Divide and Conquer parallelism with OpenMP: Sorting

PARALLELISM 2019/2020 Q1

Alejandro Gallego Rodriguez
Oriol Catalán Fernandez

Group 1203
05/12/2019

Introduction

In this assignment, we will work with a Mergesort algorithm. This algorithm consists of a combination of Divide and Conquer strategy and a sequential quicksort.

The algorithm divides the list into sublists until we achieve a determined size, at this time we use a quicksort algorithm to sort these sublists. When the sorting is completed, the algorithm merges all the sorted sublists into a single sorted list.

Knowing this, during the lab sessions, we will apply different parallelization strategies, seen in theory class, in order to reach a peak performance and see their functioning using different tools such as, Tareador and Paraver.

The parallelization strategies that we will use are: Leaf and Tree parallelization. Also, we will observe the cut-off mechanism in the Tree one and his working behaviour.

Session 1
Task decomposition analysis for Mergesort

1. Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependencies that appear.

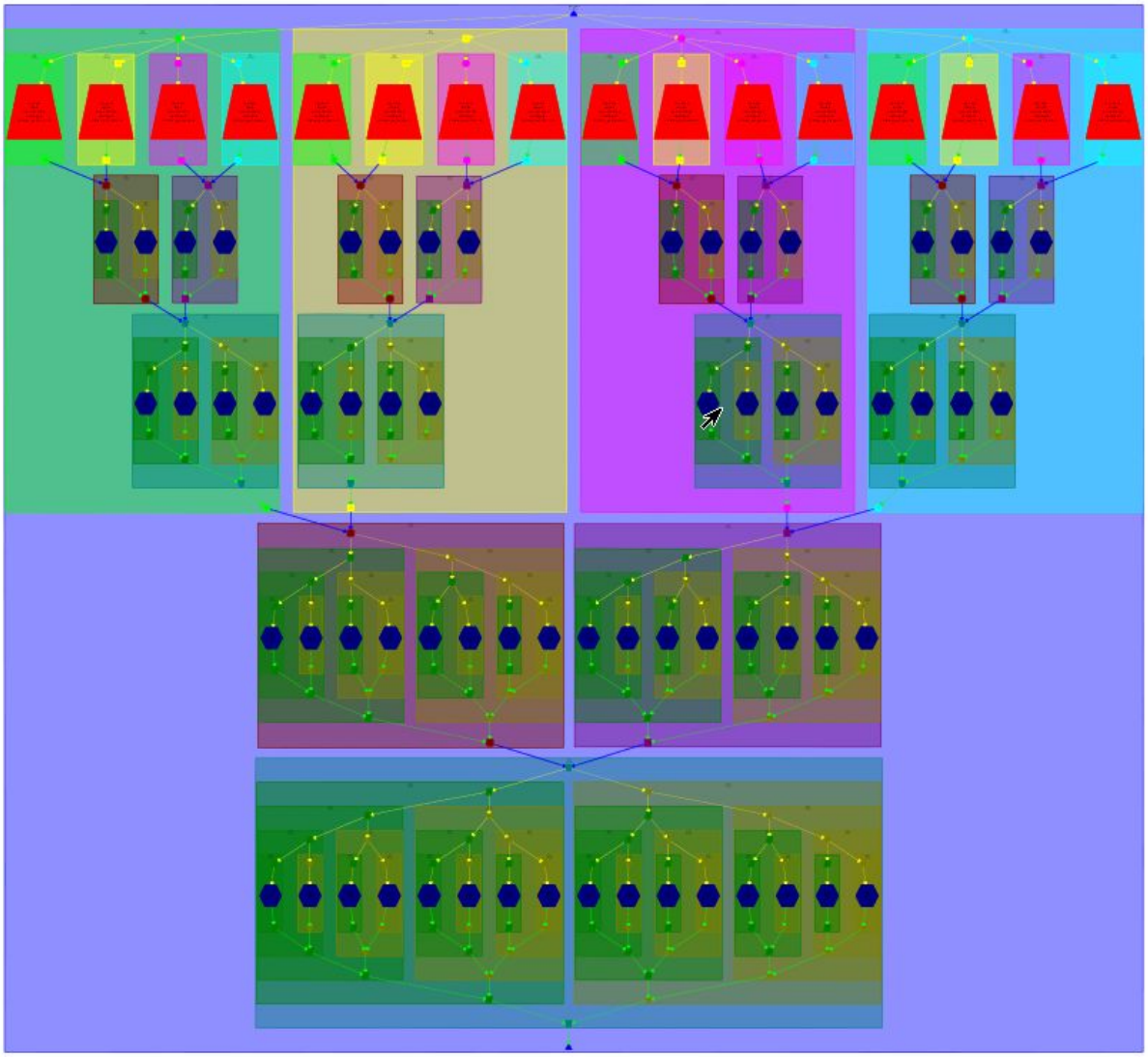


Figure 1.1: Task dependency graph of Mergesort algorithm

```
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
                tareador_start_task("merge11");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("merge11");

                tareador_start_task("merge12");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("merge12");
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("multi1");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("multi1");

                tareador_start_task("multi2");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("multi2");

                tareador_start_task("multi3");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("multi3");

                tareador_start_task("multi4");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("multi4");

                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
        tareador_start_task("cas_base");

        basicsort(n, data);

        tareador_end_task("cas_base");

    }
```

Figure 1.2: Source code of Mergesort algorithm

We have parallelized all of recursive calls making a task for each one. These previous images show the part of the code where he can see the Tareador tasks and the dependency graph associated.

The graph starts with a row of a basic case with 16 tasks, each one calls multisort, that is another task for all of 16 basic case's task. Multisorts calls merge and this function make a lot of calls of merge task to achieve the basicmerge, when we arrive, it's necessary the path in reverse, reaching the father of all the merges of each branch. As we move forward, there are more branches with the same initial pattern until a last call to merge that can reach to the MainTareador again.

2. Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

CPUs	Total Time	Speedup
1	20.334.411.001	-
2	10.173.708.001	1.9987
4	5.086.801.001	3.9975
8	2.550.377.001	7.9731
16	1.289.899.001	15.7643
32	1.289.850.001	15.7649
64	1.289.850.001	15.7649

For each number of CPUs except 32 and 64, we reduce to half practically the previous time, achieving a double speedup from 2 CPUs to 16 CPUs, with 32 and 64, we can see that we reached the maximum speedup, and the time is the smallest, is the time of critical path.

Session 2
Parallelization and performance analysis with tasks

Leaf Parallelisation

```
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
                #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 2.1: Source code Leaf Parallelisation

First of all, we will begin to analyze the behaviour of the algorithm applying a Leaf parallelization strategy.

As we see in the code **(Figure 2.1)** the leaf strategy consists of creating a task once we reach a certain “deep” level. At first, the list suffers a partition into 4 different sublists during n >= MIN_SORT_SIZE*4L. Once the if statement is false, we begin to create tasks and execute them. Also tasks are created at the “merge” function in the base case.

Finally, there are taskwait clauses at the end of the multisort block and the merge block. These clauses are used for synchronisation purposes and to avoid data race condition. The first taskwait, waits for the multisort block to finish and provide a correct set of sublists, that will be used in the merge block. The second taskwait is used to not execute the final merge if one of the functions above is not finished yet because it will result in bad sorting.

Figure 2.2: Timeline with Leaf Parallelisation

The timeline shows pretty well the behaviour of the code. The thread 1.1.1 is responsible of creating the tasks and the other threads execute them and synchronise between them .

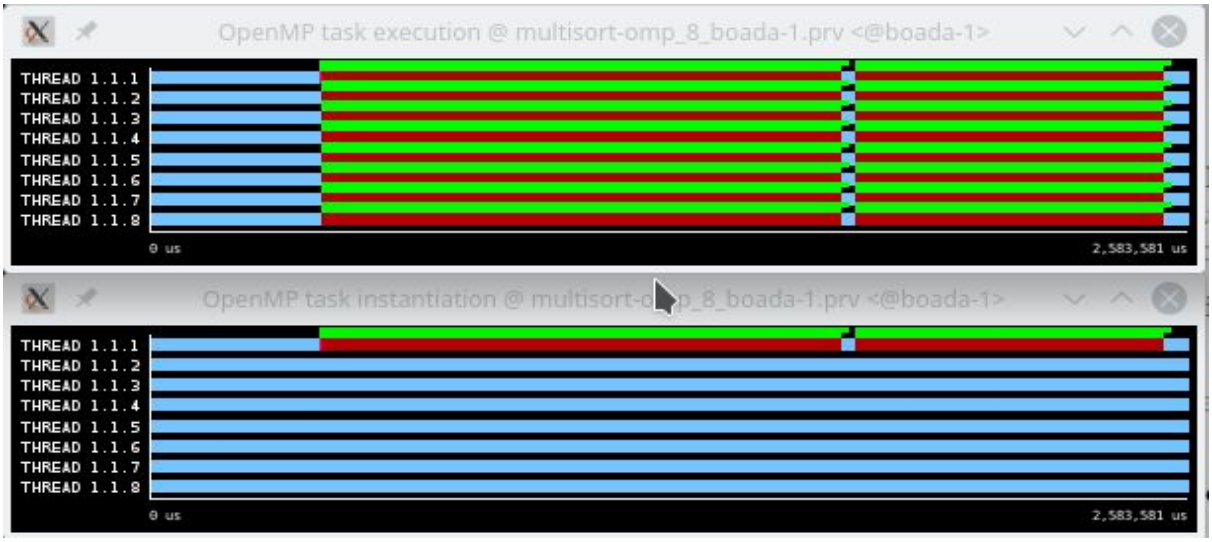
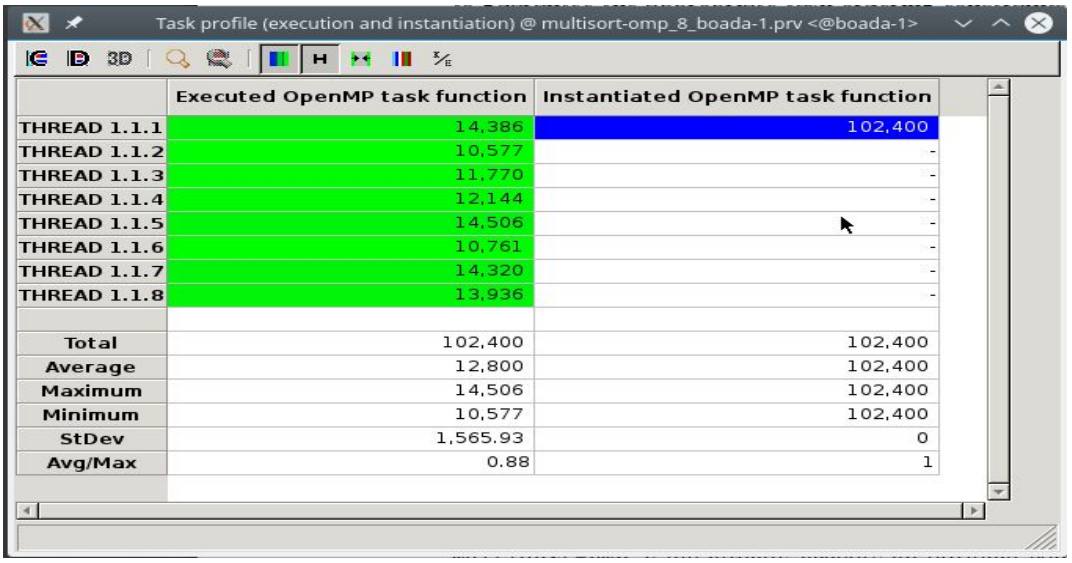


Figure 2.3: Task timeline with Leaf Parallelisation



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	14,386	102,400
THREAD 1.1.2	10,577	-
THREAD 1.1.3	11,770	-
THREAD 1.1.4	12,144	-
THREAD 1.1.5	14,506	-
THREAD 1.1.6	10,761	-
THREAD 1.1.7	14,320	-
THREAD 1.1.8	13,936	-
Total	102,400	102,400
Average	12,800	102,400
Maximum	14,506	102,400
Minimum	10,577	102,400
StDev	1,565.93	0
Avg/Max	0.88	1

Figure 2.4: Task profile Leaf Parallelisation

For the task timeline **(Figure 2.3)** and the task profile **(Figure 2.4)**, we can see that the execution of the tasks is pretty well balanced among the threads and that the only creator of the tasks is the thread 1.1.1 as we saw in the timeline **(Figure 2.2)**

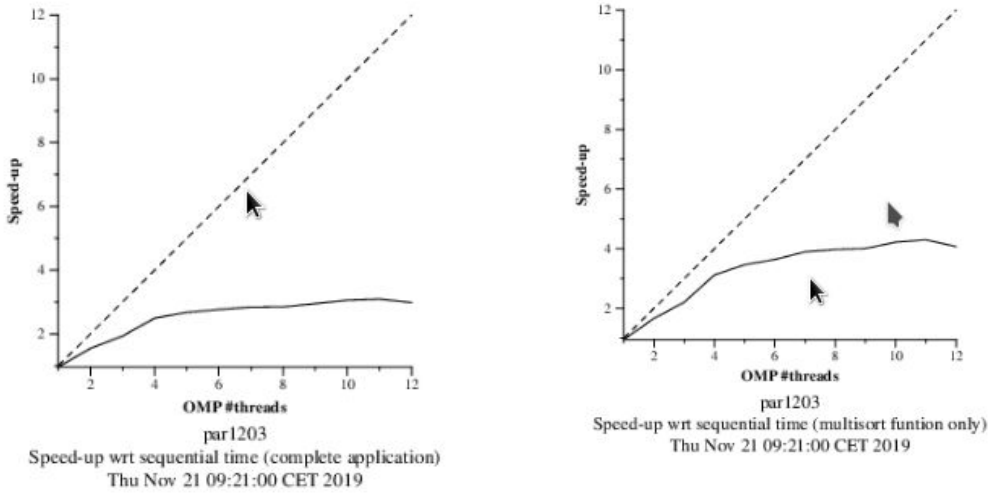


Figure 2.5: Strong scalability plots for Leaf Parallelisation

Both strong scalability plots show us a poor performance in the speed-up with respect to sequential time between the complete application and the multisort function alone.

The first plot makes us understand that with this strategy, no matter what the number of threads is, the speed-up will remain static. The second plot gives us a similar response.

The results make sense with the conduct of the leaf strategy: the execution of the sort algorithm will be sequential until we reach a determined “deep” level, only then the parallel section will begin. With this in mind, we can say that the increment of threads is not profitable and we can not take advantage of it.

These results leave us the outcome that leaf parallelisation is not the best strategy to approach this problem.

Tree Parallelisation

```
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);
        }
}

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 2.6: Source code Tree Parallelisation

The next version we will analyze is made with a Tree parallelisation strategy. As we can see in the code **(Figure 2.6)**, this strategy consists of creating tasks at the recursive level of the sorting algorithm, parallelising the creation of sublists and their merging, and a sequential execution once we reach a “deep level”.

In the recursive case, we have created two taskgroup clauses that involve the multisort block and the merge block, respectively. The taskgroup clause has an implicit barrier at the end of its block so it will wait for all the tasks inside the block to finish before continuing the execution.

Both taskgroup clauses will help us prevent data race condition and provide a correct sorting result.

You will notice that the final merge does not have a taskgroup clause. This is because once we reach that level the sublists that are passed by arguments are correct because of the previous taskgroup.

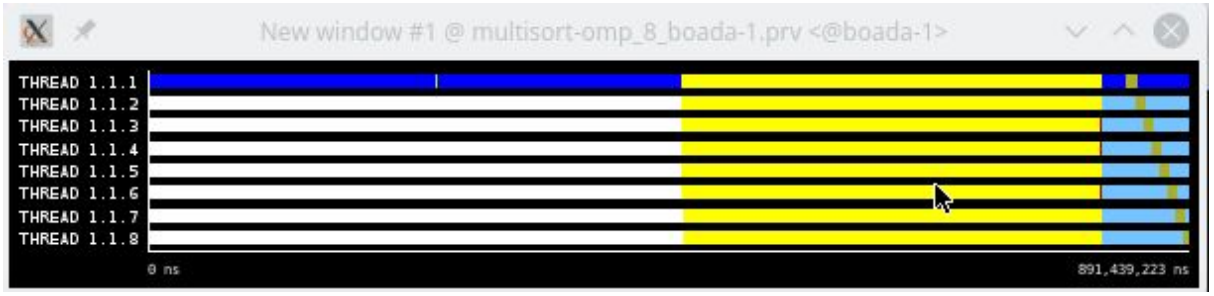


Figure 2.7: Timeline with Tree Parallelisation

This timeline shows us that the creation of tasks and their execution is divided among all threads. At the end we can see the sequential section being executed by thread 1.1.1.

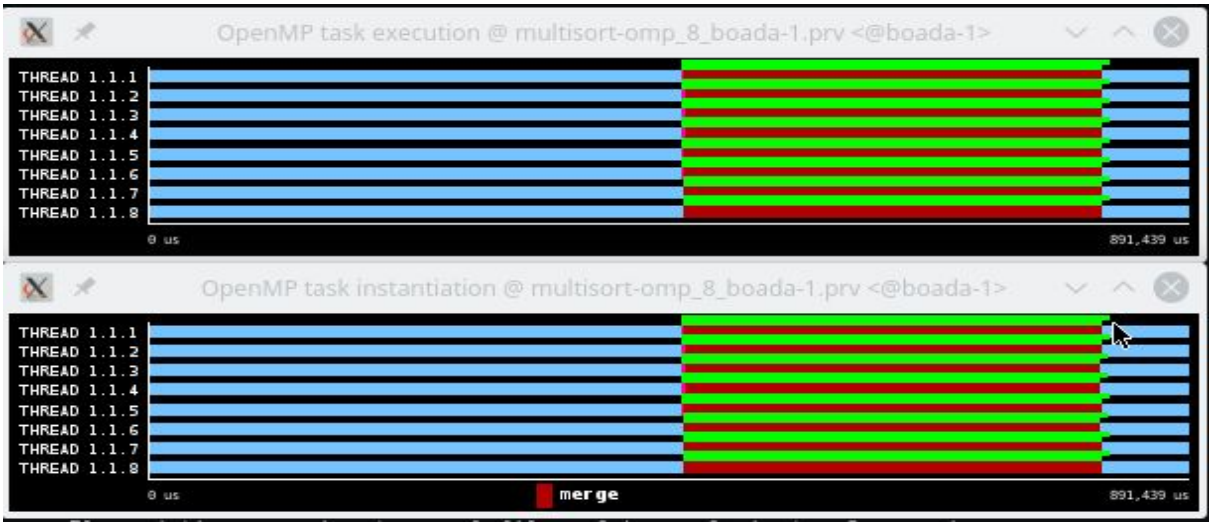
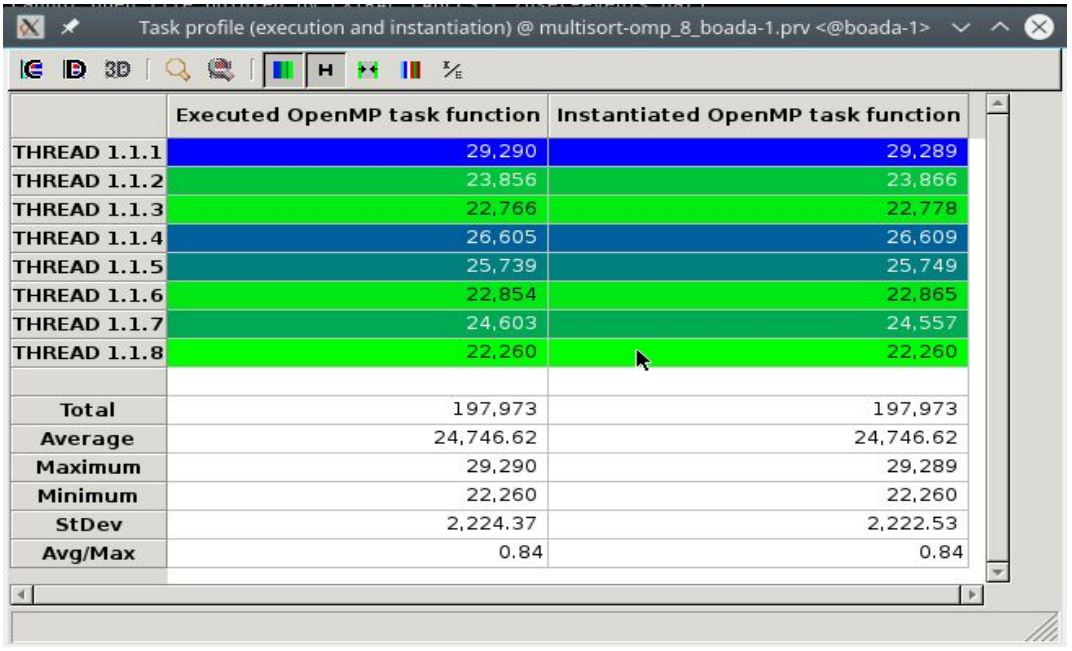


Figure 2.8: Task timeline Tree Parallelisation



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	29,290	29,289
THREAD 1.1.2	23,856	23,866
THREAD 1.1.3	22,766	22,778
THREAD 1.1.4	26,605	26,609
THREAD 1.1.5	25,739	25,749
THREAD 1.1.6	22,854	22,865
THREAD 1.1.7	24,603	24,557
THREAD 1.1.8	22,260	22,260
Total	197,973	197,973
Average	24,746.62	24,746.62
Maximum	29,290	29,289
Minimum	22,260	22,260
StDev	2,224.37	2,222.53
Avg/Max	0.84	0.84

Figure 2.9: Task profile Tree Parallelisation

Both task timelines **(Figure 2.8)** and task profile **(Figure 2.9)**, provide information about the execution of tasks and its instantiation. We can see that the execution of the tasks is well balanced among the threads. Also, unlike the leaf parallelisation, all threads contribute to the creation of tasks as we said before.

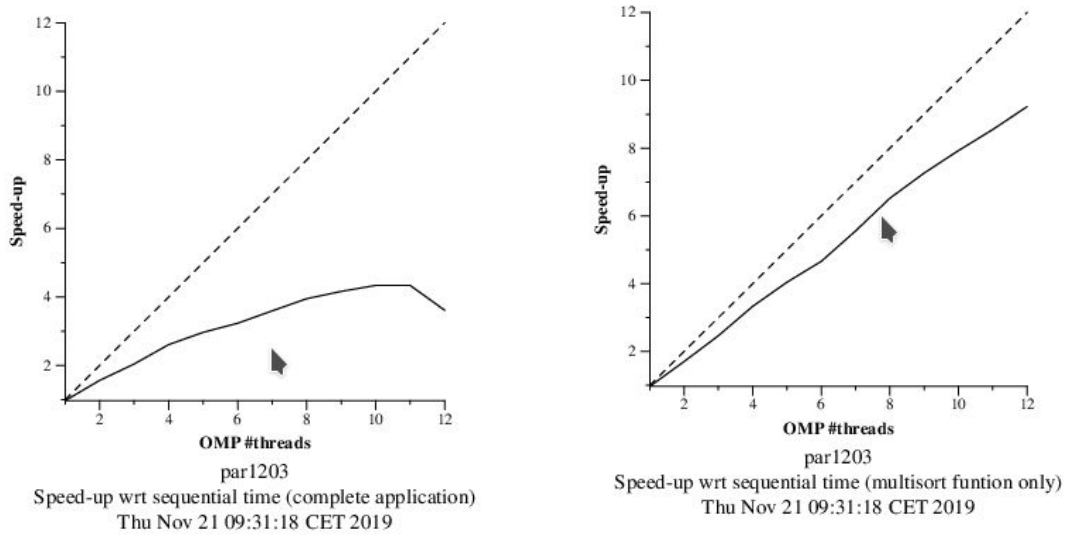


Figure 2.10: Strong scalability plots Tree Parallelisation

The speed-up with respect to sequential time of the multisort function provides us a very different result than the one seen in the leaf parallelisation. Now, with the increase of the number of threads, the sequential time decreases.

This concordes with the behaviour of the tree parallelisation that takes more advantage of the number of threads available and is more parallelizable.

On the other hand, in the left plot, the conduct is pretty similar with the leaf one. There is a slight improvement but not a significant one. This means that the majority of the application is executed sequentially.

Tree Parallelisation with CUT-OFF mechanism

```
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 taskgroup
                {
                #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 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 2.11: Source code with cut-off mechanism

The next version we will analyze is a subversion of the tree parallelisation strategy. We will add a cut-off mechanism into the original tree version.

The cut-off mechanism consists of stopping the creation of tasks in a certain moment. In the code above **(Figure 2.11)** we see an if condition declaring that if the task is “final” the multisort block stops being executed.

A task is “final” if the final clause that we implement at the pragma is true. In this clause we use a CUTOFF variable that will represent the moment to stop creating tasks in division of the list. Not only a CUTOFF variable is needed, but we also provide a new argument to the definition of the function multisort that will represent the “deep” recursive level we are and which will increase in each recursive call.

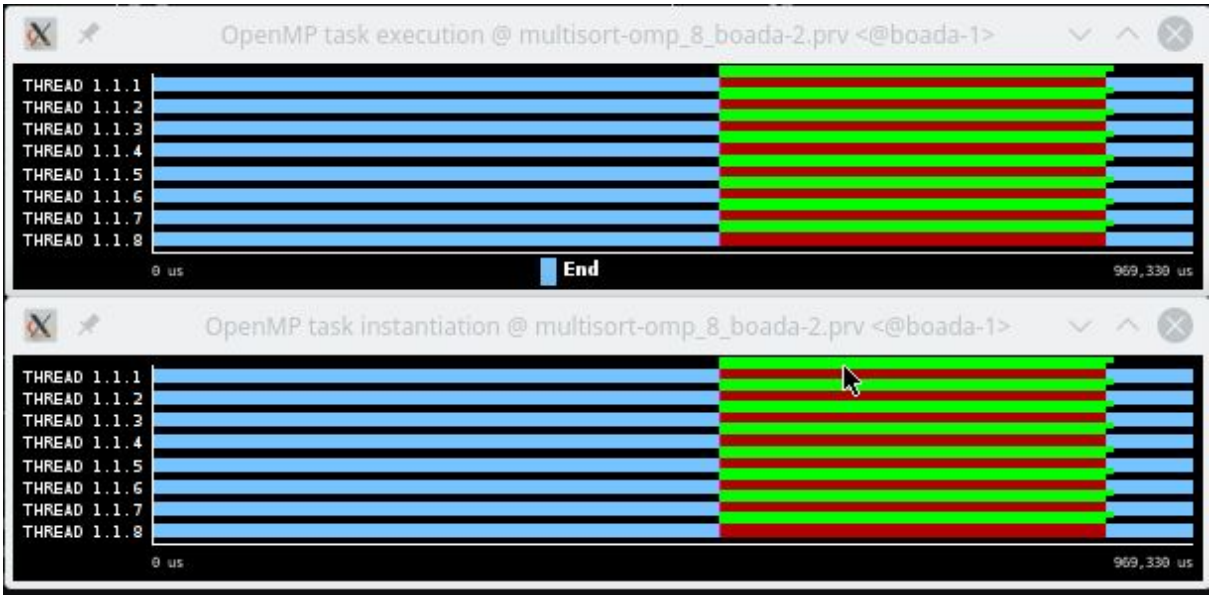
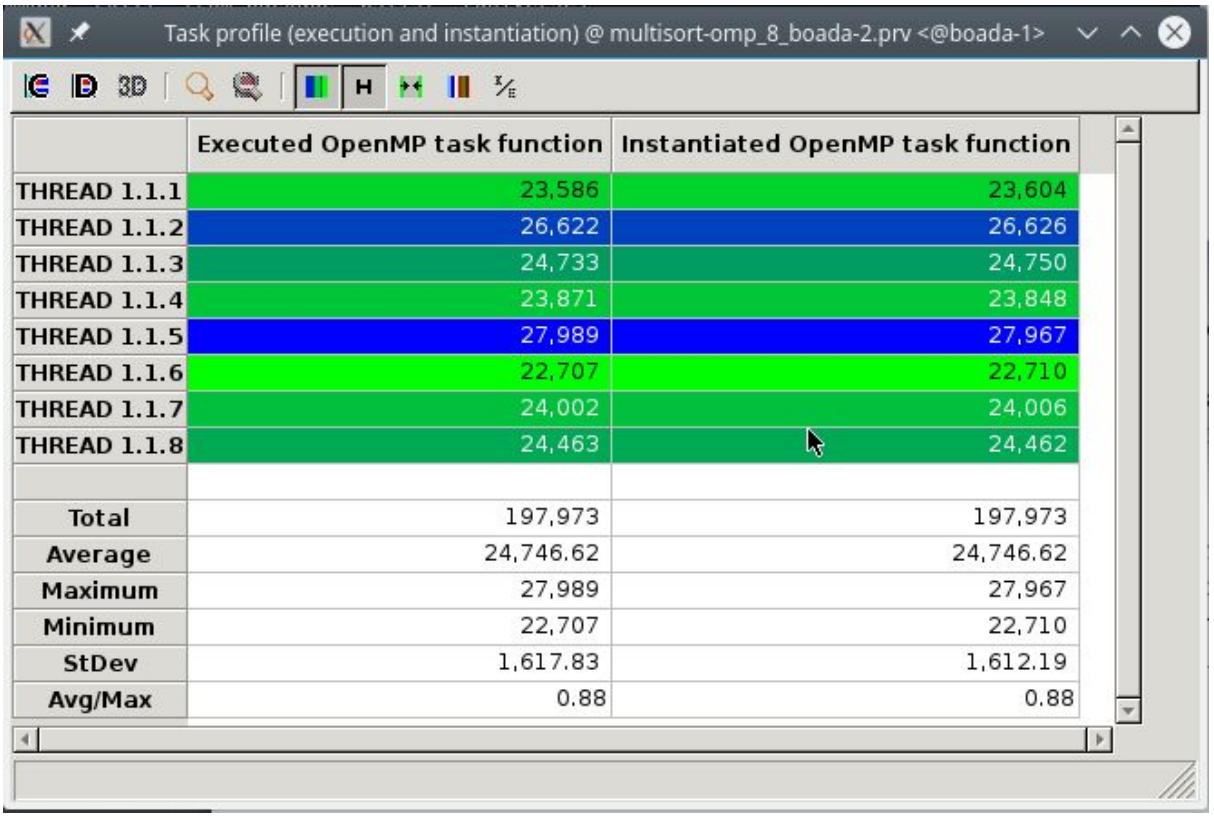


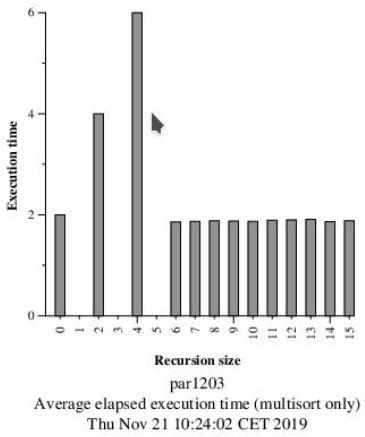
Figure 2.12: Task timeline with cut-off mechanism



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	23,586	23,604
THREAD 1.1.2	26,622	26,626
THREAD 1.1.3	24,733	24,750
THREAD 1.1.4	23,871	23,848
THREAD 1.1.5	27,989	27,967
THREAD 1.1.6	22,707	22,710
THREAD 1.1.7	24,002	24,006
THREAD 1.1.8	24,463	24,462
Total	197,973	197,973
Average	24,746.62	24,746.62
Maximum	27,989	27,967
Minimum	22,707	22,710
StDev	1,617.83	1,612.19
Avg/Max	0.88	0.88

Figure 2.13: Task profile with cut-off mechanism

The task timeline execution and instantiation **(Figure 2.12)** provides us a similar output to the tree without cut-off one. In the task profile **(Figure 2.13)** we see variation in the number of tasks executed and created in each thread.



2.14: Plots for different cut-off sizes and the execution time

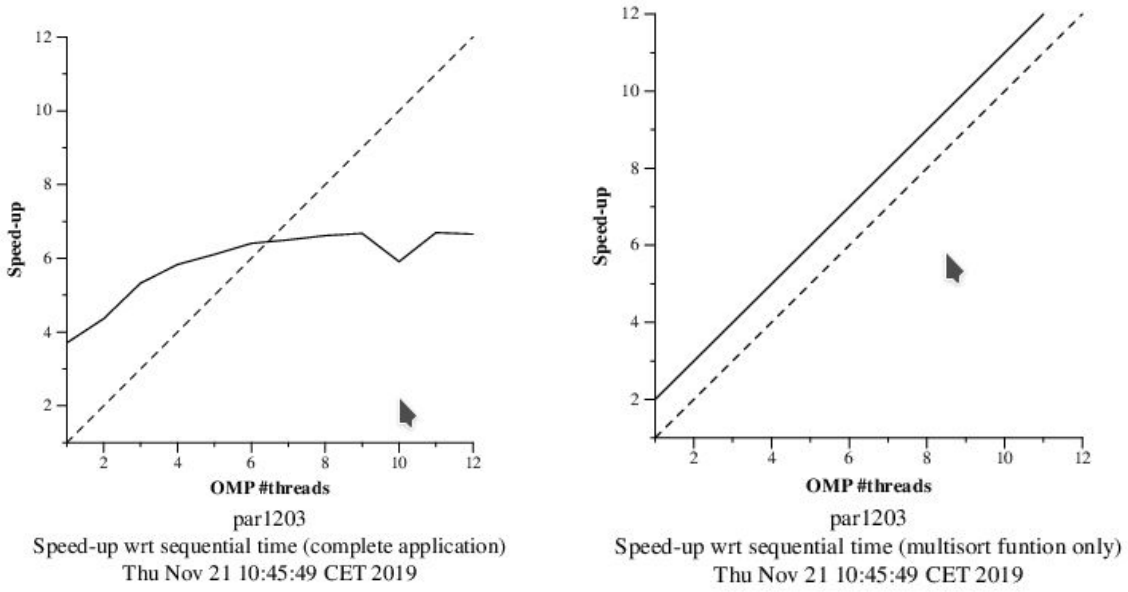


Figure 2.15: Strong scalability with cut-off mechanism

Session 3
Parallelization and performance analysis with dependent tasks

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp taskgroup
                {
                #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 taskgroup
                {
                #pragma omp task depend(in: tmp[0], tmp[n/2L])
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                }
        } else {
                // Base case
                basicsort(n, data);
        }
```

Figure 3.1: Source code Tree parallelization with dependency clauses

In this version of code, we can see that is made with Tree parallelization strategy but is made with dependency clauses, with pragma sentence depend in/out. In class, we had to implement this version of code deleting one of the barrier that we had in first Tree parallelization version. This version is faster and more efficient because the tasks with dependencies only wait his tasks, not all of the tasks of the higher level, and this is a great improvement.

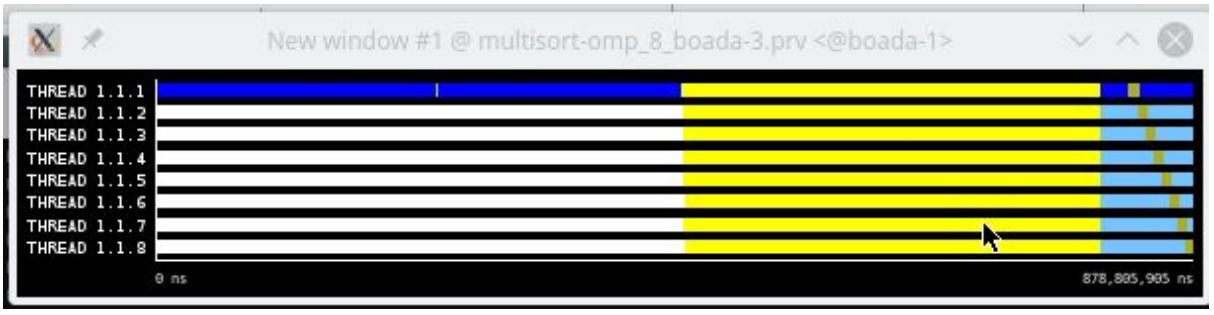


Figure 3.2: Timeline Tree parallelization with dependency clauses

This timeline shows us that the creation of tasks and their execution is divided among all threads. At the end we can see the sequential section being executed by thread 1.1.1.

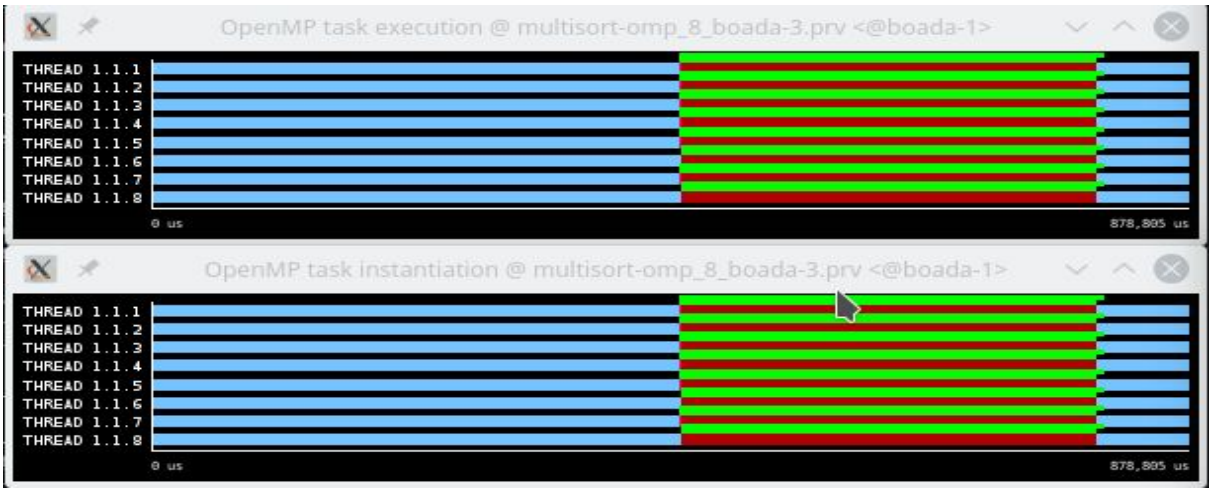
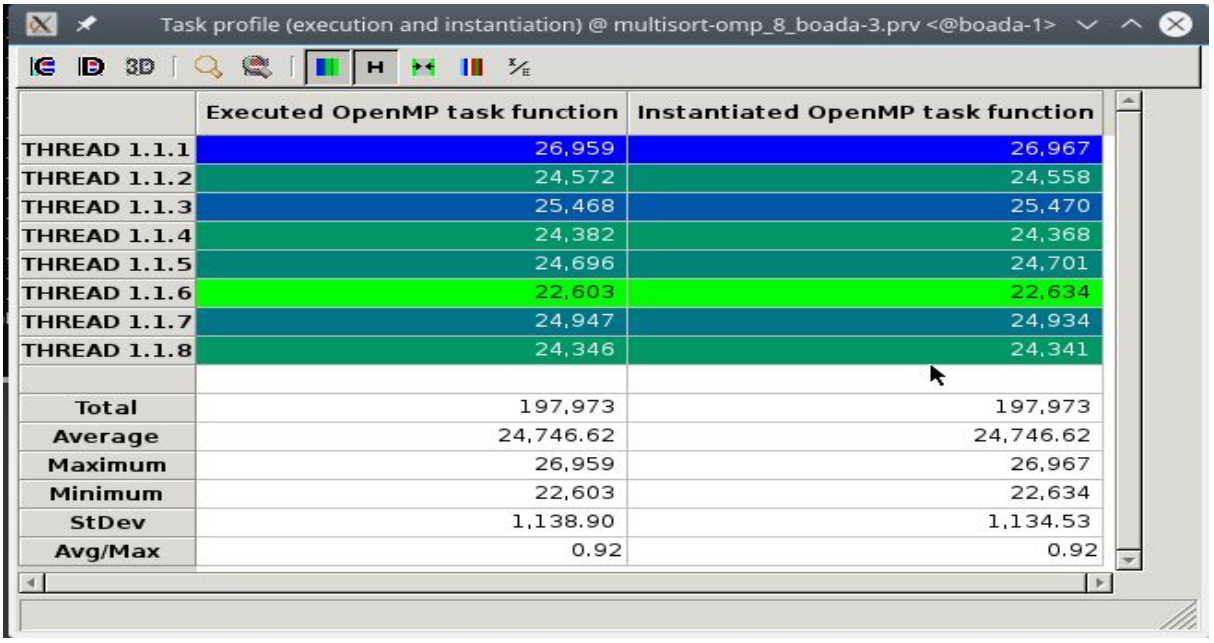


Figure 3.3: Task timeline Tree parallelization with dependency clauses



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	26,959	26,967
THREAD 1.1.2	24,572	24,558
THREAD 1.1.3	25,468	25,470
THREAD 1.1.4	24,382	24,368
THREAD 1.1.5	24,696	24,701
THREAD 1.1.6	22,603	22,634
THREAD 1.1.7	24,947	24,934
THREAD 1.1.8	24,346	24,341
Total	197,973	197,973
Average	24,746.62	24,746.62
Maximum	26,959	26,967
Minimum	22,603	22,634
StDev	1,138.90	1,134.53
Avg/Max	0.92	0.92

Figure 3.4: Task profile Tree parallelization with dependency clauses

These two previous figures **(Figure 3.3 and 3.4)** provides us with very similar information in front of previous tables and timelines of Tree strategy. The execution of tasks is balanced and several values are exactly the same as previous task profile.

OPTIONAL 1

This optional assignment consists of watching the differences in the execution of the mergesort algorithm, with tree parallelization strategy, in different nodes of the boada.

BOADA-1 TO BOADA-4

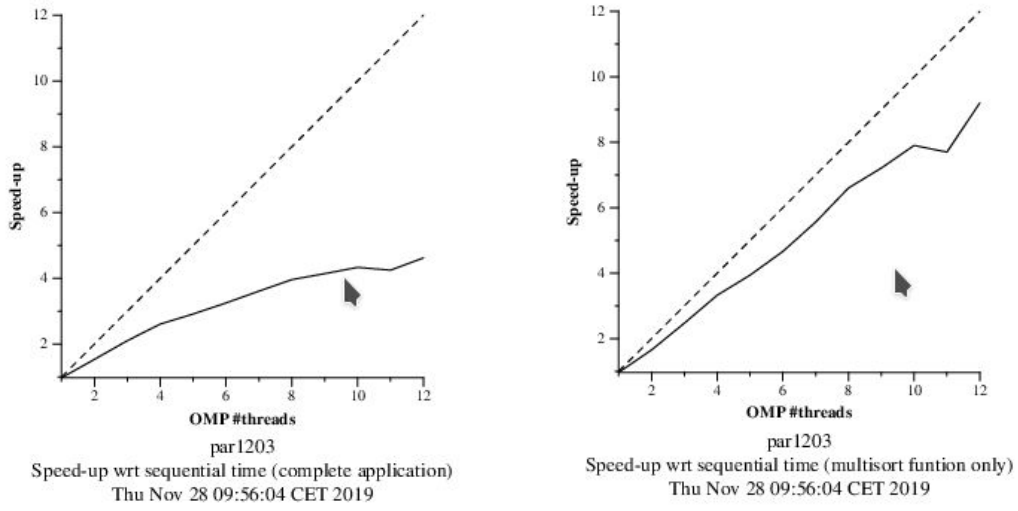


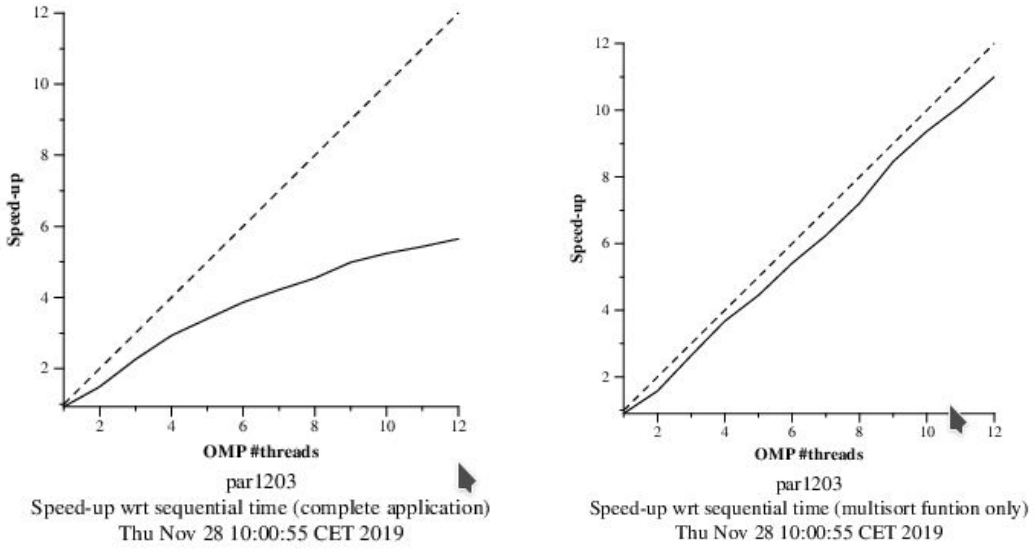
Figure 4.1: Strong scalability plots for boada-2

First of all, we will begin with the set of nodes that goes from boada-1 to boada-4. As we previously in the first lab session, this node has a total of 12 cores (processors), distributed in two different sockets. These cores have a core frequency of 2395 MHz.

The speed-up plots are similar to the one seen in the tree parallelisation strong scalability plots, this is because we have used this set of nodes in our previous analysis.

This set of nodes can be used using the keyword “execution” in the qsub command.

BOADA-5



4.2: Strong scalability plots for boada-5

The next node we will analyze is the boada-5 one. This node has the same cores as the boada-1/4 has. However, we see that the plots show a much better performance than the other one.

This improvement is due to the core frequency in this node that is 2600 Mhz, superior to the others sets.

This node can be used using the keyword “cuda” in the qsub command.

BOADA-6 TO BOADA-8

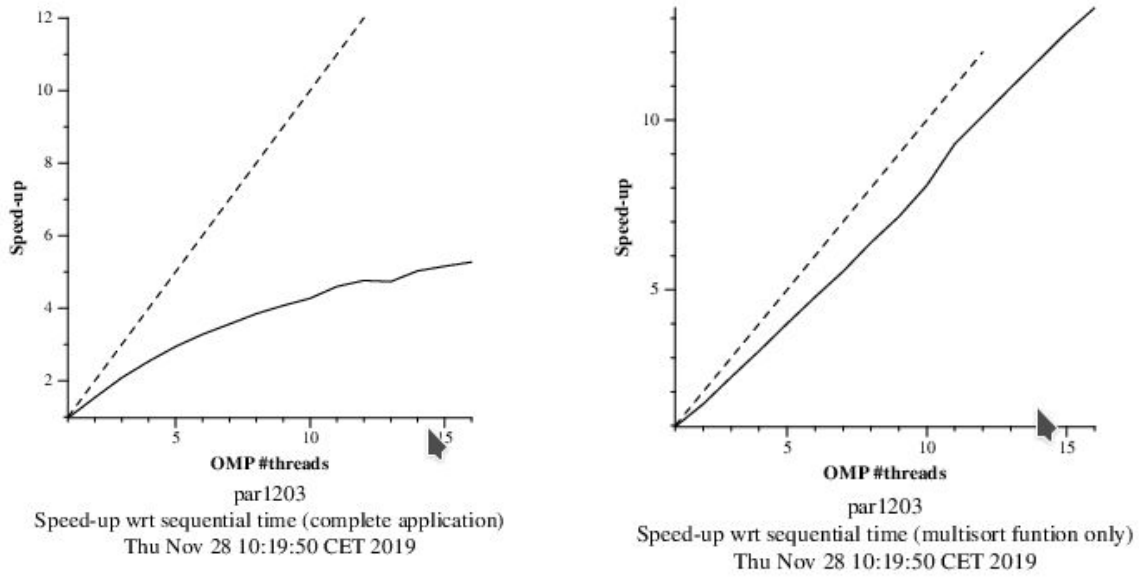


Figure 4.3: Strong scalability plots for boada-6

Finally, we will analyze the set of nodes that is compressed from boada-6 to boada-8. These set of nodes have significant differences with the other ones. The number of cores ascend to 18, divided among 2 sockets and 1 thread per core.

The strong scalability plots gives us a good performance on the execution. This happens because of the higher number of cores. Even though, the number of cores is superior, boada-5 continues to have a better performance because the core frequency of this set of nodes is 1700 MHz, very low compared to the previous ones.

This set of nodes can be used using the keyword “execution2” in the qsub command.

OPTIONAL 2

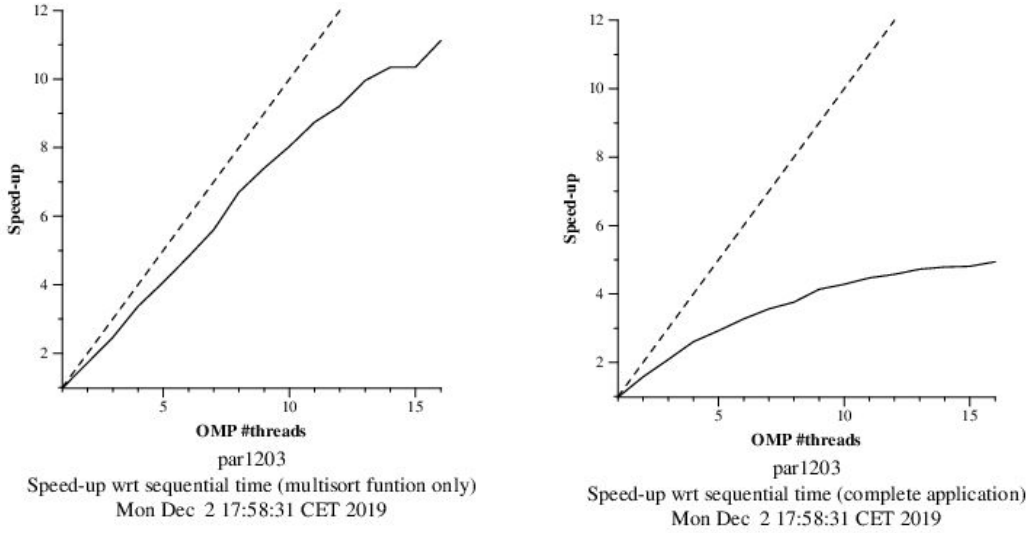


Figure 5.1: Strong scalability plots of version with parallel loops

```
static void initialize(long length, T data[length]) {
   long i;
   #pragma omp for schedule (static,1)
   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 for schedule (static,1)
   for (i = 0; i < length; i++) {
      data[i] = 0;
   }
}
```

Figure 5.2: Code of version with parallel loops

This version of the code implements the Tree parallelization strategy and also parallelize the two loops, one that initialize the data vector, and the other clear the tmp vector; and they prepare the two vectors to the multisort execution. As they are two loops, we can parallelize with the pragma sentence schedule, in this case we had used the static schedule with chunk = 1.

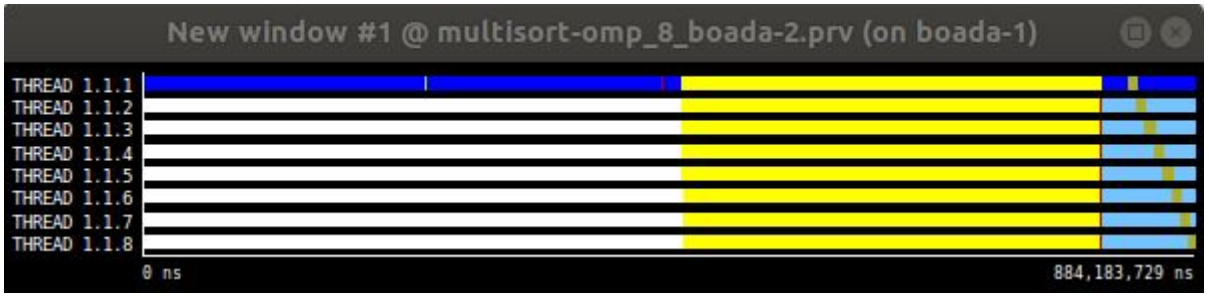


Figure 5.3: Timeline of version with parallel loops

This timeline is very similar to the other with Tree strategy, all threads are involved in creation and execution of the tasks.

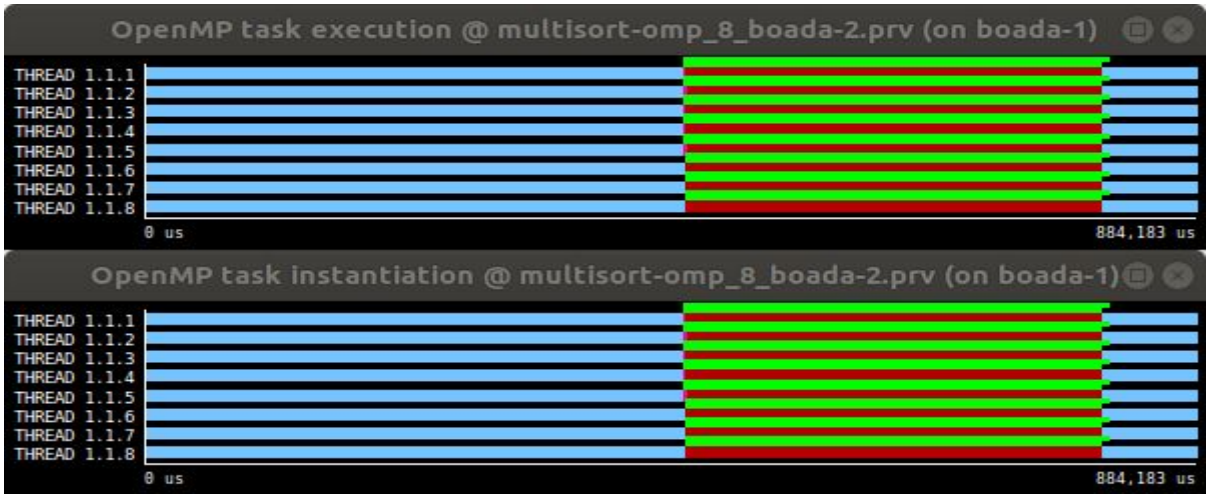
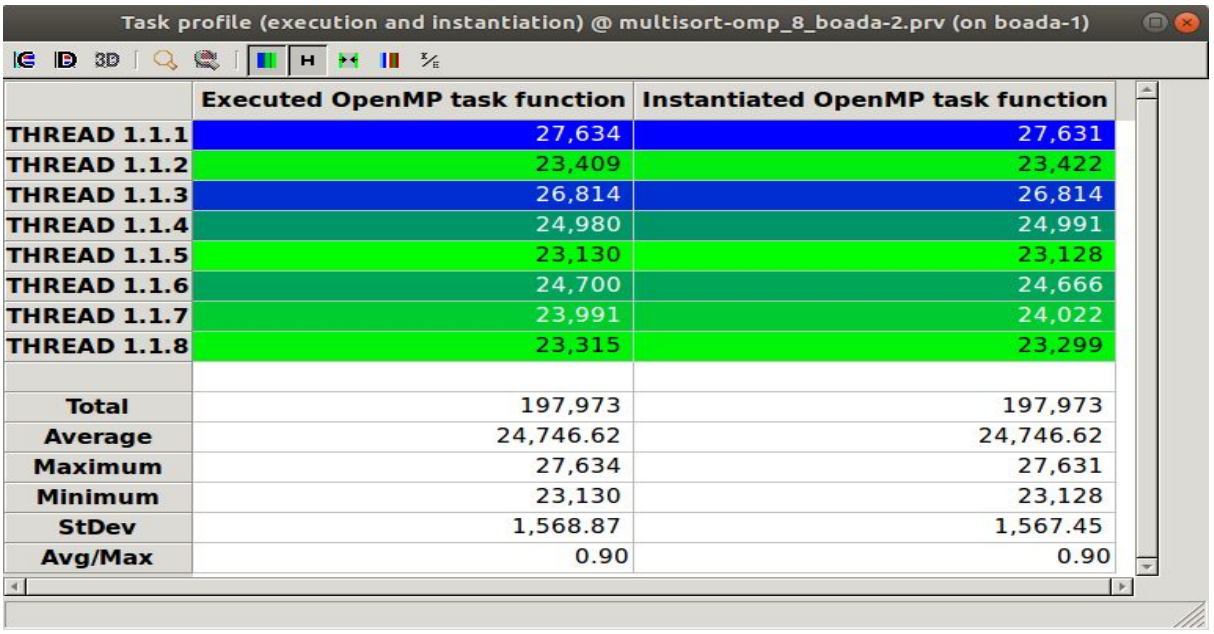


Figure 5.4: Task timeline of version with parallel loops



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	27,634	27,631
THREAD 1.1.2	23,409	23,422
THREAD 1.1.3	26,814	26,814
THREAD 1.1.4	24,980	24,991
THREAD 1.1.5	23,130	23,128
THREAD 1.1.6	24,700	24,666
THREAD 1.1.7	23,991	24,022
THREAD 1.1.8	23,315	23,299
Total	197,973	197,973
Average	24,746.62	24,746.62
Maximum	27,634	27,631
Minimum	23,130	23,128
StDev	1,568.87	1,567.45
Avg/Max	0.90	0.90

Figure 5.5: Task profile of version with parallel loops

The difference that we can see **(Figure 5.5)** respect the others versions of Tree parallelization strategy is the balance of the execution and the instantiation, that are more balanced (the values of Maximum and Minimum are more close to Average), but the total of tasks and the Average are the same.