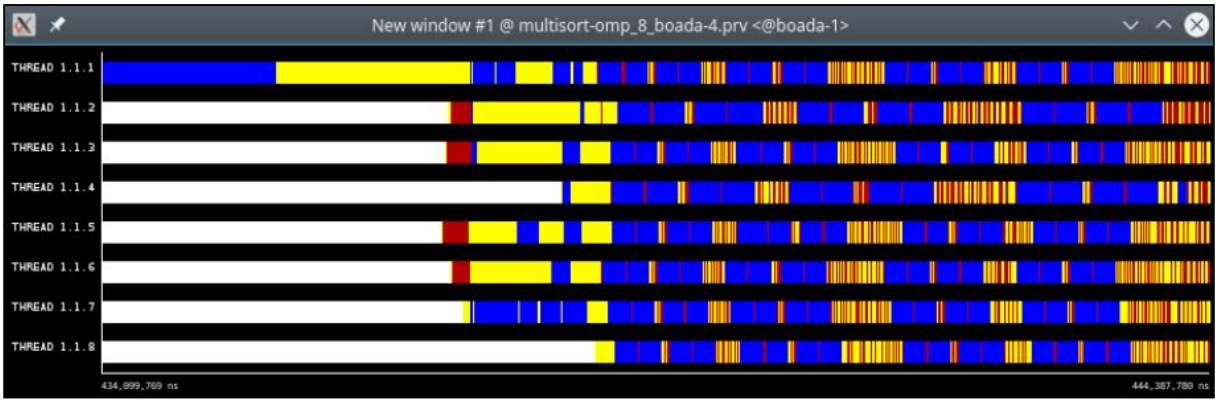

Image 18: Tareador from the tree version code with tasks dependencies.

The scalability plot and the paraver capture show us the code takes more time waiting to the dependences to end that in the original tree version code that spends much more time running (blue in the paraver timeline). But there is not much difference comparing both scalability plots.