Parallelism

Divide and Conquer parallelism with OpenMP: Sorting

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Jia Long Ji Qiu: par2212
Jiabo Wang: par2220
Spring 2021-22
12 May 2022

Index

Introduction

In this laboratory assignment, we are going to study the different recursive task decomposition strategies in OpenMP, which will be implemented to parallelize the sorting algorithm Mergersort so we can analyse the results obtained and report the conclusions.

Session 1: Task decomposition analysis for Mergesort

Leaf strategy

In the recursive task decomposition using leaf strategy, it can be observed in the figure 1 that a task is created for each invocation of *basicsort* and *basicmerge*, which is the objective of the leaf strategy. Since the implementation of multisort consists first in multiple recursive invocations before the merge invocations are done, there are certain dependencies between the leaf tasks.

Basicsorts can be independently done as each task works with a different section of the array to sort. However, *basicmerge*s cannot be done until the corresponding *basicsorts* finish as the parts of the array to merge need to be sorted first. Moreover, dependencies between *basicmerges* are provoked as well, since merge is also done recursively and two arrays cannot be merged if each array has not been merged yet.

The granularity of the tasks is controlled by the minimum size variables for both sorting and merging. In the case of *basicsort*, since the minimum size for sorting is 2048, for an array of 32768 elements, 16 invocations of *basicsort* are done (32768/4 = 8192 and 8192/4 = 2048, thus there are four invocations of size 8192 leading each one to four invocations with size 2048, where the basic cases are reached). The same reasoning goes for *basicmerge*, with a total of 64 invocations (32 in the main call, 8 in each recursive call of *multisort*).

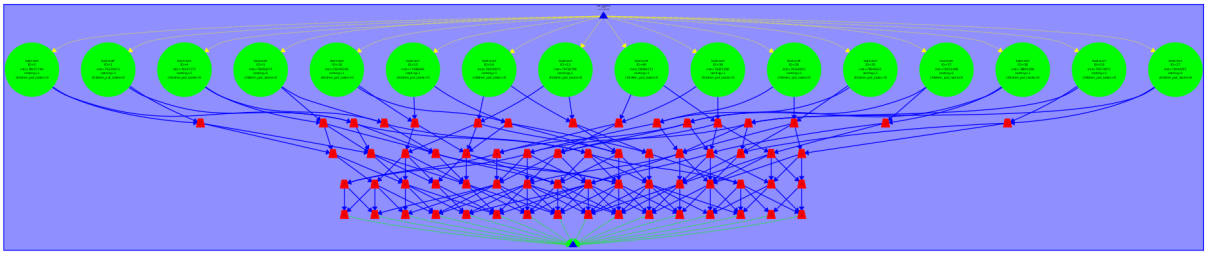

Figure 1: TDG of the program with leaf strategy

In the traces generated with Paraver in figure 2, it can be observed that there is a lot of workload coming from *basicsort* tasks. Dependencies between *basicsort* and *basicmerge* tasks can be clearly visualised in a context of execution time as well; there are regions in black which represent that a certain thread is waiting for synchronisation before proceeding to execute the *basicmerge* task.

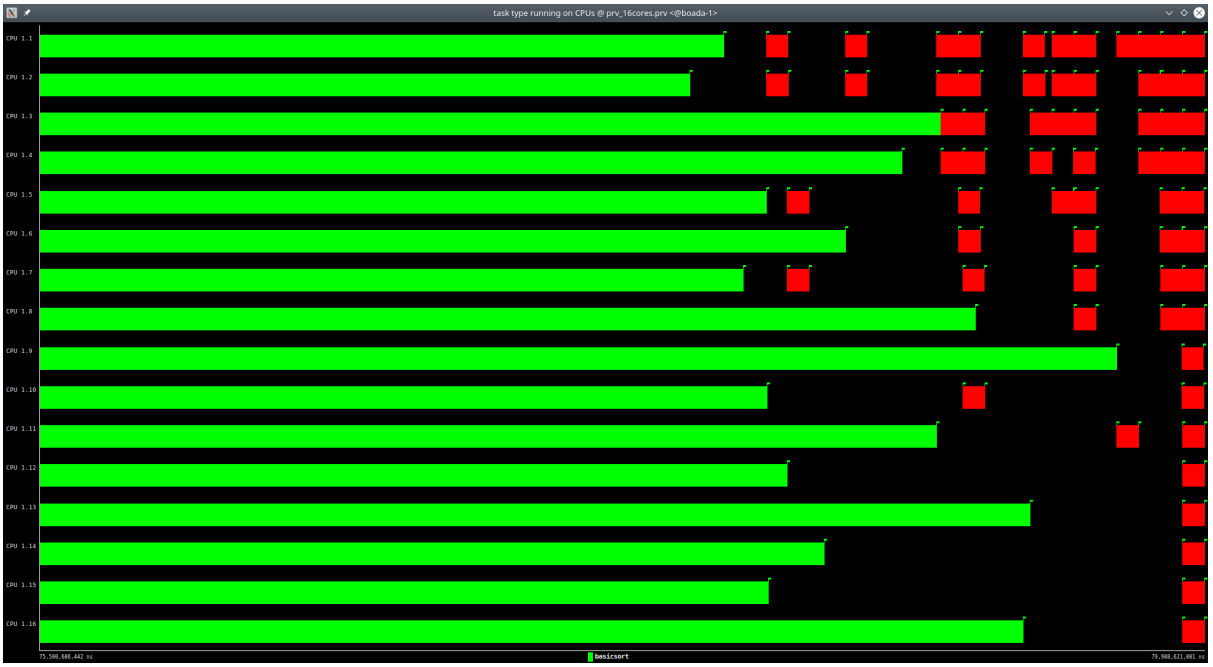

Figure 2: trace timeline of the leaf strategy

Tree strategy

After analysing the leaf strategy, now we are going to do the same with the tree strategy. As we can see in the Task Decomposition Graph of Figure 3, there are tasks inside other tasks, this is because each recursion call is creating new tasks. The four main tasks in the top of the TDG (the green ones) correspond to each of the four multisort calls, they are independent so we can see in the graph that they are arranged horizontally. Inside each of the four tasks, we can see the same distribution of the tasks, because as said before this is a recursion call. Finally, the three tasks in the bottom of the graph (the red ones) correspond to the three merge calls of the function, in this case there are dependencies as can be seen in the graph because every recursive call to merge needs its array parameters to be sorted before.

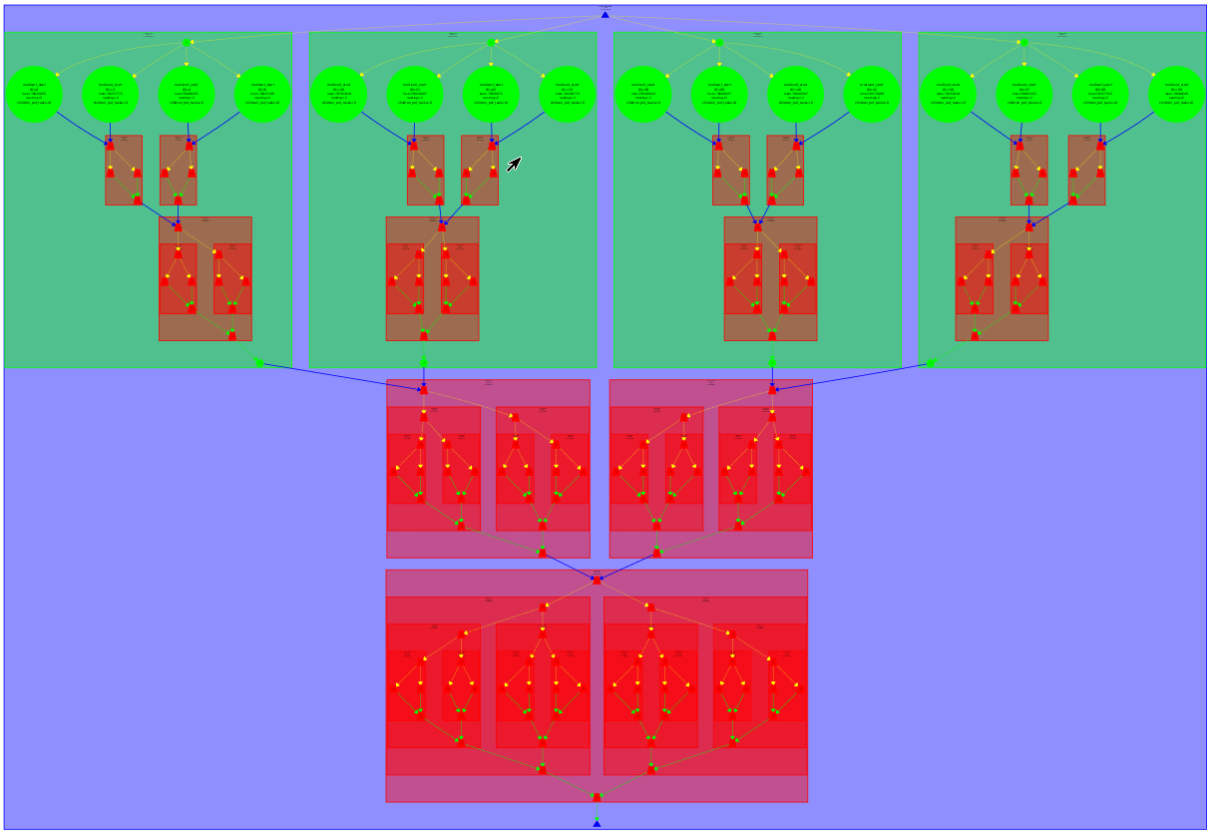

Figure 3: TDG of the program with tree strategy

Now we look at the trace timeline of the tree strategy (Figure 4 and Figure 5). We can observe that the multisort function calls (represented in green colour) are the tasks that last most in the timeline, whereas the red ones represent the merge call functions that have negligible time. Also we can see that the red ones are executed after the green ones, this is because of the dependencies commented before, we have to sort before merge.

Figure 4: trace timeline of the tree strategy

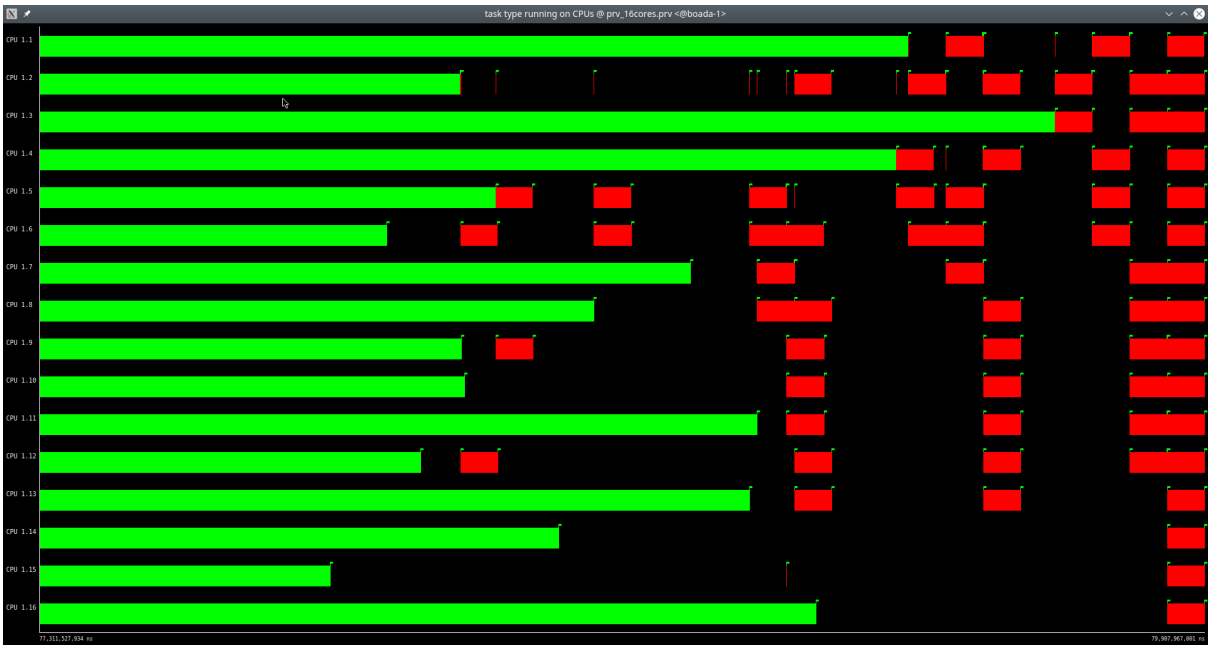

Figure 5: zoomed final part of the trace timeline of the tree strategy

Comparison between leaf and tree strategies

The main difference between those two strategies is in the way tasks are generated. In the leaf strategy, each task corresponds to a simple function invocation, while in the tree strategy each recursive invocation is a task itself, each one leading to more tasks.

There are no major differences between both execution timelines, most of the workload is coming from sorting tasks in both traces, and dependencies are related to merging tasks.

Dependencies are similar in both strategies, the first two merging operations cannot be done before sorting is done, and the last merge cannot be done if the arrays to merge have not been merged yet. To protect these dependencies, a taskwait or taskgroup construct should be used before the first two merge invocations and another before the last one.

Session 2: Parallelisation with OpenMP tasks

As we have analysed in the previous session, there are two recursive task decomposition strategies: the leaf strategy and the tree strategy.

Before starting to comment on the different parallelisation of both strategies, we must explain that the two strategies have the same code in the main function to create the parallelisation, as shown below (Figure 6).

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

Figure 6: Main function of the multisort program

Leaf strategy in OpenMP

The parallelisation with leaf strategy has been implemented as follows:

```
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 7: Merge and multisort functions for leaf strategy

As commented before in the analysis, explicit tasks consist of the invocations of *basicsort* and *basicmerge*. In order to protect the dependencies between the tasks generated, taskwait constructs have been used before the first two merges and the last one. Another option would have been the use of taskgroup for the four multisort invocations and those two merge invocations, as taskgroup implicitly implements a task barrier at the end.

Since the output of several executions of the program were correct, we have assumed that the implementation is also correct.

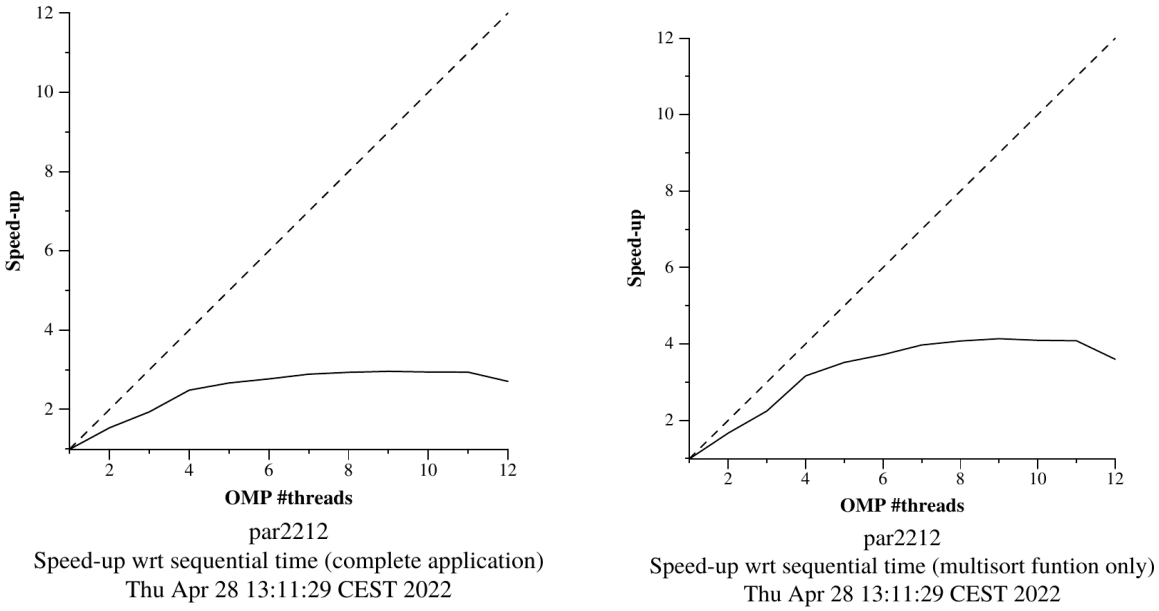

Figure 8: Speed-up al application (left) and speed-up multisort function (right)

The next step was to analyse the scalability of this implementation. As it can be seen in the figure 8, the speed-up is very poor. In order to see what is happening during the execution, we have generated different traces with Paraver.

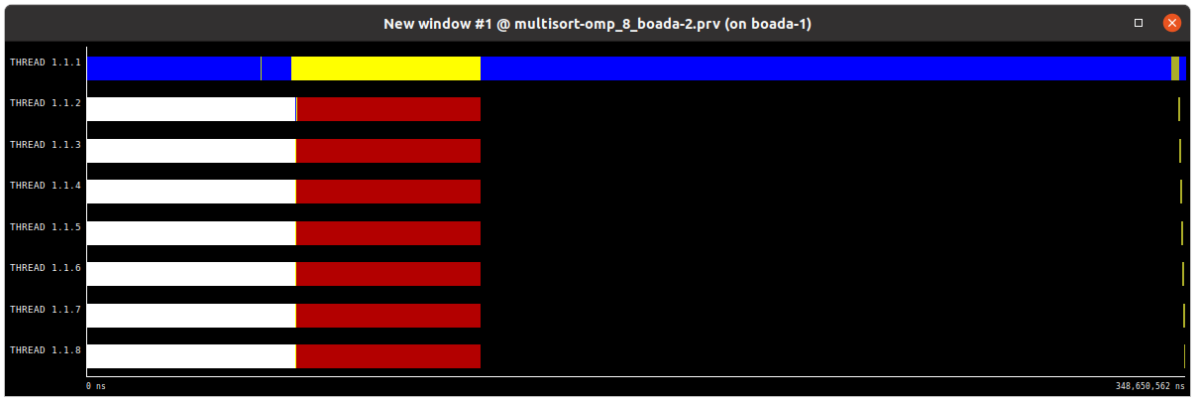

Figure 9: Trace of the Leaf strategy

Analysing first the trace created by default, it can be observed that the parallel region just compounds a small portion of the overall execution, which means that even if the scalability of the multisort function was great, the scalability for the whole application would not be as good, because the sequential part would stay the same.

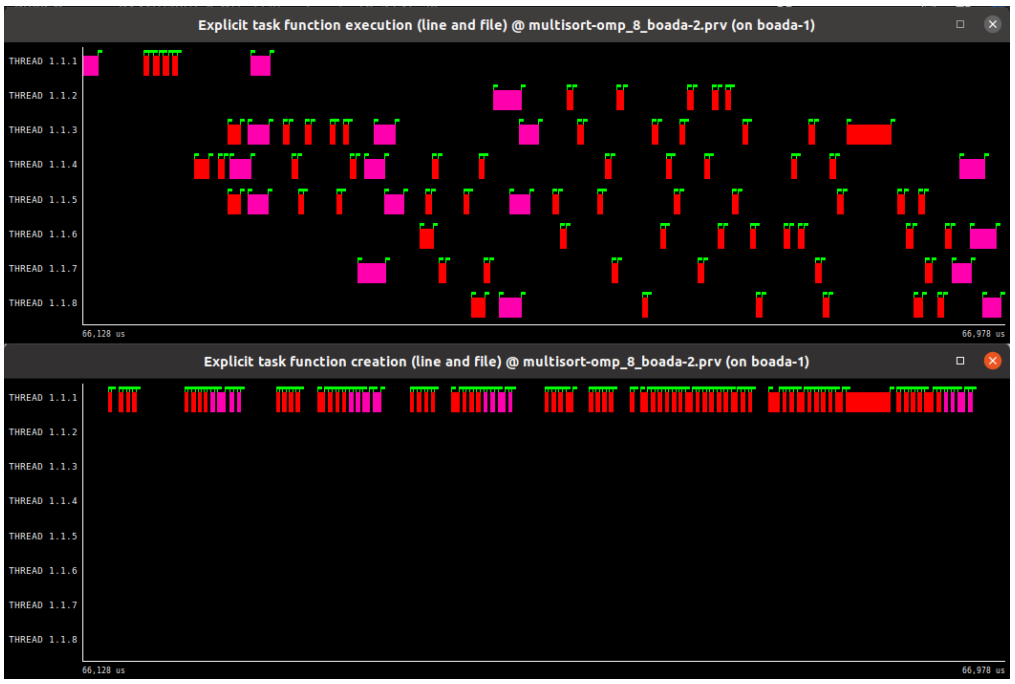

Figure 10: Trace of the task function execution and creation

Now, taking a look at the interesting traces, which are the traces for explicit task function creation and execution, if we zoom into the important parallel regions, it is noticeable that there are not many tasks being executed simultaneously. In fact, the program is not generating enough tasks to simultaneously feed all processors, leading to a big portion of time used for synchronisation.

The reason for this is that since tasks are being generated for the base cases of each recursive function, and there are several sequentialisated recursive invocations through the function itself, a set of tasks (being the tasks those which work with the same part of the array to multisort) cannot be created before the previous recursive invocation finishes. Therefore, sorting or merging functions of different parts of the array are never done simultaneously, so the execution ends up being similar to a sequential execution.

Tree strategy in OpenMP

As we have said before, the tree strategy consists in creating a task for each invocation of a new recursive call.

Because of that, a #pragma omp task construct is put before each recursive call in the multisort and merge functions. But this is not enough, because between the multisort calls and the merge calls we have to put constructs to synchronise the different tasks. As we can see below in the code (Figure 11 and Figure 12), the solution we propose is by putting #pragma omp taskwait, so that the current task waits on the completion of child tasks of the current task to continue its execution.

```
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
    }
}
```

Figure 11: Merge function for tree strategy

```
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
        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);
    }
}
```

Figure 12: Multisort function for tree strategy

In this strategy, we have one thread for each call to the recursive function, which means that we have more tasks and so, more threads can work at the same time. For the same reason, as we can see in the plots below (Figure 13), the tree strategy has a better Speed-Up than the Leaf Strategy, and we can observe that the speed-up of the multisort algorithm is always increasing.

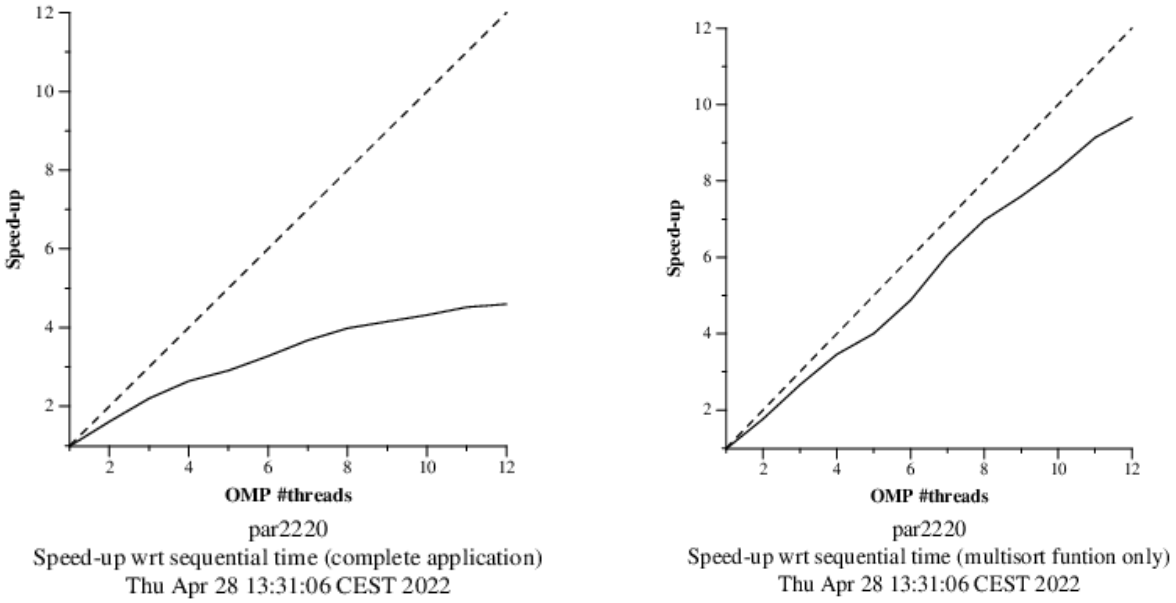

Figure 13: Speed-up al application (left) and speed-up multisort function (right)

As we can observe in the traces below created by Paraver (Figure 14), initially there is only one thread running (thread 1.1.1), but later there are yellow regions, where all the 8 threads are working, and finally again it is only the first thread that is running. This is because we have only parallelised one part of the program, not the whole program. So just in the yellow region, all the 8 threads are running together. It means that there is a big sequential portion in our execution and it just parallelises the multisort and merge functions.

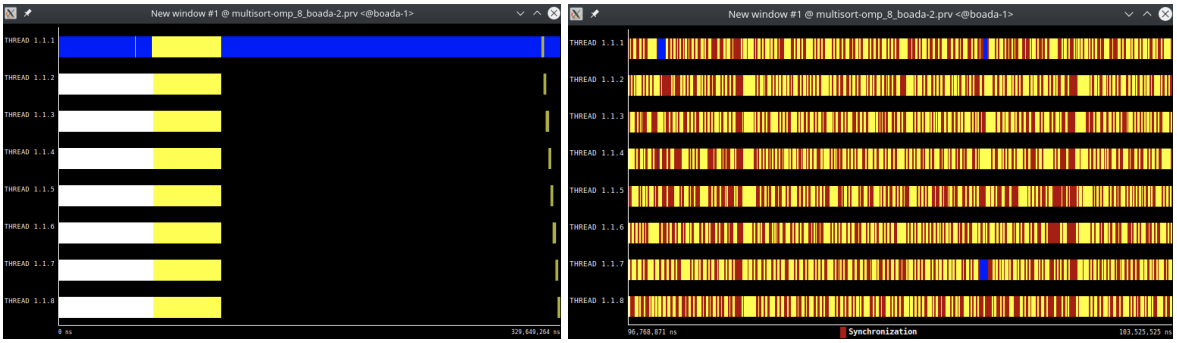

Figure 14: Trace of the Tree strategy (left) and the yellow region zoomed (right)

If we zoom in on the yellow region (right part of Figure 14), we can see that it does tasks such as running (blue region), synchronisation (red region) and scheduling and fork/join (yellow region).

Figure 15: Trace of the task function execution and creation

Finally, observing the traces of explicit task function execution and creation (Figure 15), we can see that all the threads contribute not only in the execution, but also in the creation of the tasks. Although the first tasks are created by the same thread, the execution of these tasks are made by all the threads, and during the execution of the tasks, they create more tasks inside this task, so for this reason, all threads are creating tasks and executing them.

Task granularity control: the cut–off mechanism

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.

The main function remains the same as the normal tree strategy, as shown in Figure 16.

```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp, 0);
```

Figure 16: Main function of the multisort program

We have modified the parallel code implementing the tree strategy to include a cut–off mechanism based on recursion level. We have used the construct *#pragma omp task final (d >= CUTOFF)* and omp in final intrinsic to implement our cut–off mechanism as said in the statement.

Both merge and multisort function are as shown in above (Figure 17 and Figure 18):

```
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);
        }
    }
}
```

Figure 17: Merge function for tree strategy with cut-off

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

Figure 18: Multisort function for tree strategy with cut-off

Once we have implemented the cut-off version for the tree strategy, we have generated the traces for the case in which it is only allowed task generation at the outermost level (-c 0), (Figure 19 and Figure 20) and for level 1 cut–off (-c 1) (Figure 20 and Figure 21).

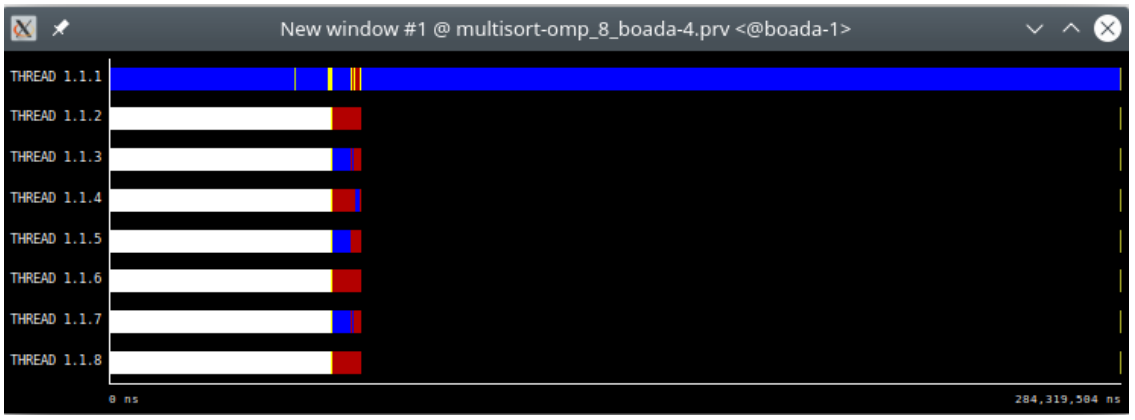

Figure 19: Trace of the new window with cut-off 0

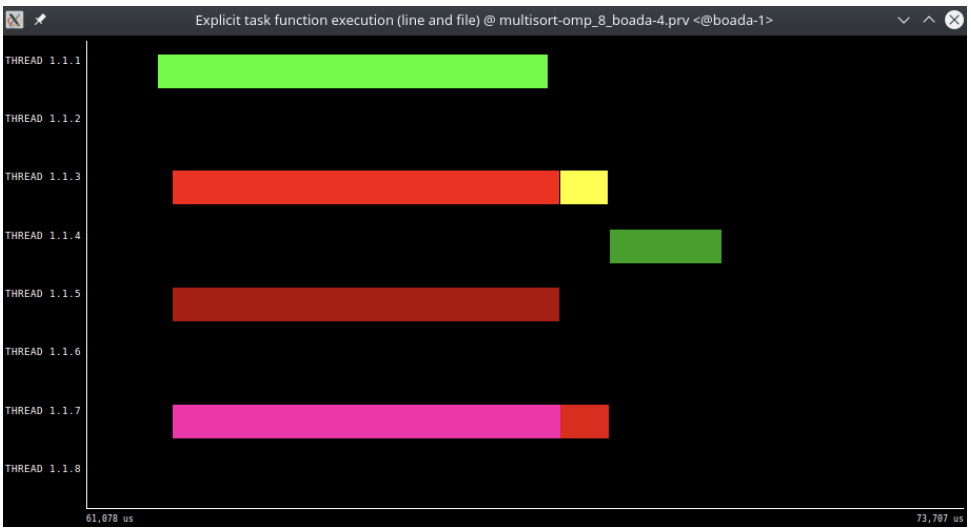

Figure 19: Trace of the task function execution with cut-off 0

Figure 20: Trace of the new window with cut-off 1

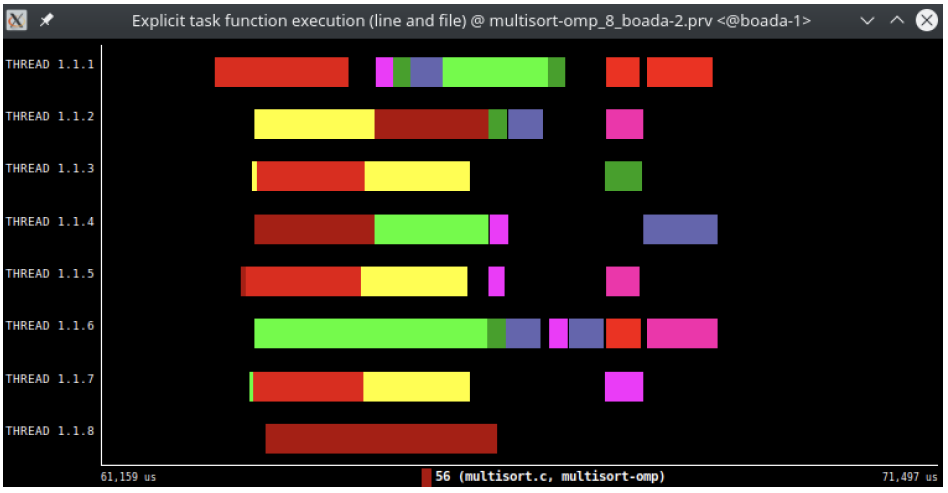

Figure 21: Trace of the task function execution with cut-off 1

The main differences we can observe between both options is that having cut-off 0, we just allow the program to create and execute 7 tasks, the 4 multisort and the 3 merge (stop at the outermost level), whereas having cut-off at level 1, we create and execute more tasks as we can see in both explicit task function execution (Figure 19 and Figure 21).

Once we have analysed the cut-off mechanism, now we are going to explore values for the cut–off level depending on the number of processors used. We have submitted the submit-cutoff-omp.sh script specifying as argument the number of processors to use. We have used 8 processors, and by visualising the graph obtained (Figure 22), we have concluded that using a cut-off between 5 - 7 is when we obtain the best results. So we have decided to use 6 as the optimum value for the cut-off.

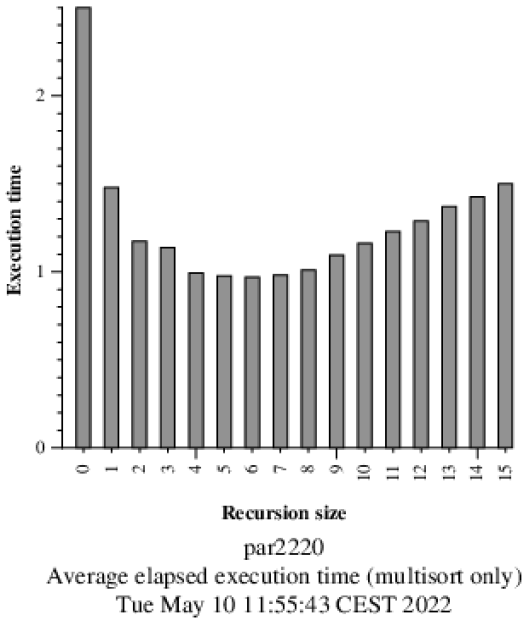

Figure 22: Execution time of the cut-off sizes using 8 processors

Finally, we have obtained the speed-up plots submitting the submit-strong-omp.sh with the values for sort size and merge size set to 128 and for two values for cut-off:

- The original one in the script (the maximum, 16). (Figure 23)
- And the optimum value we have found before in the exploration (6). (Figure 24)

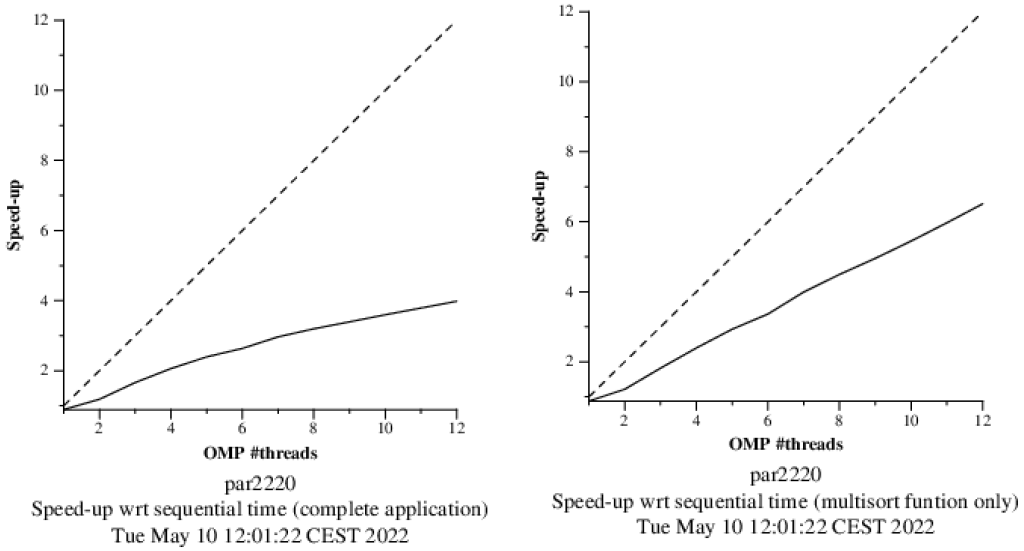

Figure 23: Speed-up al application (left) and speed-up multisort function (right) with cut-off 16

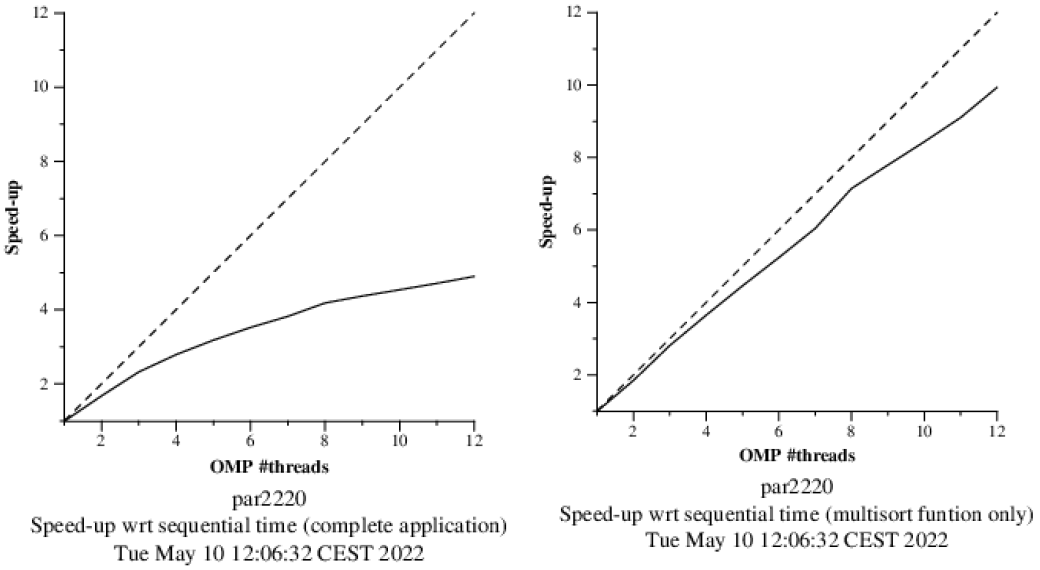

Figure 24: Speed-up al application (left) and speed-up multisort function (right) with cut-off 6

By comparing the two plots, we can see that for the complete application the speed-up just performs a little bit better, but for the multisort we can see a much better speed-up. This is because the excess of tasks generated on the original tree strategy version generates an overhead that is reduced in the cut-off version.

Session 3: Parallelisation and performance analysis with task dependencies

The implementation for the tree parallelisation using task dependencies has been done as follows:

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

            #pragma omp task final (d >= CUTOFF) depend(in: data[0], data[n/4L]) depend(out: tmp[0])
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, d+1);
            #pragma omp task final (d >= CUTOFF) 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, d+1);

            #pragma omp task final (d >= CUTOFF) depend(in: tmp[0], tmp[n/2L])
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);

            #pragma omp taskwait
        }
        else {
            multisort(n/4L, &data[0], &tmp[0], d+1);
            multisort(n/4L, &data[n/4L], &tmp[n/4L], d+1);
            multisort(n/4L, &data[n/2L], &tmp[n/2L], d+1);
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], d+1);
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, d+1);
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, d+1);
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);
        }
    }
    else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 25: Code snippet of the multisort function with dependencies added

In the implementation of multisort, dependency conditions have been added in the task constructs used in the cut-off version. This was done by specifying each array region to sort in a multisort invocation as an output dependency of the task, which will be treated later as input dependencies for the corresponding merge tasks. In the case of the last merge invocation, since it consists in a merge of the two previous merged arrays, tmp[0] and tmp[n/2L] must be protected as well by specifying them first as an output of the two first merge tasks and as an input of the last one. In the case of the recursive invocations in the merge function, no dependencies exist in the same level, so no dependency clauses have been added. For both functions, however, the last taskwait constructs must remain there in order to protect dependencies between different levels.

Again, after multiple executions of the program without throwing any errors about unordered positions, it has been assumed that the implementation is correct.

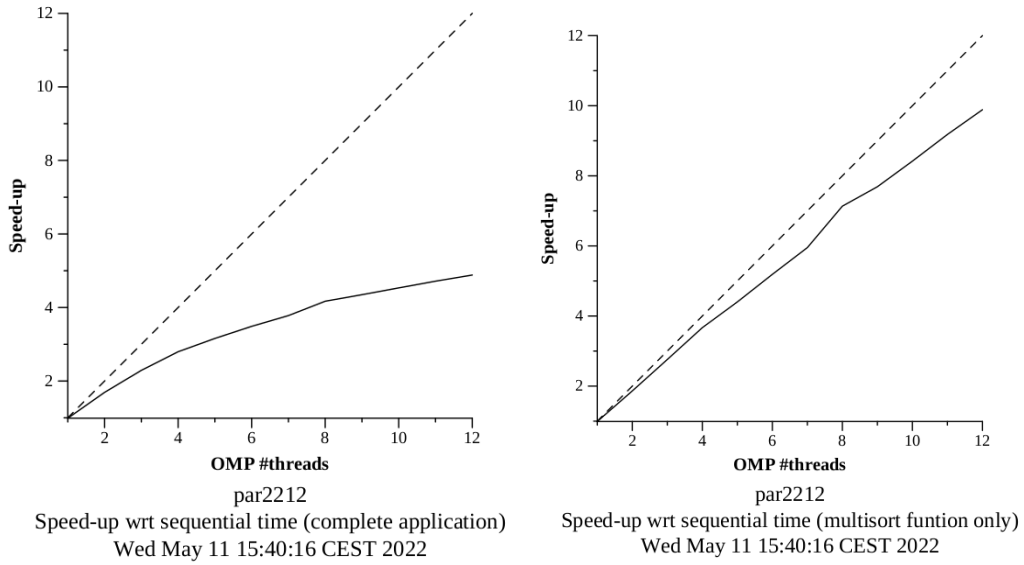

Figure 26: Speed-up al application (left) and speed-up multisort function (right) with cut-off 6

The scalability plots are shown in figure 26, specifying again the optimal parameters used in the with cut-off version.

In terms of performance, compared to the previous tree implementations (without cut-off and the optimal obtained with cut-off), there is not any noticeable difference, since the speed-up is quite similar in all versions.

In terms of programmability, this version was quite more complex to code, since the analysis of dependencies has to take into account the variables used in each task.

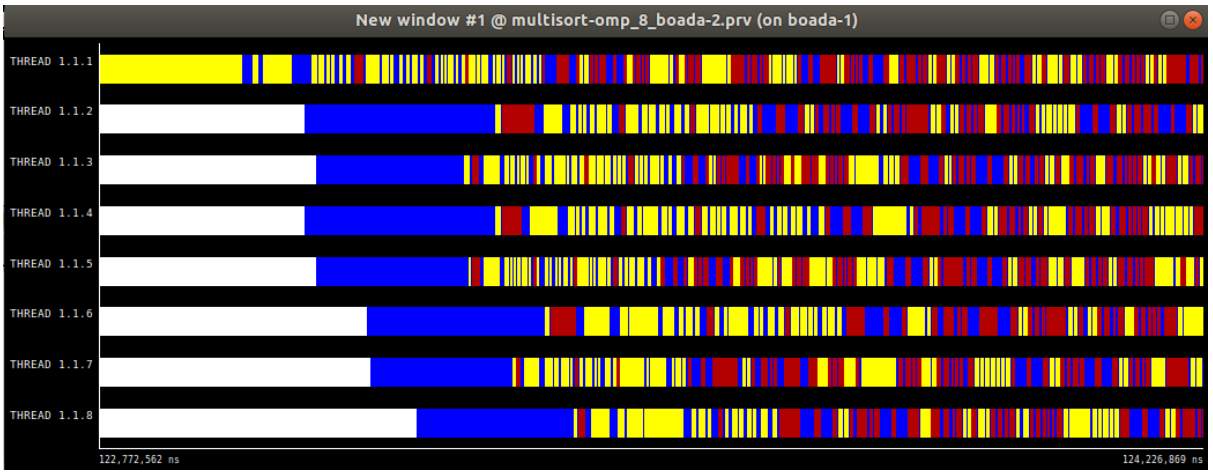

Figure 27: Trace of the new window

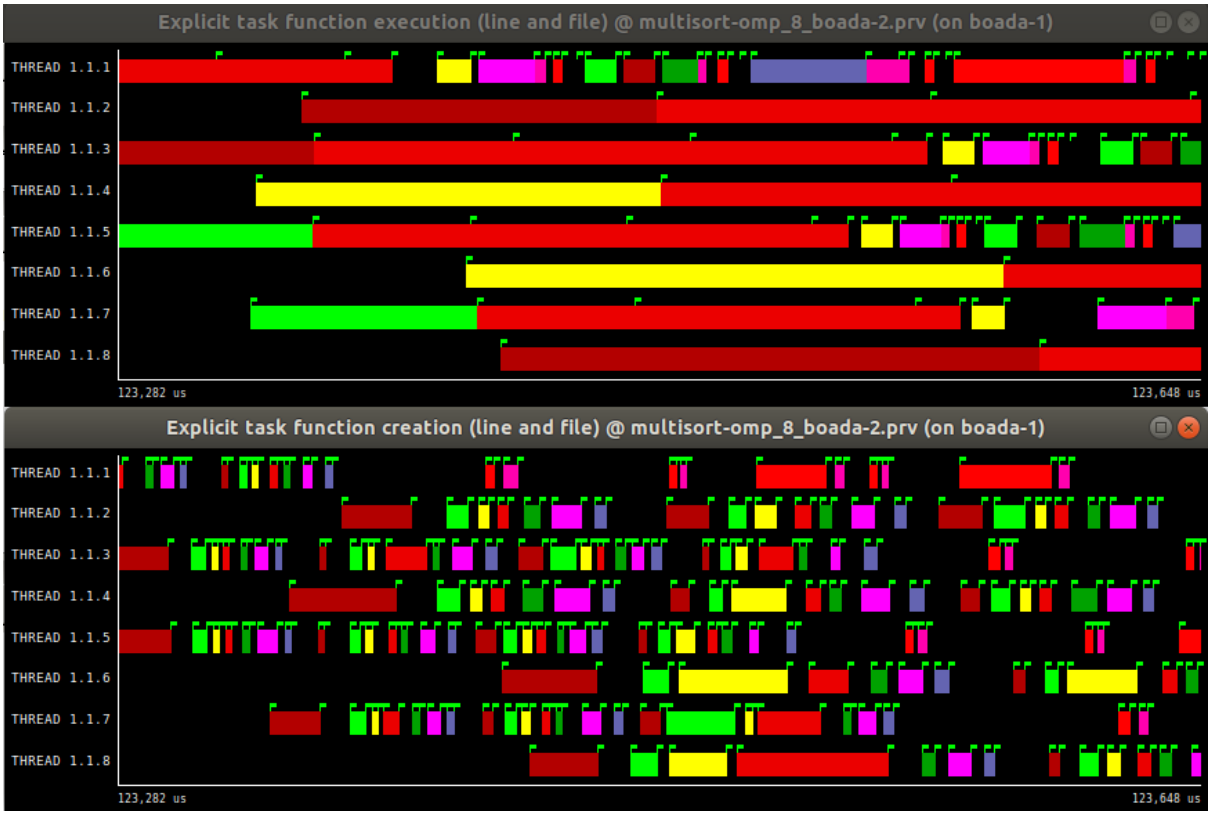

Figure 28: Trace of explicit task function execution and creation

The traces obtained with Paraver shown in figures 27 and 28 do not show major differences compared to the previous tree version either. Again, all threads contribute to the creation and execution of explicit tasks, and there are enough tasks to be executed simultaneously, so a good parallelism is achieved.

Optionals

Optional 1

Now we are going to explore the scalability of our tree implementation with cut-off when using up to 24 threads.

Observing the speed-up plots (Figure 28), we can see that the performance is growing by increasing the number of threads. It is still growing when using more than the 12 physical cores available in boada-1 (by editing np NMAX variable of the submit-strong-omp.sh script to 24) because each boada (what it is a node) has 2 sockets, each socket has 6 physical processors, and each physical processor has 2 threads. So, in total we have 2 * 6 * 2 = 24 threads.

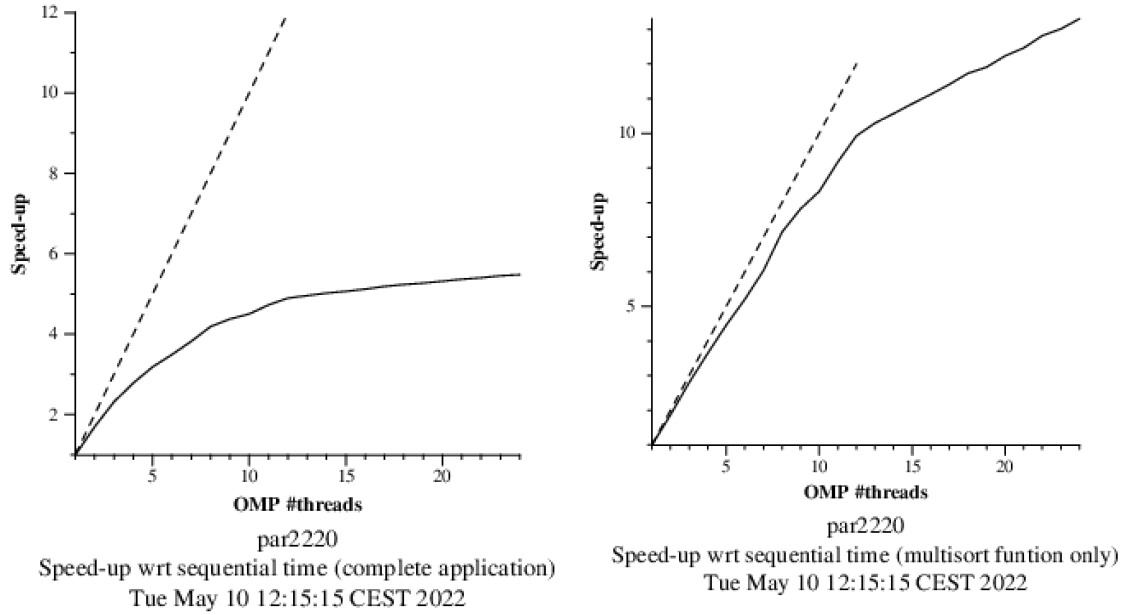

Figure 28: Speed-up al application (left) and speed-up multisort function (right) with cut-off 6 and 24 threads

Optional 2

In this last part we were asked to complete the parallelisation of the multisort program by parallelising the two functions that initialise the data and tmp vectors.

To do so, for the data vector initialisation, implicit tasks were defined as follows:

```
static void initialize(long length, T data[length]) {
    #pragma omp parallel
    {
    long i;

    int n_threads = omp_get_num_threads();
    int id = omp_get_thread_num();

    long block_size = length/n_threads;

    for (i = id * block_size; i < (id + 1) * block_size; i++) {
        if (i==id * block_size) {
            data[i] = rand();
        } else {
            data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
    }
    }
}
```

Figure 29: initialize function

In this implementation, the data vector is being distributed in blocks depending on the number of threads available, so that each thread initialises a block of this vector. To keep the randomness of the original program, the first iteration of each thread assigns at the corresponding position a random number, so that the next positions will be generated randomly too.

In the case of the initialisation for the tmp vector, a taskloop construct with nogroup was used to simply create the explicit tasks, since there is not any dependency between iterations.

```
static void clear(long length, T data[length]) {
    #pragma omp parallel
    #pragma omp single
    {
    long i;

    #pragma omp taskloop nogroup
    for (i = 0; i < length; i++) {
        data[i] = 0;
    }
    }
}
```

Figure 30: clear function

As a result, the scalability for the complete application has improved drastically, as can be seen in the figure 31 .

In the trace generated with Paraver, it can be observed that there is a small region before starting the execution of the multisort function where all threads are already running tasks, which correspond with the explicit and implicit tasks created for the initialisation of data and tmp vectors, as it was expected.

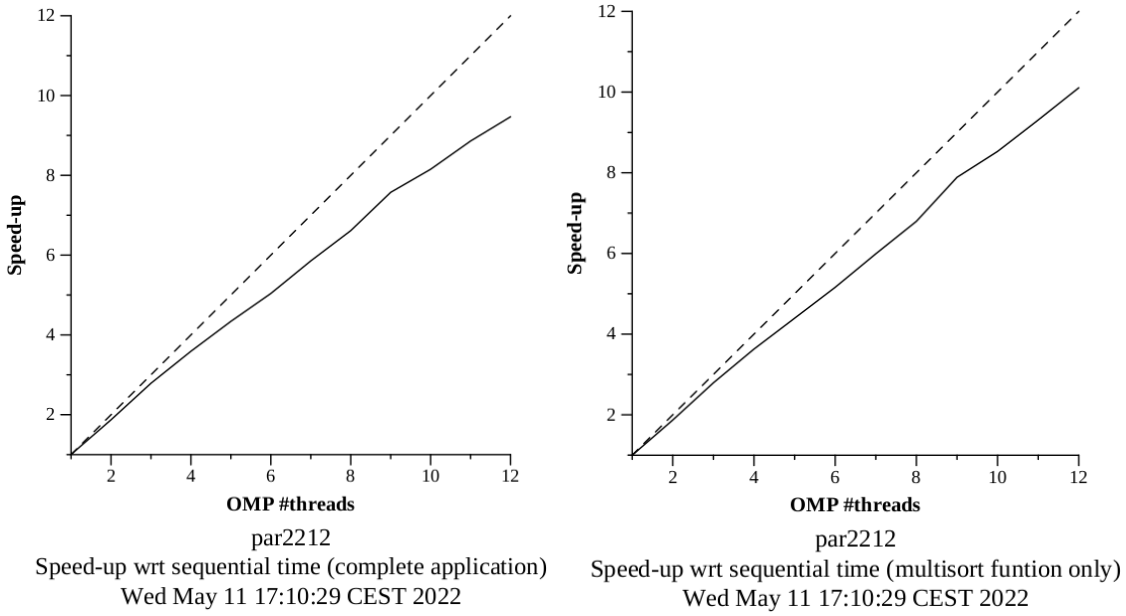

Figure 31: Speed-up al application (left) and speed-up multisort function (right)

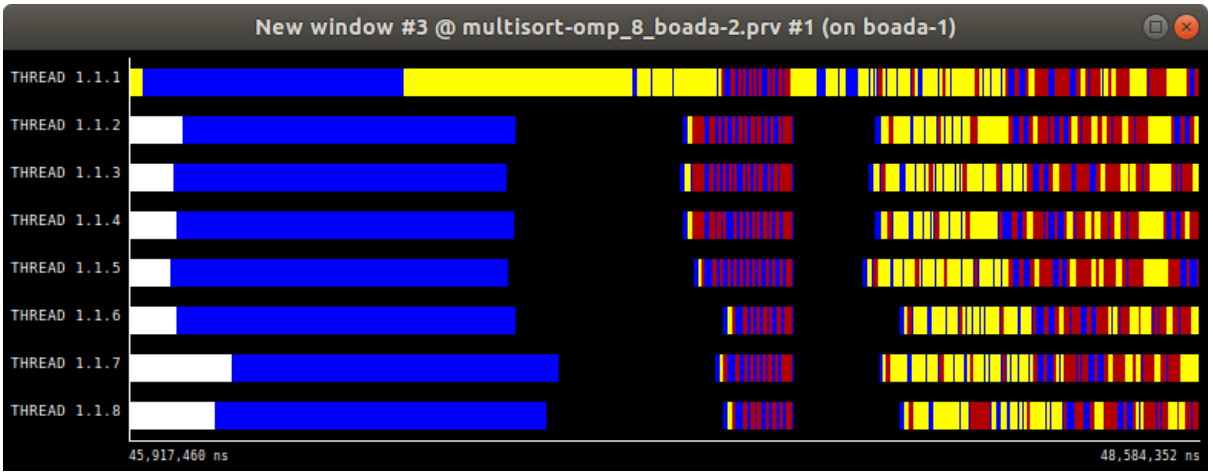

Figure 32: Trace of the new window

Conclusions

Through this assignment we have explored the different ways to achieve parallelism in recursive programs, from a leaf strategy to a tree strategy.

First, we have studied how both strategies, leaf and tree, work. By using the Tareador tool, we have investigated the potential task decomposition of both strategies, their implications in terms of parallelism and the task dependencies generated. Furthermore, by comparing the traces obtained from the Paraver and their Task Decomposition Graph, we have noticed differences between those two strategies.

In the next session, we started implementing both strategies using OpenMP. By putting the necessaries constructs of *#pragma omp task* and *#pragma omp taskwait*, we have managed to implement both strategies, and by analysing the different speed-up plots and the traces generated by paraver we have learned how both strategies perform and their characteristics.

Moreover, in order to control the granularity of the tasks, we have implemented the cut-off mechanism with the tree strategy. First, as always by the help of the traces created by Paraver, we have understood how the cut-off mechanism works. Then, we have found the optimum value for the cut-off and finally we have seen how the performance of the program has improved by adding the cut-off mechanism by looking at the speed-up plots generated.

Finally, in the last session, we have added the tasks dependencies to avoid some taskwait/taskgroup synchronizations. Again, we have analysed how it works by looking at the traces created by Paraver and its performance in comparison to the other version by looking at the strong scalability plots.