PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

PAR2303
Joan Manuel Ramos Refusta
Àlex Aguilera Martínez

1. Analysis with Tareador

In this part of the deliverable, we are going to understand the potential parallelism that the ”divide and conquer” strategy provides when applied to the sort and merge phases.

Firstly, we execute multisort.c in order to see the sequential time execution of our program:

```
par2303@boada-1:~/lab4$ ./multisort -n 32768 -s 32 -m 32
Arguments (Kelements): N=32768, MIN_SORT_SIZE=32, MIN_MERGE_SIZE=32
Initialization time in seconds: 0.847806
Multisort execution time: 6.178766
Check sorted data execution time: 0.015220
Multisort program finished
```

Figure 1, execution of sequential multisort

Now, we have to insert properly the Tareador instrumentation to obtain the task dependency graph. Therefore, we have to modify merge and multisort functions including tareador_start_task () or tareador_end_task() for each task. You can see the result in figure 2 and 3.

```
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_1");
                basicmerge(n, left, right, result, start, length);
                tareador_end_task("base_1");
        } else {
                // Recursive decomposition
                tareador_start_task("merge_3");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("merge_3");
                tareador_start_task("merge_4");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("merge_4");
        }
}
```

Figure 2, function merge of the multisort-tareador.c code

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("multisort_0");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("multisort_0");

                tareador_start_task("multisort_1");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("multisort_1");

                tareador_start_task("multisort_2");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("multisort_2");

                tareador_start_task("multisort_3");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("multisort_3");

                tareador_start_task("merge_0");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("merge_0");

                tareador_start_task("merge_1");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("merge_1");

                tareador_start_task("merge_2");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("merge_2");
    } else {
        // Base case
        tareador_start_task("base_2");
        basicsort(n, data);
        tareador_end_task("base_2");
    }
}
```

Figure 3, function multisort of the multisort-tareador.c code

We haven't followed any strategy in this part, nor tree strategy nor leaf strategy. We wanted to see all the task dependencies in the generated Tareador graph. Therefore, we have inserted tareador_start_task() and tareador_end_task() that way.

Then, we obtain the following task dependency graph:

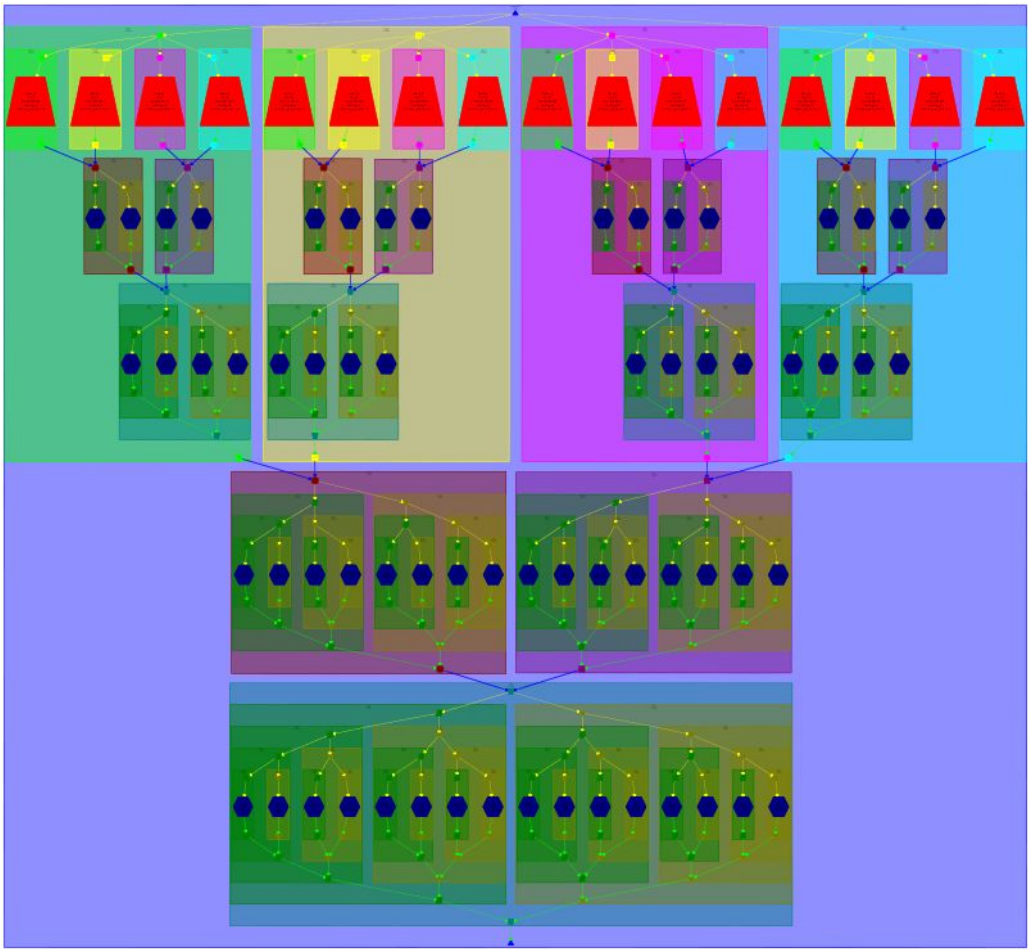

Figure 4, graph dependence of multisort-tareador.c

Finally, we have written a table with execution time and the speedup in order to predict the parallel performance and scalability with different number of processors (for 1, 2, 4, 8, 16, 32 and 64 processors). Additionally, we have included the different generated timelines to see how our program scales with more or less processors.

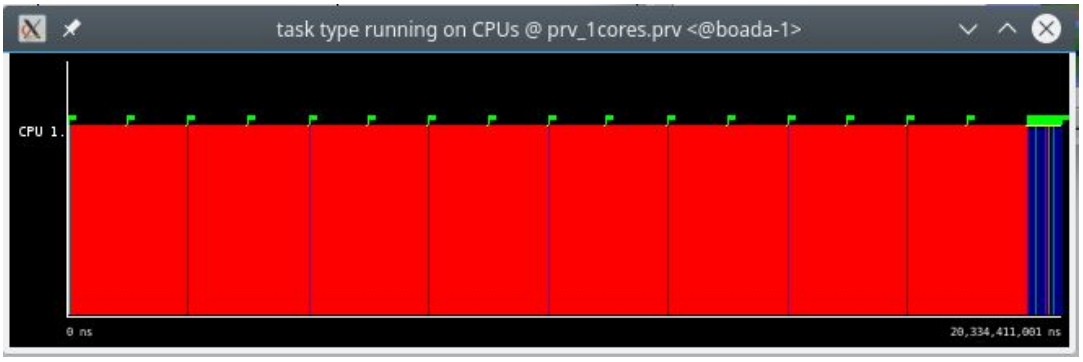

Figure 5, timeline per processor with 1 CPU

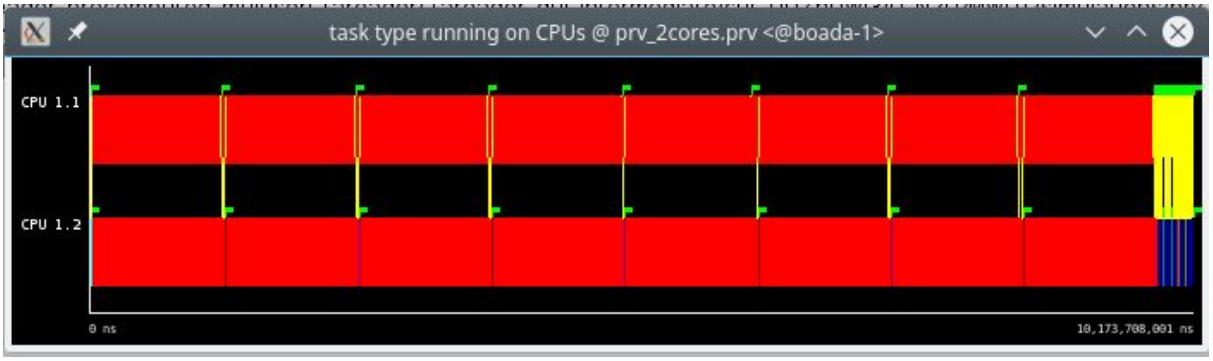

Figure 6, timeline per processor with 2 CPU

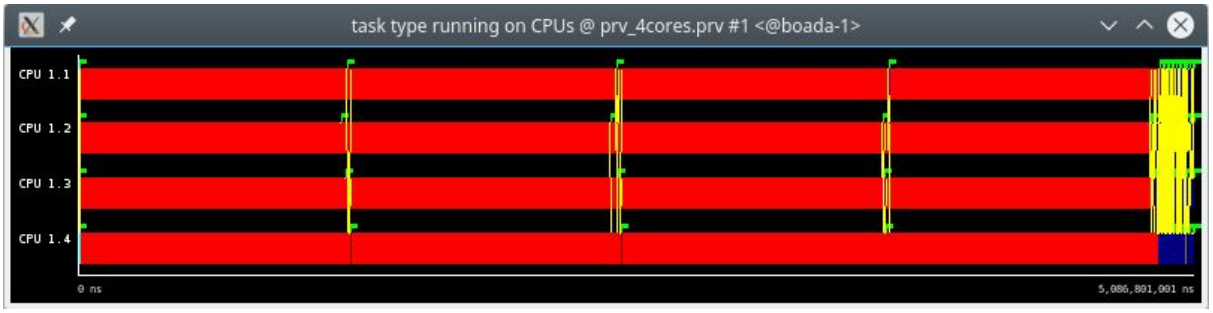

Figure 7, timeline per processor with 4 CPU

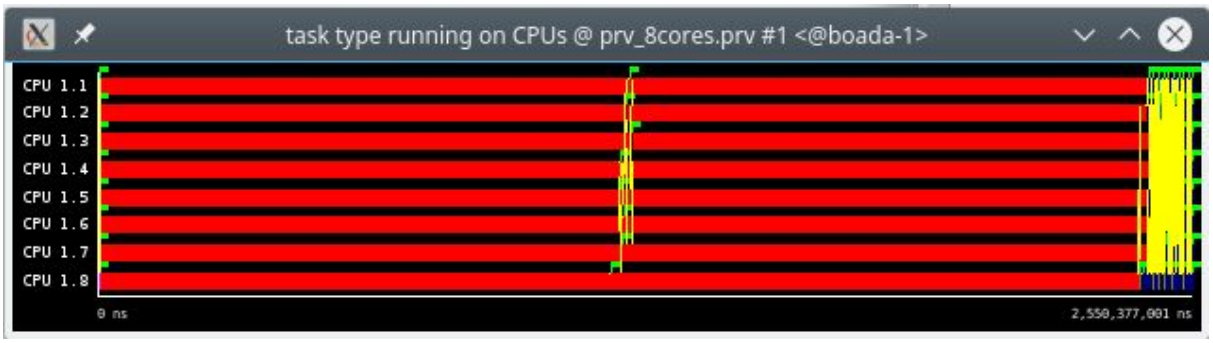

Figure 8, timeline per processor with 8 CPU

Figure 9, timeline per processor with 16 CPU

Figure 10, timeline per processor with 32 CPU

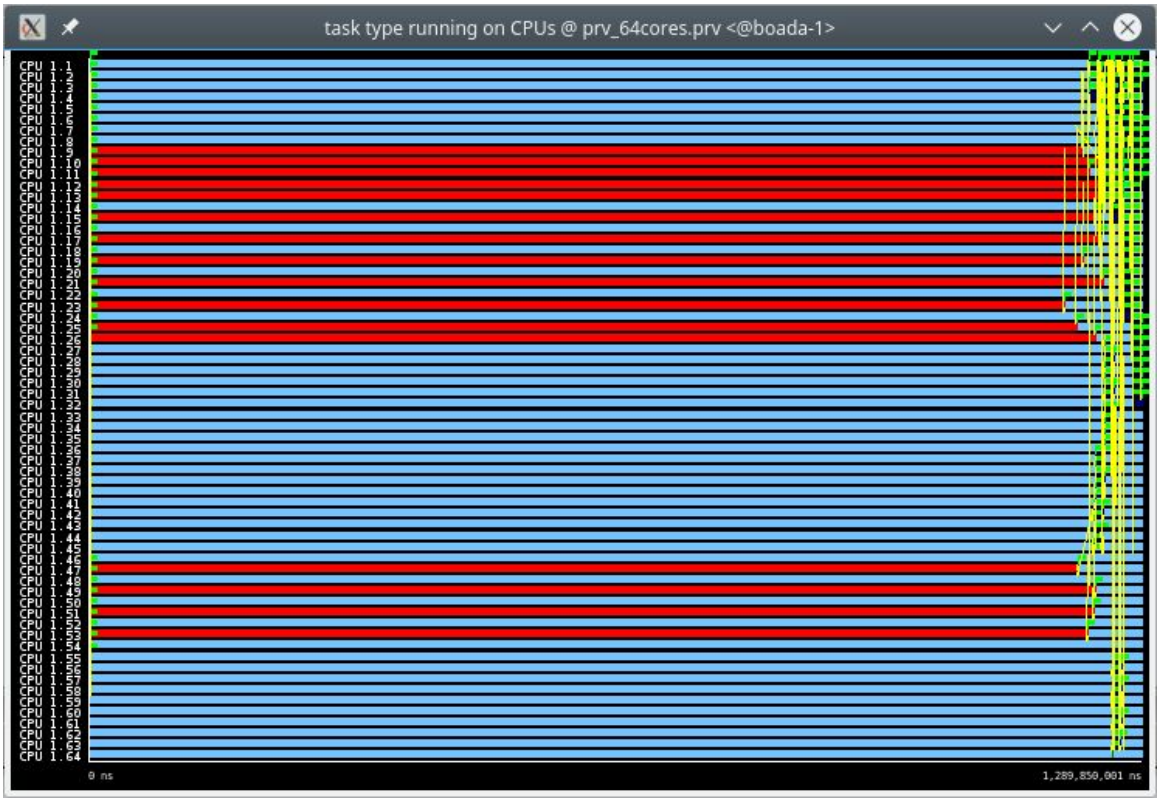

Figure 11, timeline per processor with 64 CPU

Threads	Execution time	Speed-up
1	20334411001	1
2	10173708001	1.998
4	5086801001	3.997
8	2550377001	7.973
16	1289899001	15.764
32	1289850001	15.764
64	1289850001	15.764

Figure 12, table of mandel-tareador.c execution times (in microseconds) and speed-ups

As we can observe, the speed-up increases until reaches the 16 CPUs, that's because the Pmin of our dependence graph is 16, so the most efficient way to parallelize the program is with 16 tasks, more tasks are unworthy.

Moreover, we can appreciate this in the timelines with 32 and 64 CPUs, (Figure 10 and 11, respectively) where only 16 CPUs are used (red) and the others are idle (blue).

2.Parallelization and performance analysis with tasks

In this second part of the deliverable, we are going to parallelize the original sequential code using OpenMP, following the task decomposition analysis that we have conducted in the first part. We will explore two different parallel versions; Leaf and Tree.

Before explain the two versions, it is important to know that each version has the following code in the main section:

```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp);
```

Figure 13, code of the multisort call in the main section

We need these two clauses in order to generate tasks and to have only one implicit task in that part of the code of our versions.

Let's start first with tree version.

This version consists in creating a task for each call in a new level of recursivity, generating parallelism and improving our program. Therefore, we need to do a #pragma omp task at every recursive call in multisort and merge functions. Moreover, we need the #pragma omp taskgroup in order to synchronize the different tasks between multisort and merge, while the last call to merge executes by his own and doesn't need a taskgroup.

```
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);
    }
}
```

Figure 14, code of multisort-omp-tree.c

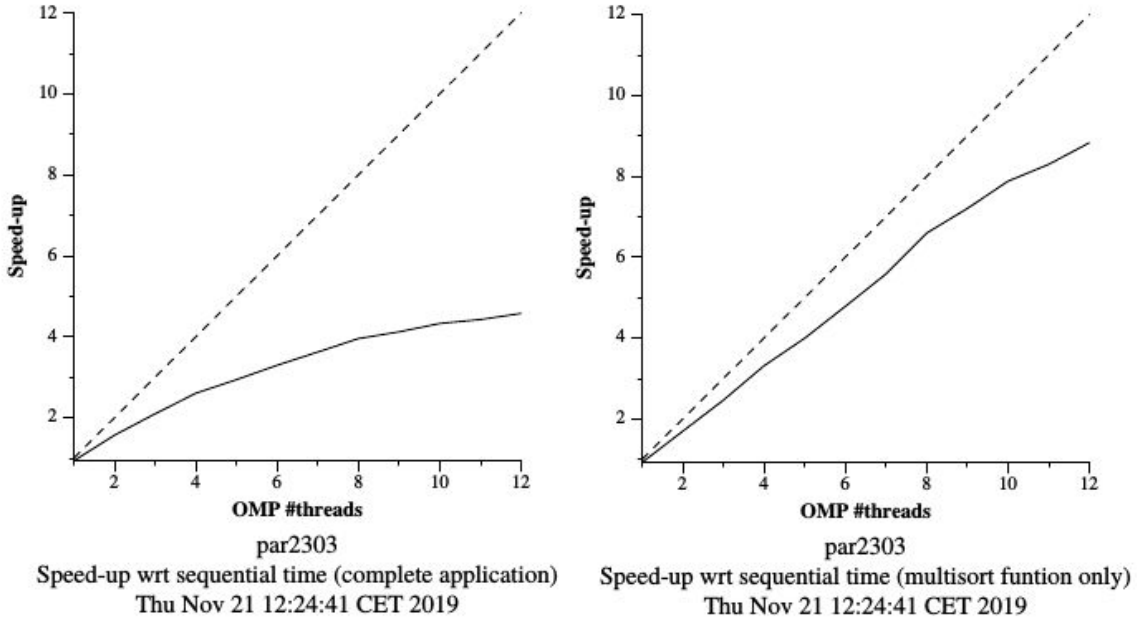

Figure 15, speed–up (strong scalability) plots of multisort-omp-tree.c

As we can observe, in the second plot of the speed-up (the one with the parallelizable part only), has a better speed-up than the complete application. That's because we have 4 recursive calls to multisort (and creates 4 new tasks at every call), so we obtain a higher overhead at synchronization and creation of tasks every iteration.

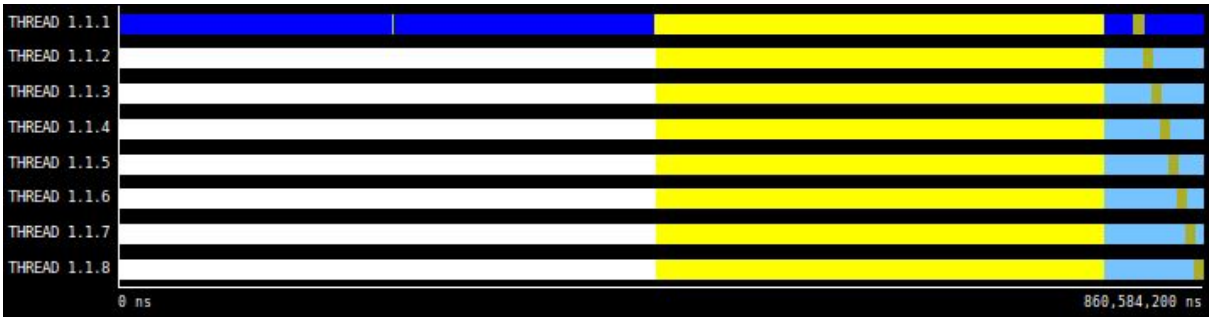

Figure 16, timeline of multisort-omp-tree

Now, let's see how the leaf version behaves.

This version consists in creating a task for each leaf (base case) in the tree generated by the recursive calls. Therefore, we need to add an #pragma omp task at every base case for both functions. The taskwait clauses it's important at the multisort function because we need to do the 4 calls to multisort and then the 2 calls to the merge (recursive calls) and we have to wait all of them.

```
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);
    }
}
```

Figure 17, code of multisort-omp-leaf.c

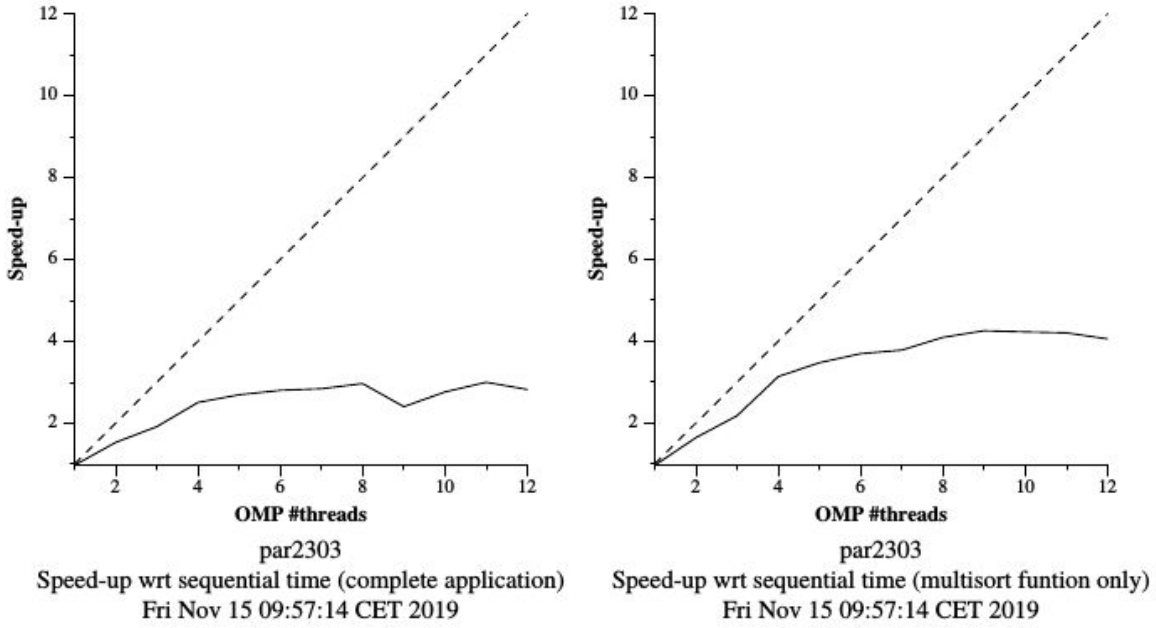

Figure 18, speed–up (strong scalability) plots of multisort-omp-leaf.c

In the leaf strategy, we have a single thread generating all the tasks and a thread executing one task for each leaf reached. In this strategy we have a lot of threads (at leafs) without work to do, and that's the reason we have a bad speed-up (max speed-up with 8 threads).

Figure 19, timeline of multisort-omp-tree

Cutoff version

Finally, we have to include a cut–off mechanism in the Tree version to control the maximum recursion level for task generation. We have to make use of the OpenMP final clause and omp_in_final() which we learned in theory class.

We modify our code including a new parameter depth (initialized to 0 in main) which counts the depth of our tree in order to cut off when depth < CUTOFF. You can see the result in figures 20 and 21.

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

Figure 20, function merge of multisort-omp-tree-cutoff.c

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

                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L,depth+1);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L,depth+1);
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,depth+1);
            }
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 21, function multisort of multisort-omp-tree-cutoff.c

Is there a value for the cut-off argument that improves the overall performance?

In order to ask this question, we are going to obtain the execution time with the value of cut-off = 0, 1, 2, 3, 4, 5.

CUT-OFF = 0

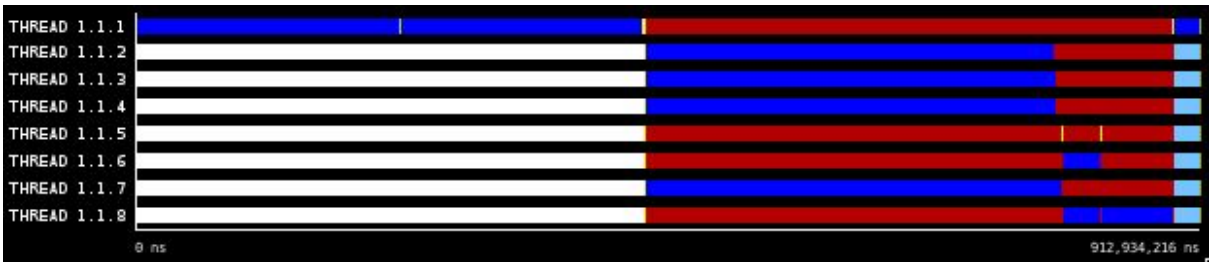

Figure 22, time line of multisort-omp-tree-cutoff with CUT-OFF = 0

CUT-OFF = 1

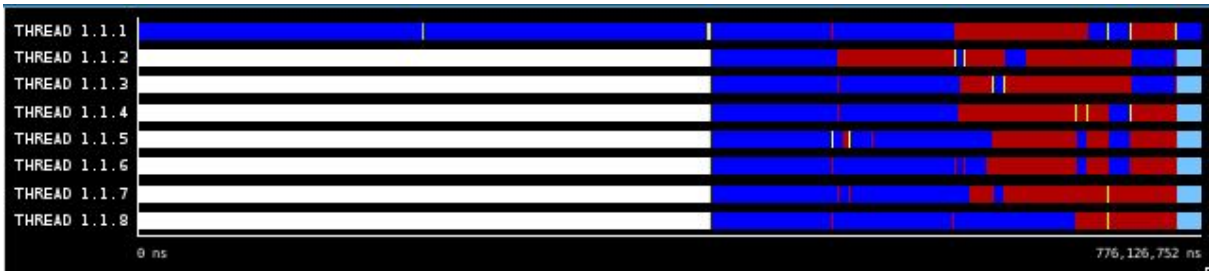

Figure 23, time line of multisort-omp-tree-cutoff with CUT-OFF = 1

CUT-OFF = 2

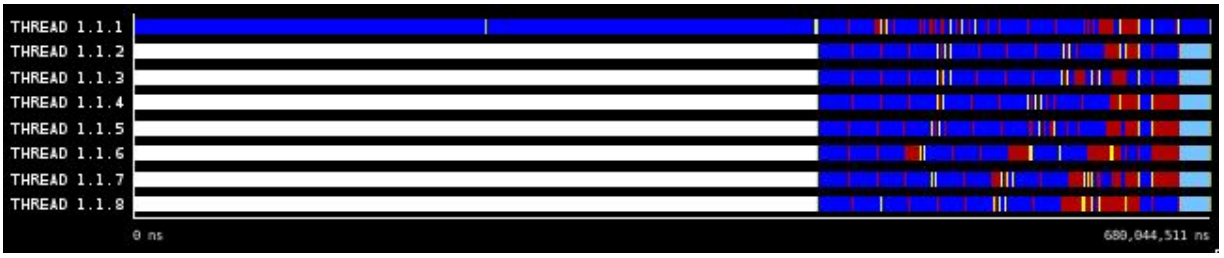

Figure 24, time line of multisort-omp-tree-cutoff with CUT-OFF = 2

CUT-OFF = 3

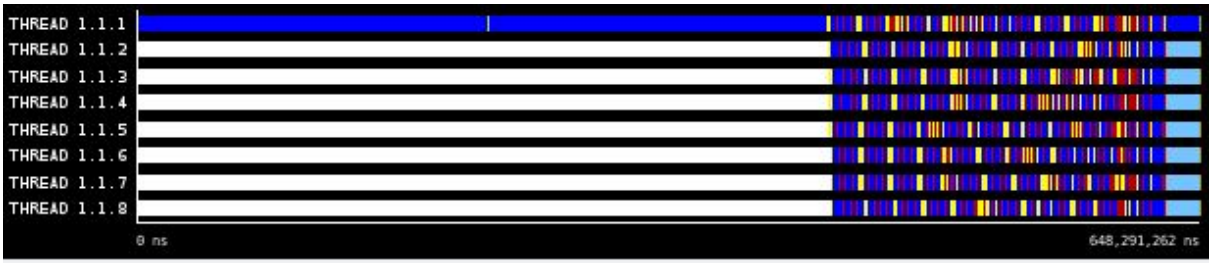

Figure 25, time line of multisort-omp-tree-cutoff with CUT-OFF = 3

CUT-OFF = 4

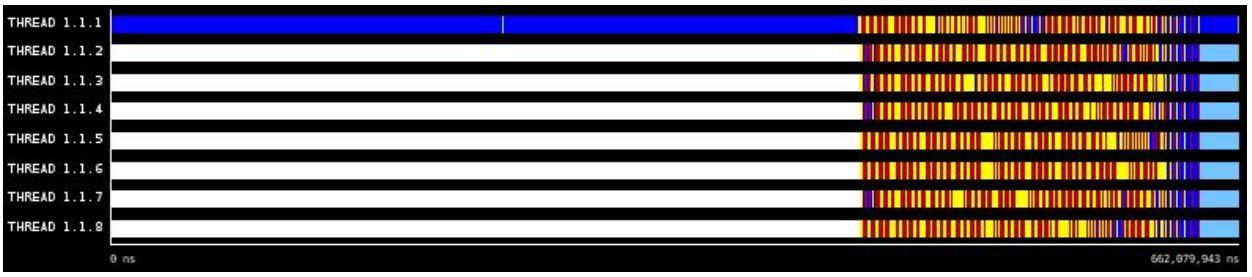

Figure 26, time line of multisort-omp-tree-cutoff with CUT-OFF = 4

CUT-OFF = 5

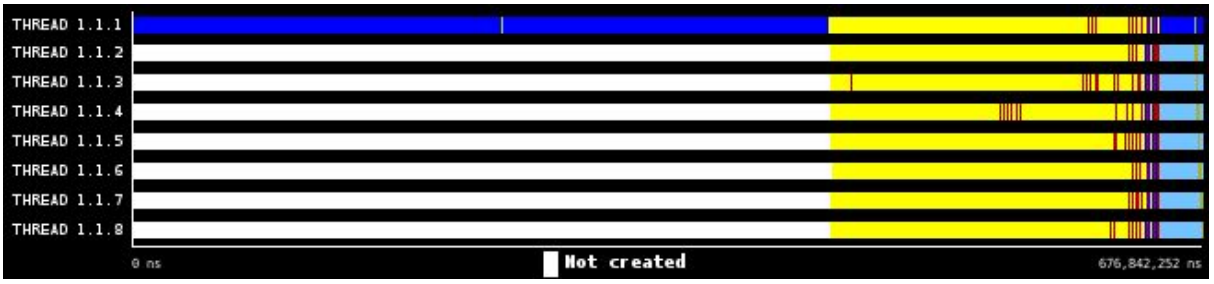

Figure 27, time line of multisort-omp-tree-cutoff with CUT-OFF = 5

As we increase the value of CUT-OFF, we can intuit an improvement at the execution time because we are going deeper in the tree creating new tasks for every recursive call. Therefore, we can observe at the timelines that as we increase the value of CUT-OFF, we create more tasks and that means more overhead in our program (the yellow part). We can see that as well in the generated scalability plots (figure 29), the speed-up improves if we compare it with tree version without cutoff.

Finally, we can see that for the CUT-OFF = 5, we obtain a worse execution time, we know that, at this point, we are going to obtain worse execution times because it is not worth it to create more tasks (overhead increases when we create more tasks).

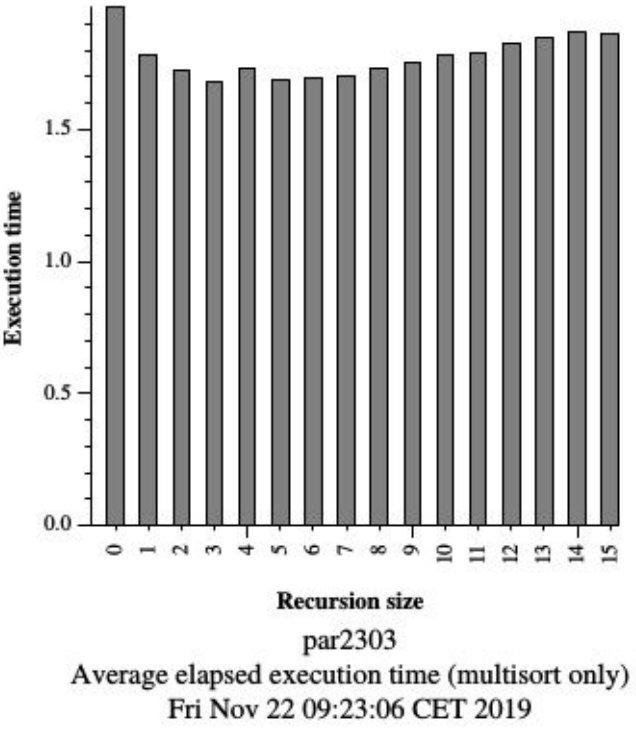

Figure 28, submit-cutoff-omp.sh

Moreover, we found out, looking the generated plot with the submit-cutoff-omp.sh script, that the optimum value of CUT-OFF is 3 (Figure 28) due to the fact that it has the best of the execution times that we are checking.

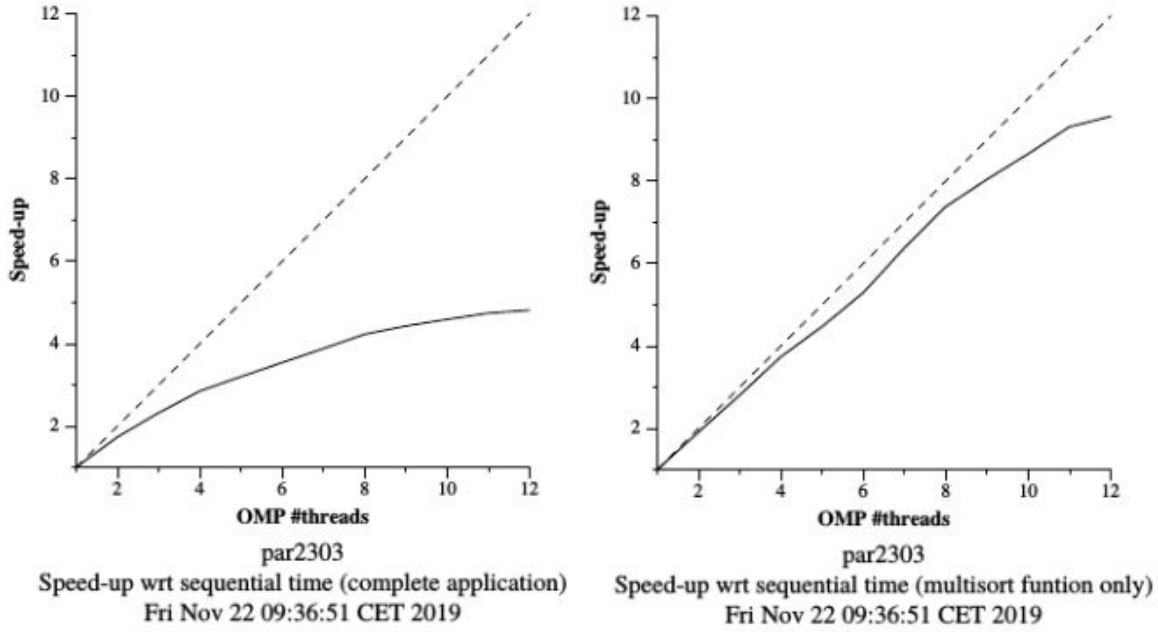

Figure 29, scalability plots with speed-up of cutoff version

3.Parallelization and performance analysis with dependent tasks

In this third part of the deliverable, we are going to change the Tree parallelization of the previous part with the depend clause in order to express dependencies among tasks and avoid some of the taskwait/taskgroup synchronization that we had to introduce.

We only have to modify the multisort function as you can see in the figure below. The merge function continues the same as tree version. (see figure 14)

```
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])
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

            #pragma omp taskwait
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 30, function multisort of multisort-omp-tree-depend.c code

If we compare the speed-up plots of this version (figure 31) with the ones of the tree version (figure 15), we can see that there isn't any significant difference to comment between the plots. Maybe, this version should be a bit better, but we obtain a very similar paraver trace and scalability plots to the tree version.

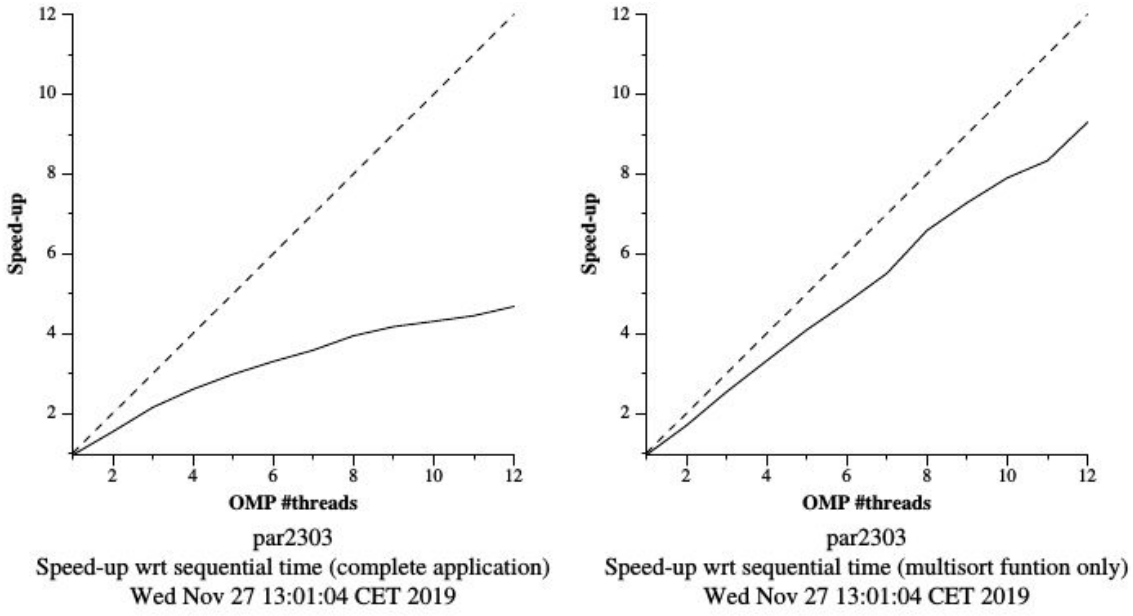

Figure 31, scalability plots with speed-up using dependent tasks

And finally, if we generate the paraver trace, we obtain a very similar timeline (figure 32) as the tree version with taskwait/taskgroup synchronization (figure 16).

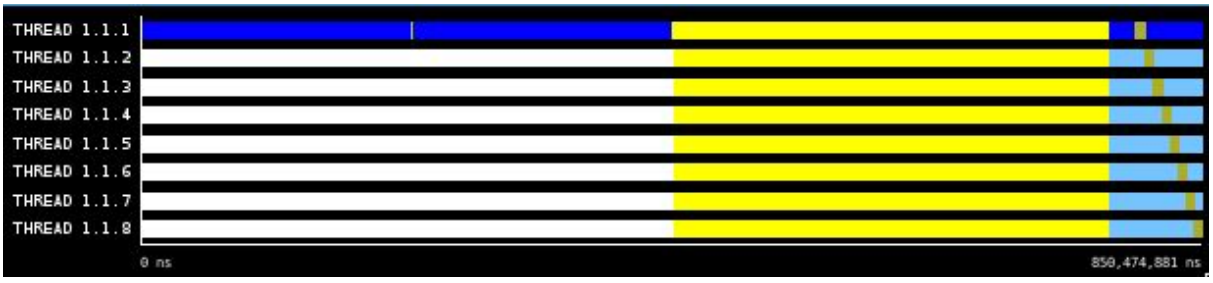

Figure 32, timeline of multisort-omp-tree-depend

So, we can conclude this part of the deliverable saying that the difference between the tree and depend version are insignificant because we obtain a very similar results with depend version.

4.Optional

Optional 1: Complete your scalability analysis for the Tree version on the other node types in boada. Remember that the number of cores is different in boada-1 to 4,boada-5 and boada-6 to 8 as well as the enablement of the hyperthreading capability. Set the maximum number of cores to be used (variable np NMAX) by editing submit-strong-omp.shin order to do the complete analysis. HINT: consider the number of physical/logical cores in each node type and set np NMAX accordingly.

As we know, the boada 1 to 5 has the same number of processors, 12, so If we put the value of np_NMAX to 12, we will obtain the maximum number of cores of each boada.

Firstly, we use qsub -l execution ./submit-strong-omp.sh: (boada 1 - 4)

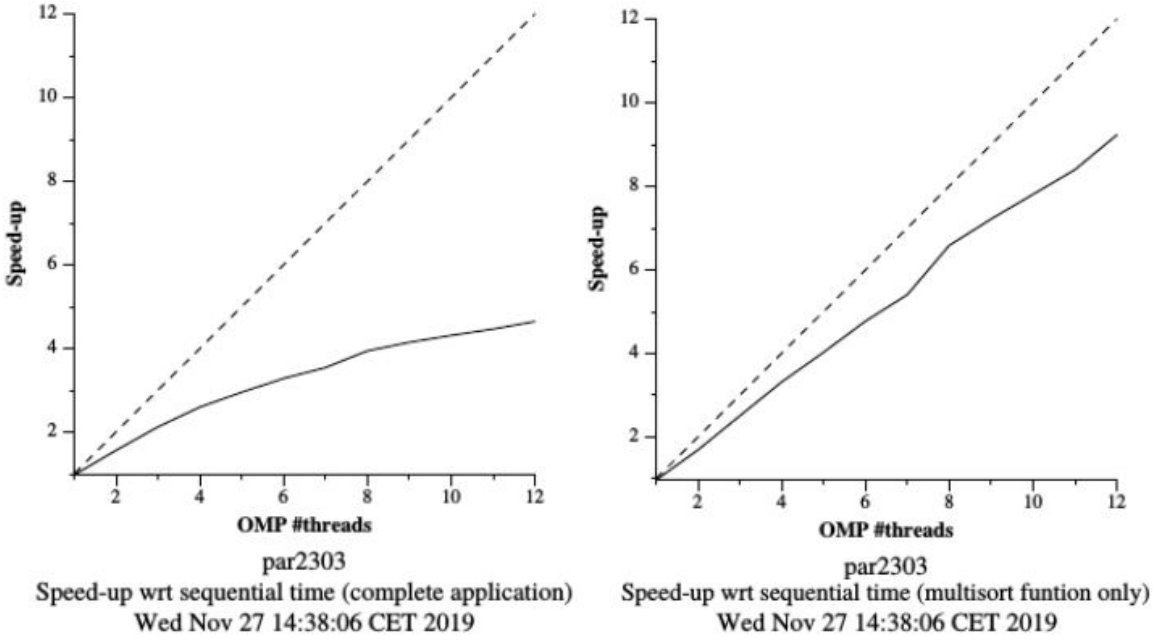

Figure 33, scalability plots with speed-up using qsub -l execution

Secondly, we use qsub -l cuda ./submit-strong-omp.sh: (boada 5)

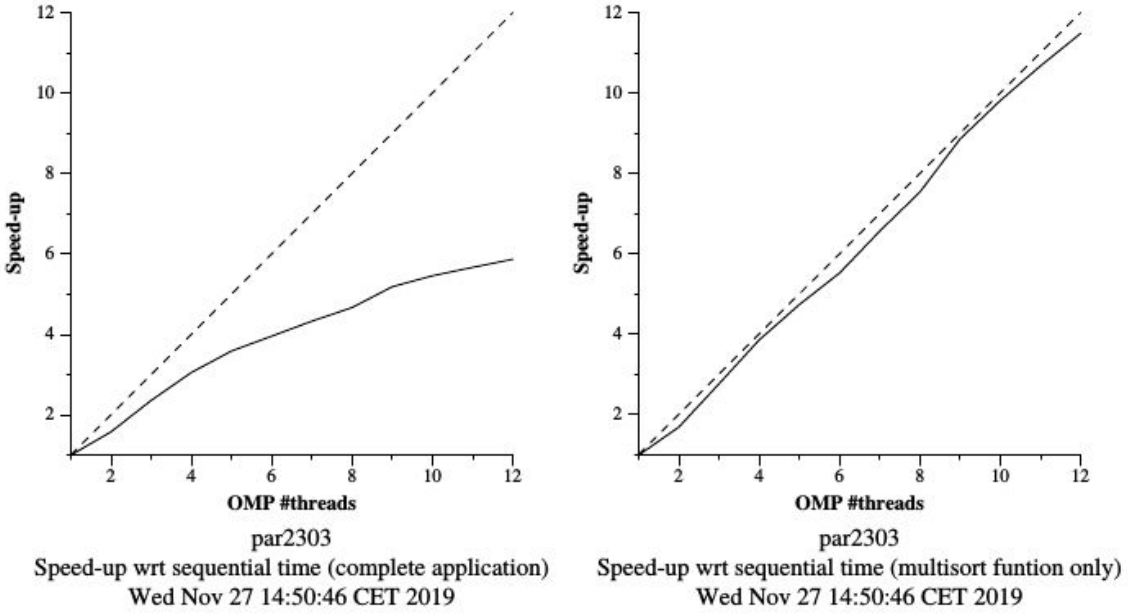

Figure 34, scalability plots with speed-up using qsub -l cuda

Now, for the boada 6-8 we need to change the value to 16, because the processor has 16 cores, so we reach the maximum number of processors.

So, finally, we use qsub -l execution2 ./submit-strong-omp.sh: (boada 6-8)

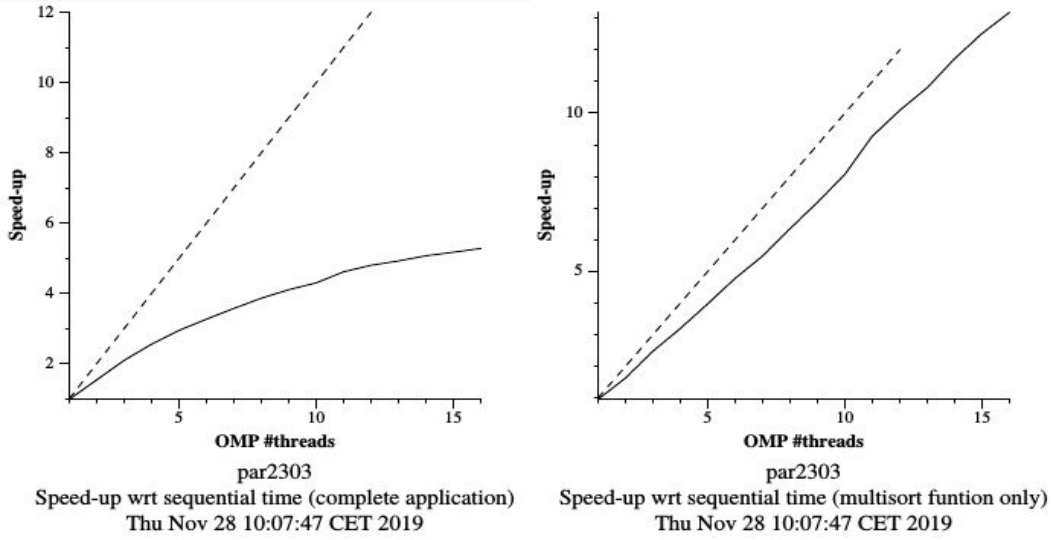

Figure 35, scalability plots with speed-up using qsub -l execution2

As we can observe by the plots obtained, they are very similar between them. So we can corroborate now that the optimum value of np_NMAX is 12 for boada 1-5, and 16 for boada 6-8 (max. num. of processors).

Optional 2: Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors 1. Analyze the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines

```
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;
   }
}
```

Figure 36, parallelization of the initialisation of tmp and data vectors

We can oberve that in the Speed-up plots that this version gives better execution times that the Tree Startegy. That's because the new version makes the code more parallel.

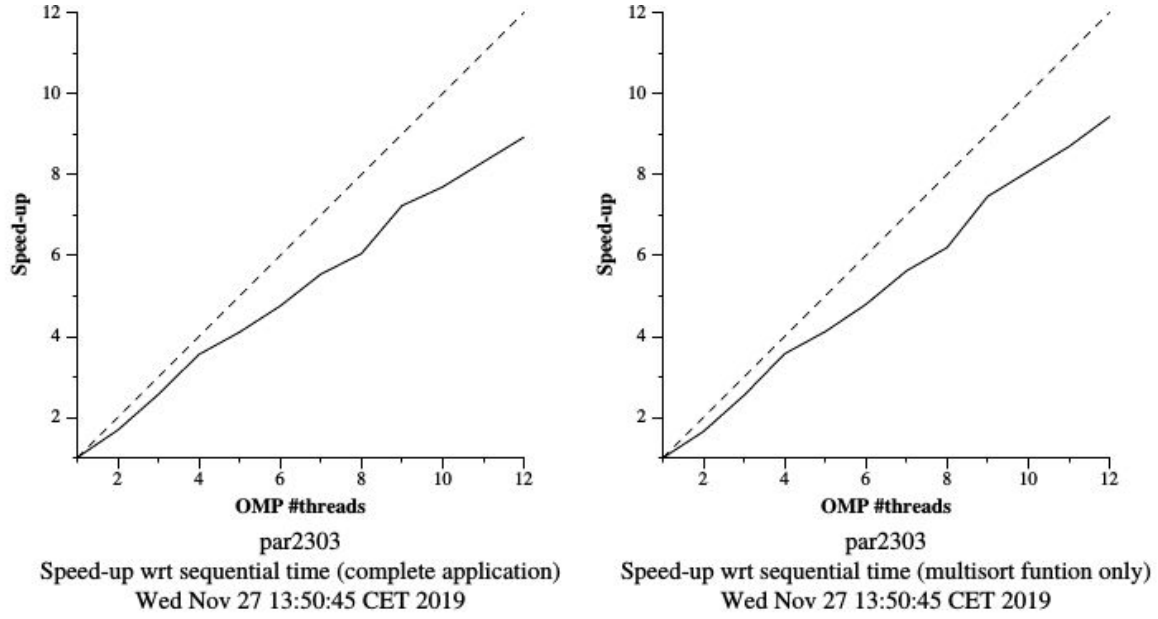

Figure 37, scalability plots with speed-up of the new version of multisort-omp-tree.c

In the paraver timeline we can see that there is less time working on one single thread, this also convince us that the new optional code is better, because it can stay less at the start of the code.

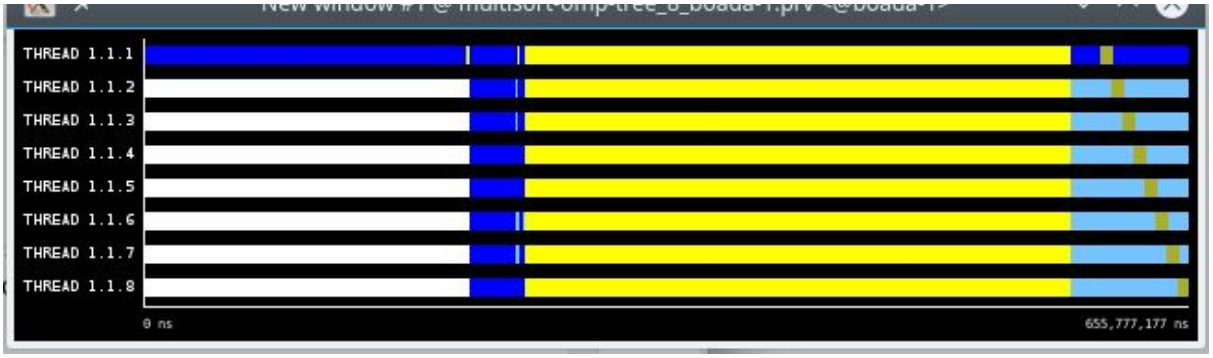

Figure 38, timeline of the new version of multisort-omp-tree.c

5.Conclusions

The aim of this deliverable has been to make use of the different recursive task decomposition strategies, which we have seen in theory class, in a practical example; the multisort program.

Firstly, we have used the tareador tool in order to see the potential parallelism that the ”divide and conquer” strategy provides. For it, we included the creation of tasks in merge and multisort functions of our code and we obtained a task dependency graph to observe that there was recursive and we could improve the execution time making it parallel.

Secondly, to parallelize our code, we used the two recursive task decomposition strategies; Leaf and Tree. Looking at our results in the second part of the deliverable, we can conclude that the tree strategy is always better than the leaf one, if we don’t have too much overhead or many dependencies. Moreover, we could see how the cutoff control behaves in the tree version and see with which value of depth (4) it is not worthy to continue creating tasks in our program because of the overheads.

Finally, we have used another method to parallelize our code without the use of taskwait/taskgroup; using the depend clause. We couldn't see any significant difference in performance between the normal version, but we know that this version is better because the tasks will not suspend until all the tasks finishes as taskwait/taskgroup does.

To sum up, we think that we have learned and have put it into practice interesting things about recursive task decomposition which we hadn’t used it in a real situation, only in theory class.