Universitat Politècnica de Catalunya

Parallelism

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Carlota Catot Bragós
Ferran Martínez Felipe
PAR1110

Quadrimestre Primavera 2017-2018

Contents

1 Introduction

One of the most interesting structures to deal with in Computer Science is the Tree Structure. This structure, based on a graph without cycles, is used widely in different kind of applications (sorts, searches, ...).

This lab session consists in the study of different strategies related to the parallelization of a Tree Task Structure, where each node is a call of a recursive function (except the leafs). To do that, the two task strategies are based on a thread executing all the tasks until the leafs and then different threads executing different leaf tasks (Leaf Strategy) or threads generating tasks for each node of the tree and different threads executing it (Tree Strategy).

To make the study of the tasks strategies, in the following sections we are going to start with the analysis of the task dependencies using Tareador. After that, we are going to analyze the performance of the two task strategies when we don't have dependencies, to finalize the study by repeating the same analysis but with task dependencies.

2 Analysis with Tareador

In order to parallelize any piece of code starting by trying to parallelize the real code is not the best approximation. In order to study which is the best approach that we should use to parallelize the code, a study about the task dependencies is needed.

In the following subsections we are going to study how we should parallelize the code (basically how to parallelize the merge and multisort functions) and the dependencies generated due to the parallelization.

NOTE: In the following subsections we are going to study different pieces of code, but the pragmas needed to start the parallelization (pragma omp parallel and pragma omp single) are not going to be included

2.1 Merge Function

There are two functions that compound the multisort algorithm: The multisort function itself and the merge function.

For the merge function, the approach selected to study the task dependency was related to create a task for each recursive call to the merge function. With this approach, a task is generated for each of the nodes of the merge recursion tree, and the childs (in this case the merge functions that call basicmerge) are going to be executed for the same task that calls basicmerge.

To do that, a call for creating a new task (the combination of tareador_start_task and tareador_end_task) was placed around each call to the merge function.

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

Figure 1: Merge function with Tareador API calls

2.2 Multisort Function

For the multisort function the same strategy as in the merge function is selected. In order to generate the maximum parallelism (and then study the task dependencies) a task is created for each call to the multisort and merge functions. Comment that in the basicsort function a task creation is not needed, because the multisort task that calls to this function can execute the code of the function.

In tareador, the same code as in the merge function was added to generate the different tasks. We added the combination of tareador_start_task and tareador_end_task around each call to each of the multisort and merge calls.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("First multisort call");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("First multisort call");
        tareador_start_task("Second multisort call");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("Second multisort call");
        tareador_start_task("Third multisort call");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("Third multisort call");
        tareador_start_task("Fourth multisort call");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("Fourth multisort call");

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

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

Figure 2: Multisort function with Tareador API calls

2.3 Task dependency graph

If we observe the following dependency graph we are going to see that the code can be divided in two different sections: the multisort and the merge.

If we observe the dependency graph we can observe that the multisort algorithm is based in four recursive calls to the function with the same name, until the point that the vector reaches a size smaller than the MIN_SORT_SIZE * 4L (at this moment the recursion ends with a basicsort call).

Each multisort function, at the same time, is compound by three merge calls in order to reassemble the different slices of the initial vector into an ordered one. The merge function have two recursive calls to itself, until the vector reaches a size smaller than MIN_SORT_SIZE * 2L. This behaviour can be seen in the dependency graph, in which each merge call have or two childs (in which case we are splitting the vector into two recursive merge calls) or two merge tasks have a common child (the tasks are merging the content of the subvectors).

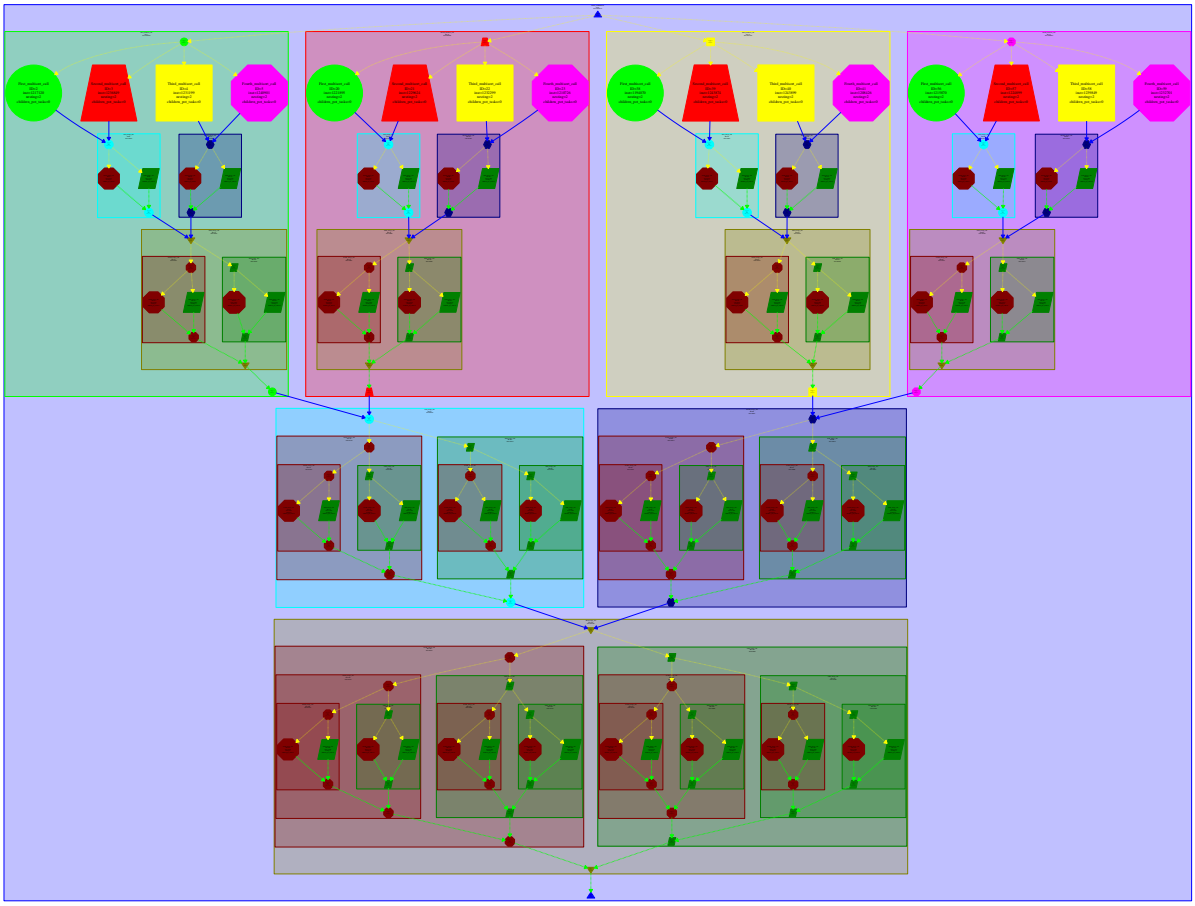

Figure 3: Task decomposition obtained using Tareador

So, related to parallelize the code, we can observe that none of the childs are dependant between them, so we can use a Leaf strategy (that consists in generate a task for each child). In addition, we can observe that we can use a Tree strategy too (generate a task for each call of the recursive functions).

Finally, we have to take into consideration that we must create a synchronization between the multisort tasks and the merge ones, because a merge task depends of different multisort task, and without a synchronization we are going to have wrong results due to data races.

2.4 Execution Time and Speed-Up for different #Threads

In the following table we can see the different Execution Times (in microseconds) and Speed-Ups for different number of threads:

Table 1: Speed-Up and Execution Time (in microseconds) for different # Threads

Threads	**Execution Time**	**Speed-Up**
1	20.334.421	1
2	10.173.720	1.999
4	5.086.725	3.997
8	2.550.595	7.97
16	1.289.909	15.76
32	1.289.909	15.76
64	1.289.909	15.76

As we can observe from 1 to 16 threads the Speed-Ups are pretty close to the ideal ones. However, from 16 threads in advance we can see that no improvements are obtained by adding more threads. This behaviour can be observed in the dependency graph too, in which we can see that at the moment that more processors are needed the number of tasks doing work is 16.

Anyway, this Speed-Up improvements are going to be studied in the following sections, in which we are going to observe that this ideal parallelism is far from the real one.

3 Parallelization and performance analysis with tasks

As we explained in the previous section, at the moment of parallelize the real code we can take two different strategies: a *Leaf Strategy*, which consists in create a task for each leaf in the dependency graph (with an additional task doing all the recursion and generating all the tasks) and a *Tree Strategy*, which consists in creating a task for each node in the tree.

Before starting to comment the different strategies, we must explain that the two strategies have the same code in the main section, that can be found under this text:

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

Figure 4: Code in the main function needed to generate tasks in Tree and Leaf strategies

3.1 Tree Strategy

As we commented in the previous sections, this strategy consists in creating a task for each call inside a new level of recursivity, generating parallelism and improving the speed-up of the program.

To do that, a #pragma omp task is placed before each recursive call in the multisort and merge functions. However, and as commented in the previous section, between the multisort calls and the merge calls is needed a mechanism to synchronize the different tasks, that is done by placing a taskgroup for the multisort calls and another taskgroup for the two first merge calls (the final merge call is created alone, so it 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);
    }
}
```

Figure 5: 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 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 6: Multisort function for Tree Strategy

In the tree strategy we have one thread for each call to a recursive function, which means that we are going to have more tasks, which derive in that more threads can work at the same time. This strategy provokes then a better Speed-Up than the Leaf Strategy (which is going to be explained in the next subsection) which can be observed in the following plotes, in which the speed-up of the multisort algorithm is always increased. This behaviour can be seen in the below plots:

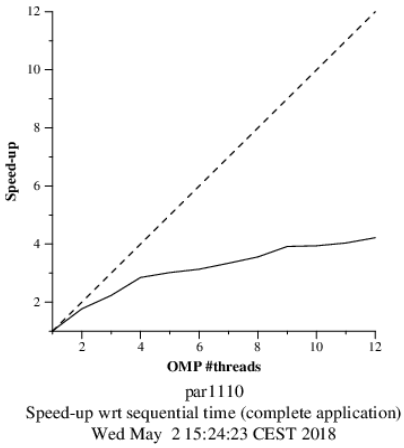

Figure 7: Speed-Up all application

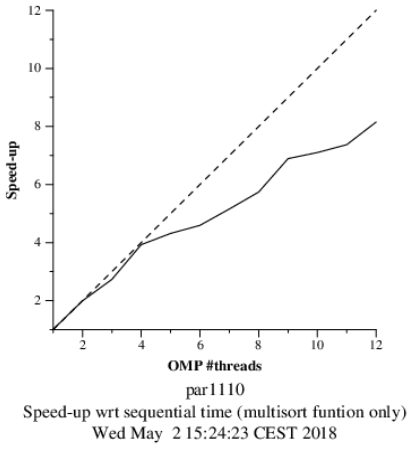

Figure 8: Speed-Up multisort function

If we take a look to the paraver traces we can observe that in the sections that the threads are working (the ones which are not white) four colors can be seen.

The yellow one corresponds to the task of fork/join different tasks. The cyan corresponds to a thread that is Idle. The blue one corresponds to a thread that is doing work. The read corresponds to synchronization between tasks.

In this case we can observe that each thread have workload generating and synchronizing different tasks. In addition, and in spite of the threads with big synchronization regions, we can observe that there are not threads in Idle state (or that the threads are nearly never in the Idle state) which means that this strategy has a better parallelism that the Leaf strategy (shown in the next section).

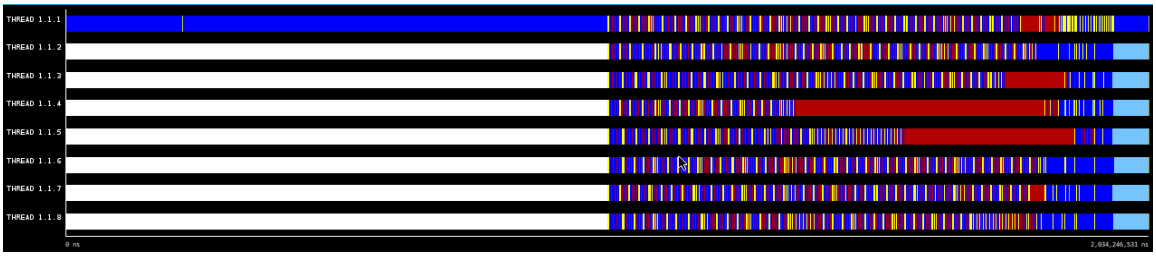

Figure 9: Thread State for Tree Strategy

3.2 Leaf Strategy

As we explained in the previous sections, this strategy consists in creating a task for each leaf in the tree generated by making the recursive calls.

To do that, a #pragma omp task is placed before each terminating function (that is any function inside merge and multisort that doesn't have recursion, basically basicmerge and basicsort). We have the same problem than in the previous section, so a taskwaits are needed to synchronize the different tasks in order to avoid data races.

```
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);
    }
}
```

Figure 10: Merge function for Leaf Strategy

```
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 11: Multisort function for Leaf Strategy

In the leaf strategy we have one thread generating all the tasks, and then a thread executing one task for each leaf. That means that we cannot give work to more threads that leafs we have. This strategy derives in a lot of threads without work to do, and this provoke a horrible speed-up (we can see that we reach the maximum speed-up with 5 threads, that correspond to 4 threads doing work and one generating tasks). This behaviour can be seen in the below plots:

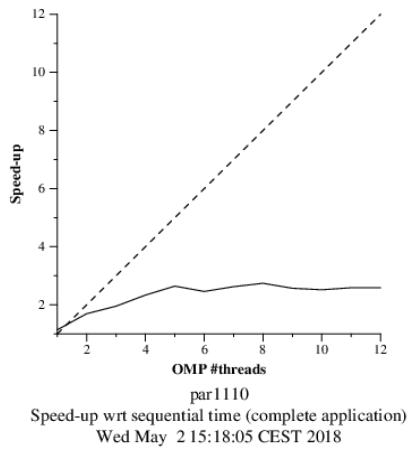

Figure 12: Speed-Up all application

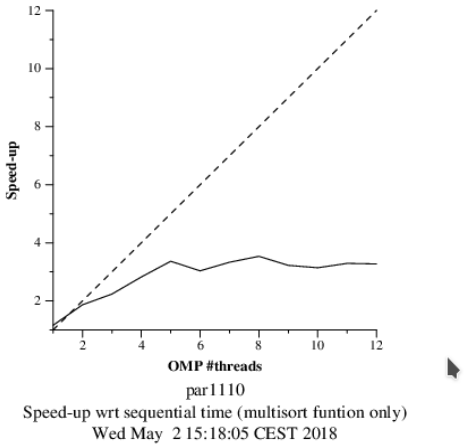

Figure 13: Speed-Up multisort function

If we take a look to the paraver traces we can observe that in the sections that the threads are working (the ones which are not white) three colors can be seen.

The yellow one corresponds to the task of fork/join the different tasks. The cyan corresponds to a thread that is Idle. The blue one corresponds to a thread that is doing work.

For the sections of work we can observe that there is a job of generation and synchronization of tasks and a job of doing work. For the processors that are doing work, we can observe that the sections of the processors doing nothing are nearly equal than the sections of the processors doing work. That means that increasing the number of threads will produce an increase of the Idle sections (due to the distributions of the work) producing an irrelevant (if exists) Speed-Up.

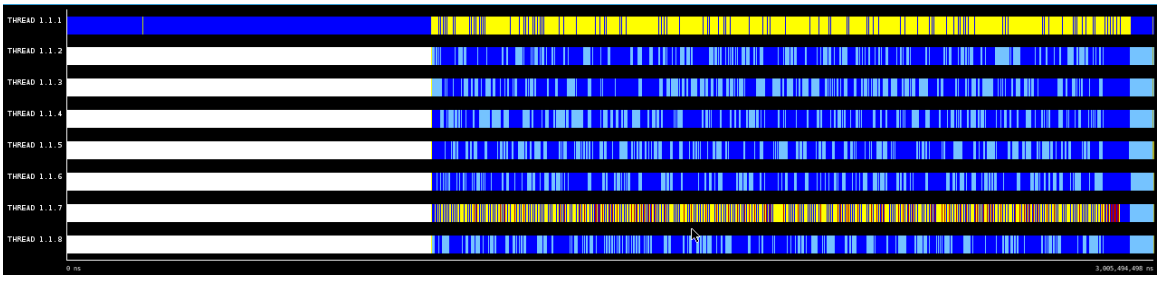

Figure 14: Thread State for Leaf Strategy

4 Parallelization and performance analysis with dependent tasks

In the previous section, we generate a code with Tree Strategy paralelization, now we will regenerate the code in order to enforce task dependence. For doing this, we will use dependencies point by point with the pragma: **#pragma omp task depend(in: ...)** for setting the values that the next operation depends on, or **#pragma omp task depend(out: ...)** for setting the values that other operations depends on, and also that the next operation will modify it, so the operation must finish after an operation with the specific dependency in *"in"* value, starts.

We can see the multisort function with this strategy in the code bellow (for seeing all the code, we attach *multisort-omp-dependency.c*):

```
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 15: Multisort code with dependence point by point strategy

In the *point by point dependence strategy*, we have the same threads than in Tree Strategy, so if we compare the speed-up plots (figures 25 and figure 26) with the ones of Tree Strategy (figures 7 and figure 8) we can see that there is no significant relevance between the plots. We think that we should obtain better results on this strategy because the tasks only have to wait for the specific values, and this should be fastest than taskgroup, that suspends the tasks until all the tasks on them have finished.

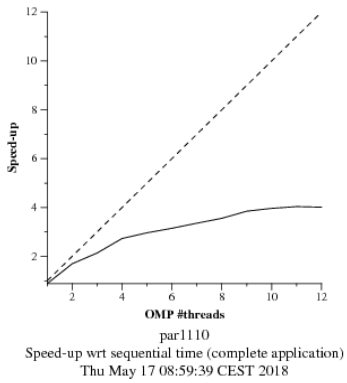

Figure 16: Speed-Up all application

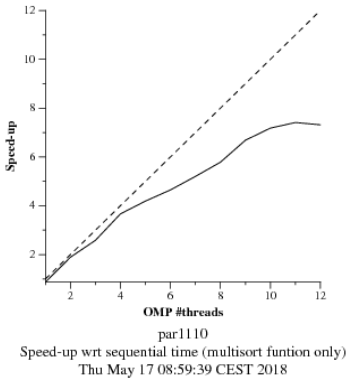

Figure 17: Speed-Up multisort function

In the paraver traces bellow, we can observe that one more time there is such similarity between the Tree Strategy one (figure 9), so for the paraver trace is applicable the same we said before.

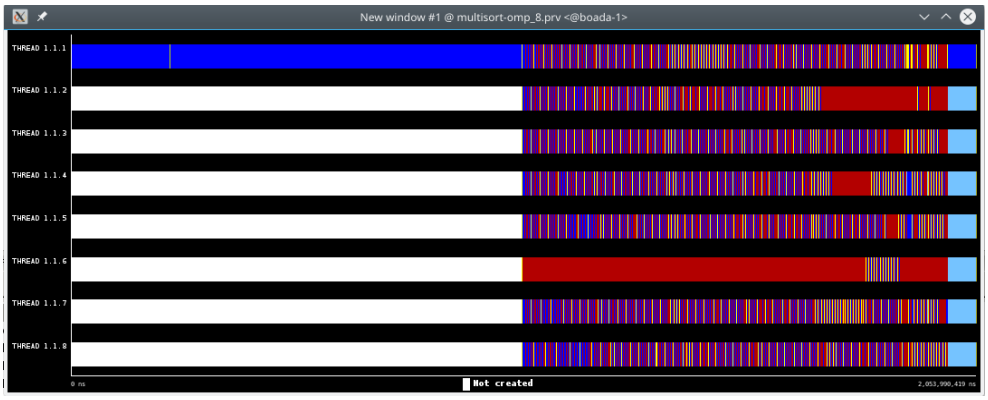

Figure 18: Thread State for Point by Point Dependency Strategy

5 Optional

5.1 Optional 1

In this subsection we will complete the parallelization tree by parallelizing the two functions that initialize the data and the tmp vectors. This is done it in the code bellow by inserting a #pragma omp parallel for after the data enty on the vectors.

```
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 19: Parallelization of the initialisation of tmp and data vectors

After making the code more parallel we can see in the Speed-Up plots that the times are better than in the Tree Strategy, that's because all global Speed-Up is also increased by making the code more parallel with the first functions the code calls.

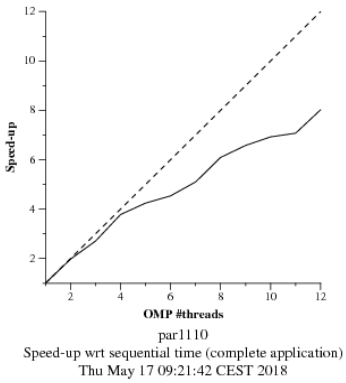

Figure 20: Speed-Up all application

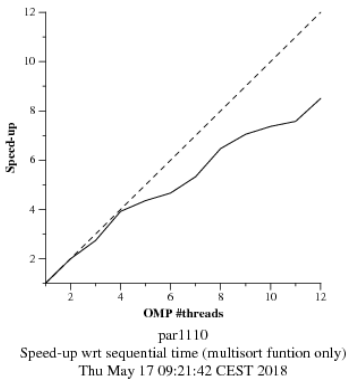

Figure 21: Speed-Up multisort function

In the paraver trace we can see that in comparison of the Tree Strategy there is lees white part, so there is less time that is working only one thread. This also convince us that this code is better, because we can stay less at the start of the code, with the initialisation.

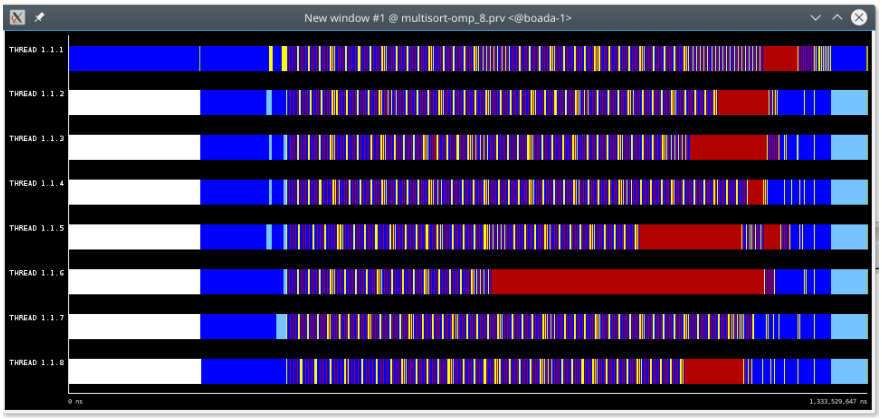

Figure 22: Thread State for Point by Point Dependency Strategy

5.2 Optional 2

In this subsection we will explore the best possible values for the sort_size and the merge_size arguments used in the execution of the program. For making a better choice of the values, we run the script 3 times for each value, to make sure that the value is more significant.

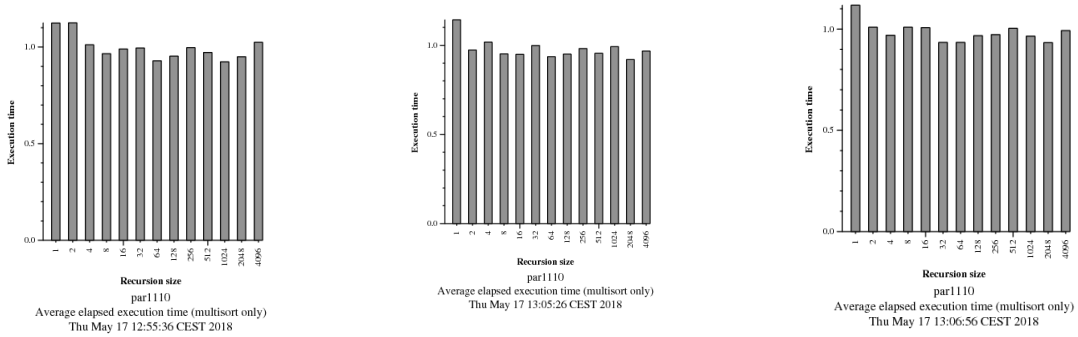

Figure 23: sort_size value analyse

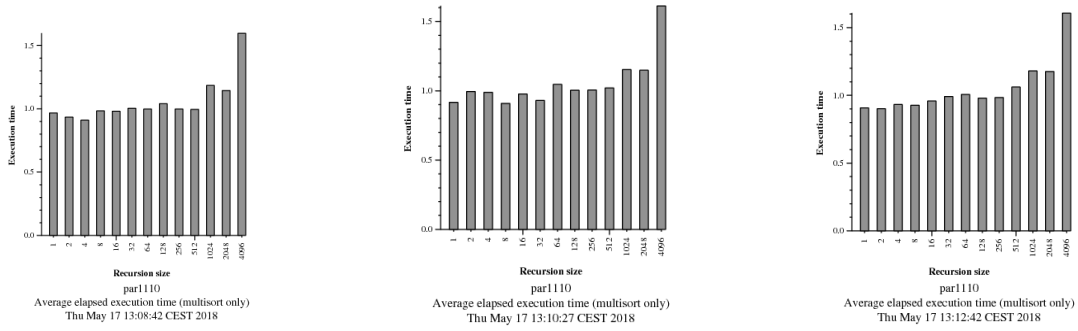

Figure 24: merge_size value analyse

With the plots of figures 23 and 24 we conclude that the best values for variables are: sort_size = 4 and merge_size = 64. With this values we extract the following Speed-Up plots by modifying the script *submit-strong-omp.sh* with the selected values.

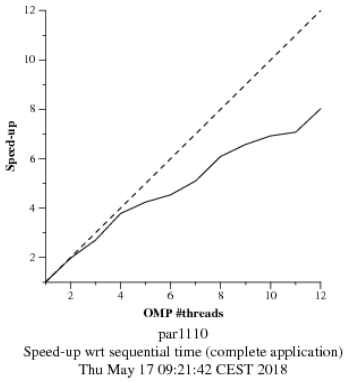

Figure 25: Speed-Up all application

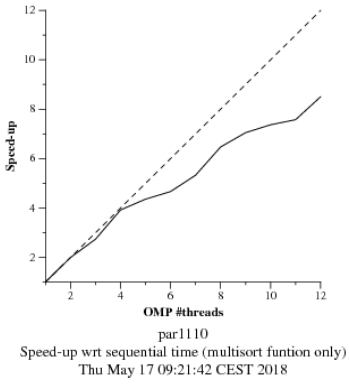

Figure 26: Speed-Up multisort function

In this plots we can see an improvement on the Speed-Up for multisort part. This is generated by getting an optimum value for the significants variables merge and sort.

6 Conclusions

The main goal of this lab session has been study the different strategies for parallelize the multisort function.

In first instance, for analizing the tareador we start by adding some tasks in merge function to see the recursion tree, then we do the same for the multisort function, adding new tasks manually for each recursive function in the code, with this we analise the tasks dependency graph and we observe that there is such recursive and it can be more parallel.

By making the different parallelizations we start with the Leaf and Tree strategy, about this, we can say that the principal thing to conclude is that with a high number of processors, the Tree Strategy is always better than the Leaf one, if we don't have overheads or many dependencies.

The second parallelization practise was by setting the dependencies point by point, here we can see (comparing with the previous ones) that if we have a few dependencies in a large code, is always better to make it point by point because in these way, the tasks will not be suspended until all the task finishes, it will be suspend until the variables the next tasks needs, have been correctly modified.

In the end, in the optional part, we see that if the function is very parallel, it's only a fraction of code, and if the rest of the code is sequential, the general parallelization is unpredictable so, we need to make the rest of the code more parallel for having more parallelization in all the code.