PAR Laboratory Deliverable

Course 2019/20 (Fall semester)

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Rafel-Albert Bros Esqueu, Joan Vinyals Ylla-Català

User: Par2201

October, 2019

Contents

1 Introduction

Divide et impera **(Philip II of Macedon)**.

There are only a few sayings that have an application in so many different aspects of life such as *Divide and conquer*. Known since ancient times, it is a concept that has influenced politics, military tactics and business all around the world throghout history.

This project, of couse, is not about any of the typical situations previously exposed. It is, in fact, about an aspect of life that no one that took this saying into consideraton before the XX century would have ever thought about.

Following our path into fully understanding parallelization, we will study and analyze two different strategies to work with a sort algorithm based on the design paradigm of *divide and conquer*: the **leaf** strategy and the **tree** strategy.

2 Analysis with Tareador

So far, the typical approach when asked to parallelize a random piece of code was to directly do it. However, this is not the smartest strategy.

Throughout the following sections we will look into our initial modification of the code of the program `multisort-tareador.c` in order to study the data dependencies the program has, being that the best and smartest process to decide which is the correct approach to parallelize the code afterwards.

2.1 Modification of `multisort-tareador.c`

The code of `multisort-tareador.c` (included in the deliverable package) is based on two main different functions among others: the *multisort* function and the *merge* function.

2.1.1 Merge function

Regarding the *merge* function we identified three main calls which can be defined as tasks. Those are the *basicmerge* and the recursive ones. In order to analyze it with *Tareador* we used its own API as you can see in the code shown in *Figure 1*. This will induce in the task dependency graph (TDG) that each *merge* task will have two child except for those found in the last level of recursion.

```
1  if (length < MIN_MERGE_SIZE*2L) {
2    // Base case
3    tareador_start_task("basic-merge");
4    basicmerge(n, left, right, result, start, length);
5    tareador_end_task("basic-merge");
6  } else {
7    // Recursive decomposition
8    tareador_start_task("merge.in.1");
9    merge(n, left, right, result, start, length/2);
10   tareador_end_task("merge.in.1");
11   tareador_start_task("merge.in.2");
12   merge(n, left, right, result, start + length/2, length/2);
13   tareador_end_task("merge.in.2");
14 }
```

Figure 1: Body of the *merge* function from `multisort-tareador.c`.

2.1.2 Multisort function

In this case, we detected four recursive calls of the function *multisort* that order a quarter of the vector part independently and then are merged together in pairs until the whole vector part has been ordered. As well as for the *merge* function, we have used the *Tareador* API as can be seen in *Figure 2*.

```
if (n >= MIN_SORT_SIZE*4L) {
  // Recursive decomposition
  tareador_start_task("multisort-1");
  multisort(n/4L, &data[0], &tmp[0]);
  tareador_end_task("multisort-1");

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

  tareador_start_task("multisort-3");
  multisort(n/4L, &data[n/2L], &tmp[n/2L]);
  tareador_end_task("multisort-3");

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

  tareador_start_task("merge.1");
  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
  tareador_end_task("merge.1");

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

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

Figure 2: Body of the *multisort* function from `multisort-tareador.c`.

2.1.3 Task dependency graph

Generating a task dependency graph is a very useful way to understand the way a particular code behaves. As revealed by the graph seen in *Figure 3* generated by *Tareador* there are two obvious different structures in the program we have been working so far.

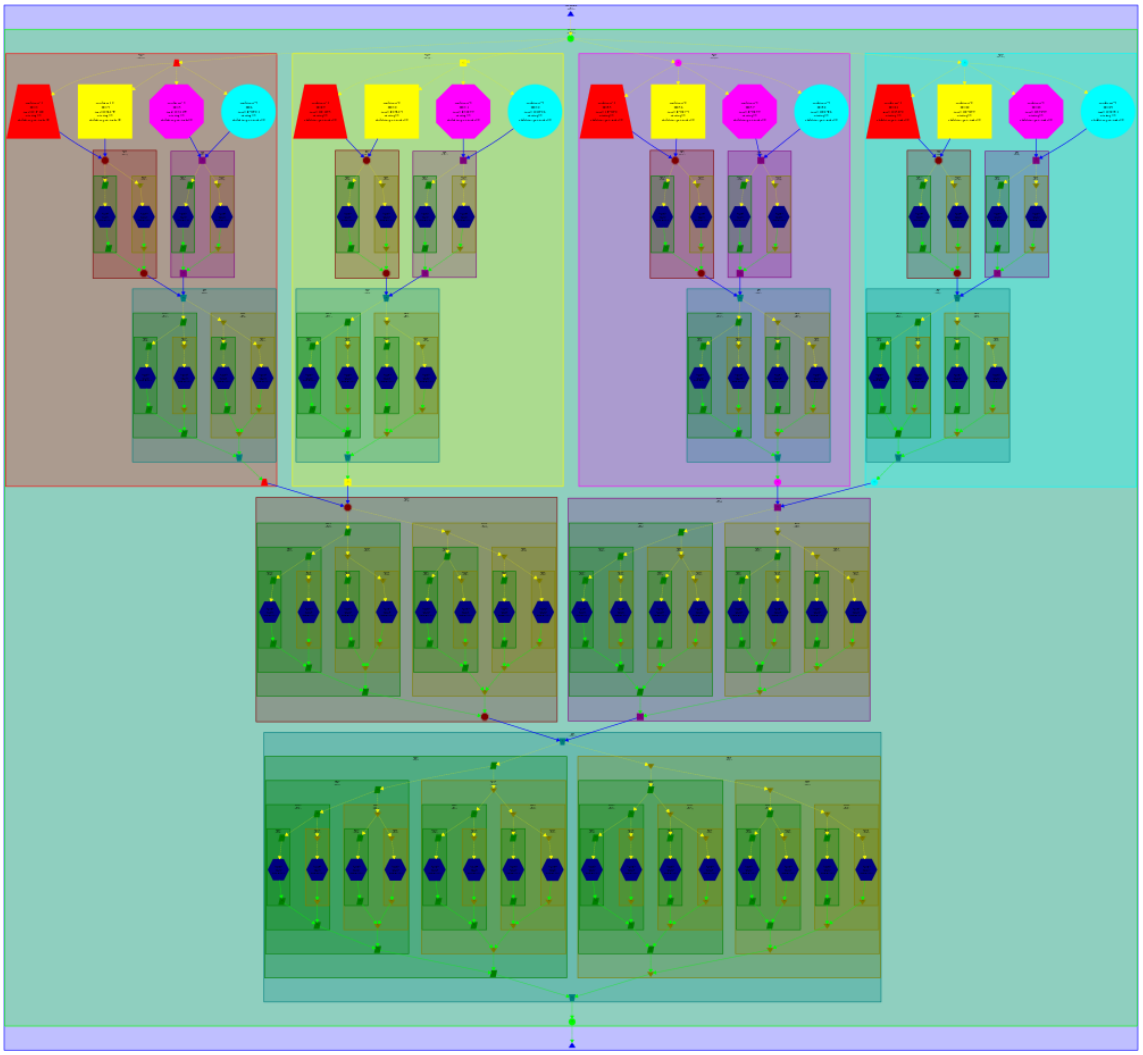

Figure 3: Task dependency graph generated with Tareador.

From one hand we observe that the multisort algorithm works basically using four recursive calls to the homonym function, as long as the vector has a size bigger or equal to the `MIN_SORT_SIZE*4L`.

On the other hand, we have the merge functions, which have two recursive calls to itself until the size of the vector is smaller than `MIN_SORT_SIZE*2L`.

2.2 Execution times and speed-ups for different amount of threads

As observed in *Table 1*, the simulation of `multisort-tareador.c` run by *Tareador* shows that the Speed-ups are close to the ideal case as far as 16 threads being used. Afterwards, there is apparently no improvement when adding additional threads to the simulation.

Processors	Execution time	Speed-up
1	20.3344	1
2	10.1736	1.99875
4	5.08605	3.99808
8	2.54898	7.97748
16	1.28445	15.8312
32	1.28216	15.8594
64	1.27471	15.9522

Table 1: Execution times (in seconds) and speed-ups predicted by Tareador for different number of threads.

We are able to justify this behaviour seen in the table above thanks to the task dependency graph analyzed in the previous section, as the biggest amount of tasks executed in parallel is 16, which is the maximum amount of processors that imply an improvement in the execution of the program.

3 Parallelization and performance analysis with *OpenMP* tasks

Following the task decomposition analysis conducted throughout the previous sections we are now going to explore two different possible ways to parallelize the given code: the *Tree* version and the *Leaf* version.

3.1 Implementation of *Leaf* and *Tree* versions

For the *Leaf* version we work on a strategy based on creating a task per every call in the recursive tree, while for the *Tree* version we create a new task for each call that implies a new level of recursivity. The code for each version is based upon two parallelizable functions, the *merge* function and the *multisort* one.

Notice how as shown in *Figure 4* there is a peace of parallelization clauses shared between both strategies, as the initial phase of the parallelization process is the same.

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

Figure 4: Shared structure for both strategies.

Firstly, for the *merge* function we parallelize the call to function *basicmerge* in the base case for the *leaf* strategy (code found in *Figure 5*) while we parallelize every recursive call to the *merge* functions in the recursive decomposition phase for the *tree* strategy (code found in *Figure 6*). Also, for this strategy we need a *taskwait* clause at the end of the recursive decomposition.

```
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 5: Body of the *merge* function from `multisort-omp-leaf.c`.

```
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 6: Body of the *merge* function from `multisort-omp-tree.c`.

Secondly, in the *multisort* function of the *Leaf* strategy (code found in *Figure 7*) we parallelize the call in the base case once again, but this time being the function *basicsort*. Furthermore, several *taskwait* clause are necessary to avoid data races between different tasks. Conversely in the *Tree* strategy (code found in *Figure 8*) every recursive call to function *multisort* and also will become a new task, while a *taskwait* clause is needed before all calls to *merge* functions.

```
  // 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: Body of the *multisort* function from `multisort-omp-leaf.c`.

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

Figure 8: Body of the *multisort* function from `multisort-omp-tree.c`.

3.1.1 Comparison between different numbers of processors

After analyzing the codes of both strategies we have included several speed-up plots obtained for different number of processors as well as captures of Paraver windows in order to reason about the observed performance.

Beginning with the *leaf* strategy, looking into the strong scalability plots shown in *Figure 9* it is obvious that the results are far from the ideal situation.

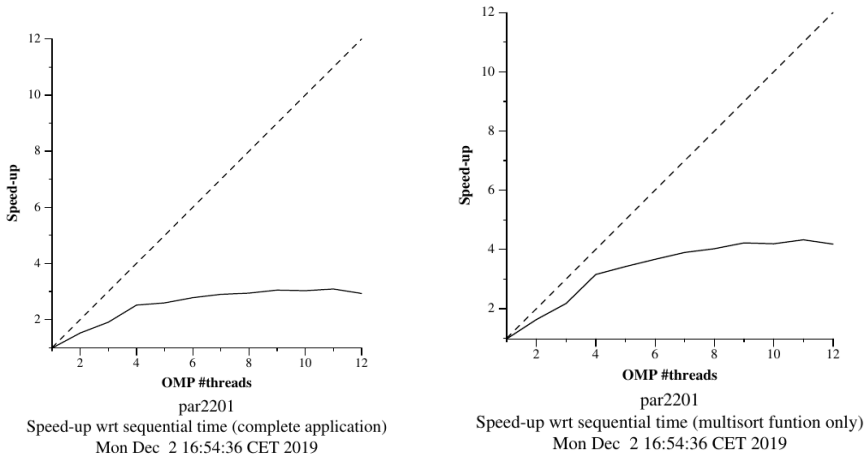

Figure 9: Multisort strong scalability plot for the *leaf* strategy.

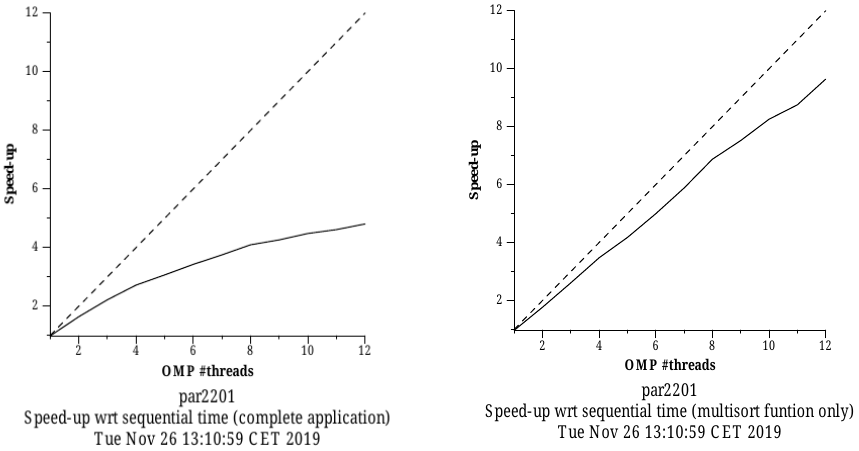

Figure 10: Multisort strong scalability plot for the *tree* strategy.

Nevertheless, that is not the case for the *tree* strategy (seen in *Figure 10*), as the results for the parallelized area are close to the ideal ones. Even so, there is still room for improvement in the performance of the whole program. We will try to complete the parallelization of the *tree* version in section *3.3 Final parallelization of the Tree version.*

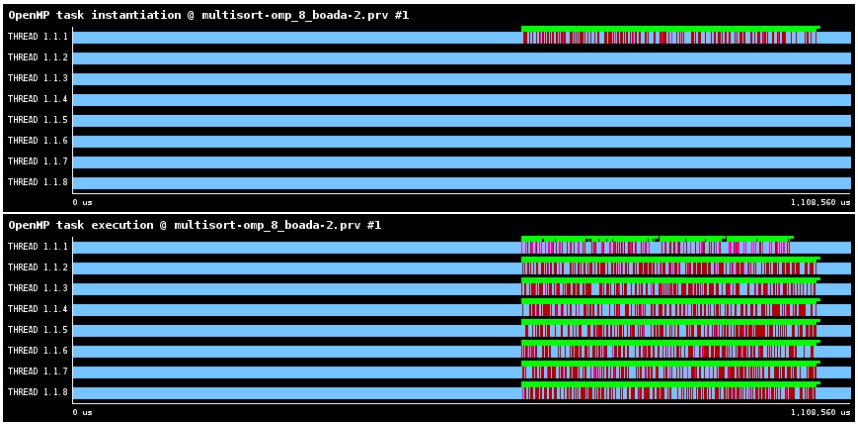

Figure 11: Task instantiation and execution views of the Paraver trace for a *leaf* strategy execution.

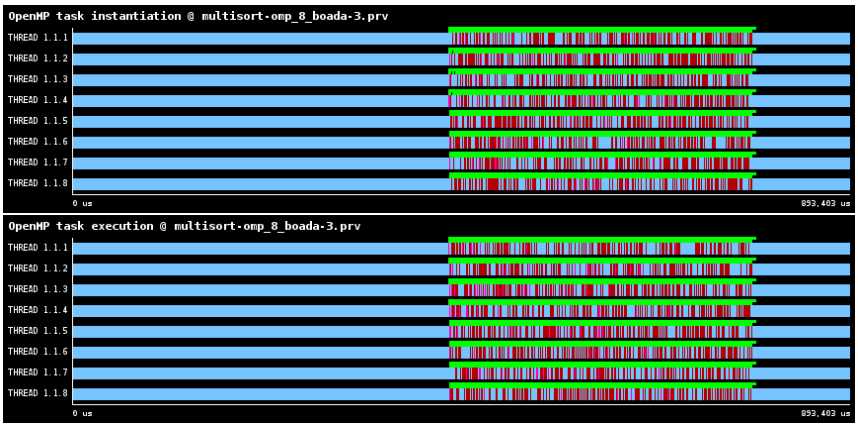

Figure 12: Task instantiation and execution views of the Paraver trace for a *tree* strategy execution.

The top images of both *Figure 11* and *Figure 12* show the task instantiation of the *leaf* and *tree* strategies respectively. Notice how for the *leaf* strategy only one thread instantiates the tasks while for the *tree* strategy all threads collaborate in creating tasks. Though the overall overhead is greater, once it is distributed among threads it performs better.

3.1.2 Task cut-off mechanism

We have not detected any value for the cut-off argument that provides a significant improve in overall performance as seen in *Figure 14*. Any selected value will imply a similar speed-up result to that of *Figure 15*, which shows no improvement in comparison to the original *tree* version.

```
1 void merge(long n, T left[n], T right[n], T result[n*2], long start, long length,
      int level);
```

```
1 void multisort(long n, T data[n], T tmp[n], int level);
```

```
1 #pragma omp task final(level ==  CUTOFF)
```

Figure 13: Body of the *initialize* function from `multisort-omp.c`.

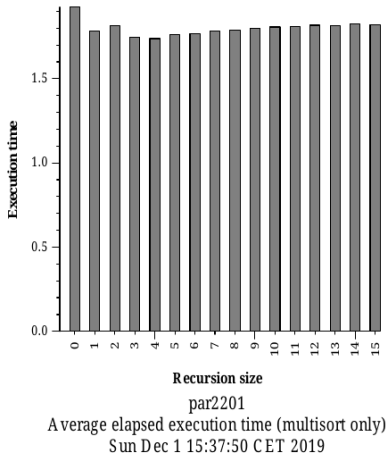

Figure 14: Average execution time plot for the *tree* strategy.

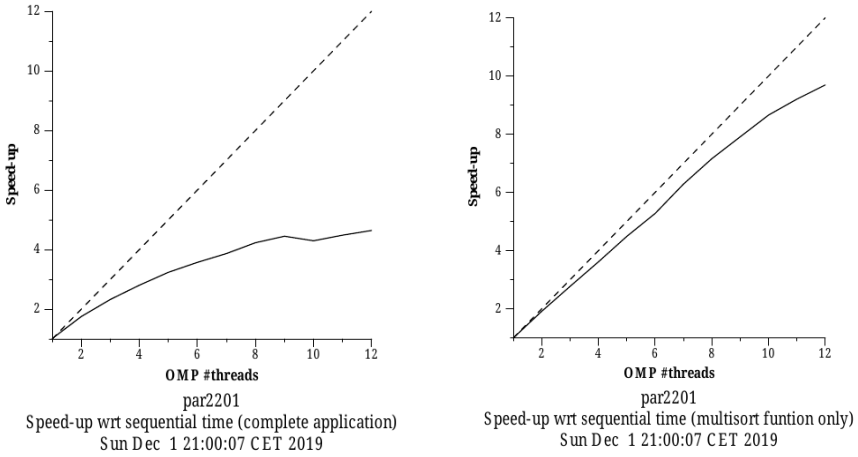

Figure 15: Multisort strong scalability plot for the *tree* strategy with recursive level control.

3.2 Additional scalability analysis for the *Tree* version

In order to go further with our scalability analysis for the *Tree* version we will run it on the other node types available in Boada. It is important to remember that the number of cores are different between boada-5 and boada-6 to 8. Consequently, we have edited the variable $npNMAX$ to specifiy the maximum amount of cores used considering the number of logical cores for each node type.

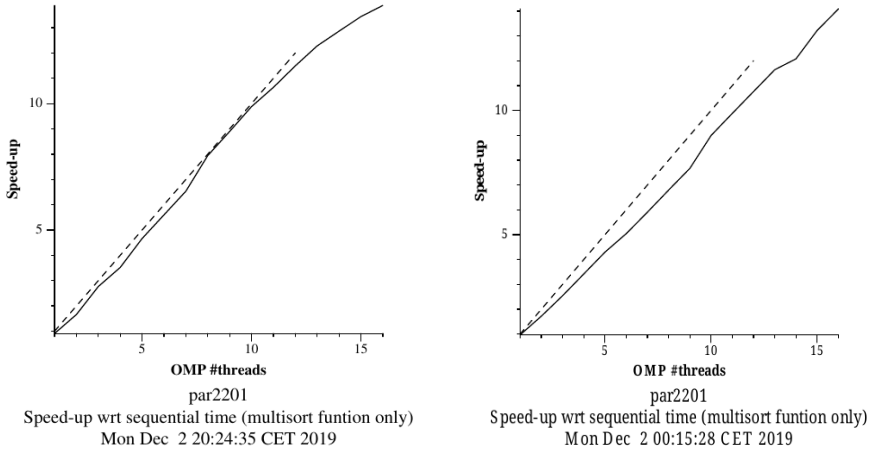

Figure 16: Multisort performance in boada-5 and boada-6

The speed-up plots obtained can be seen in *Figure 16*, the left plot has been executed in boada-5 while the right one has been executed in boada-6. It seems that in both cases the results are closer to the ideal case than the results seen for the *tree* version in previous sections.

3.3 Final parallelization of the *Tree* version

As a final analysis of the *Tree* version we will parallelize the two functions that initialize the `data` and `tmp` vectors.

```
  long i;
# pragma omp parallel
  {
    int n  = omp_get_num_threads();
    int id = omp_get_thread_num();
    data[id*(length/n)] = rand();
    for (i = id*(length/n) +1 ; i < (id+1)*(length/n); i++) {
      data[i] = ((data[i-1]+1) * i * 104723L) % N;
    }
  }
```

Figure 17: Body of the *initialize* function from `multisort-omp.c`.

```
  long i;
# pragma omp parallel
  {
    int n  = omp_get_num_threads();
    int id = omp_get_thread_num();
    for (i = id*(length/n); i < (id+1)*(length/n); i++) {
      data[i] = 0;
    }
  }
```

Figure 18: Body of the *clear* function from `multisort-omp.c`.

As seen in *Figure 17* and *Figure 18* we have parallelized the initialization of the *data* and *tmp* vectors. As a result, we have improved the execution of the program (to be seen in *Figure 19*) in comparison with the original speed-ups for the *tree* version (seen in *Figure 10*).

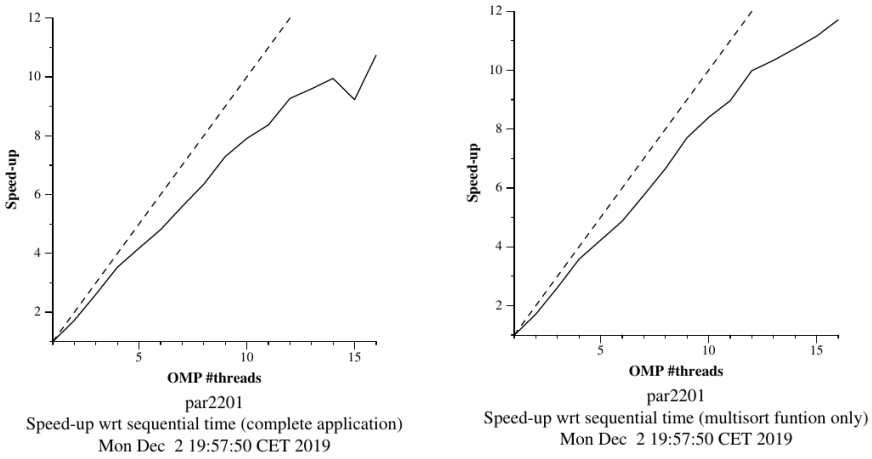

Figure 19: Performance with the initialization parallelized.

It is important to understand that in this case the great difference is shown in the speed-up plot of the whole application (in the left), as we have parallelized other structures of the code besides the *merge* and *multisort* functions.

The task instantiation and execution views from Paraver (*Figure 20*) also help to understand the improvement in the execution of the program.

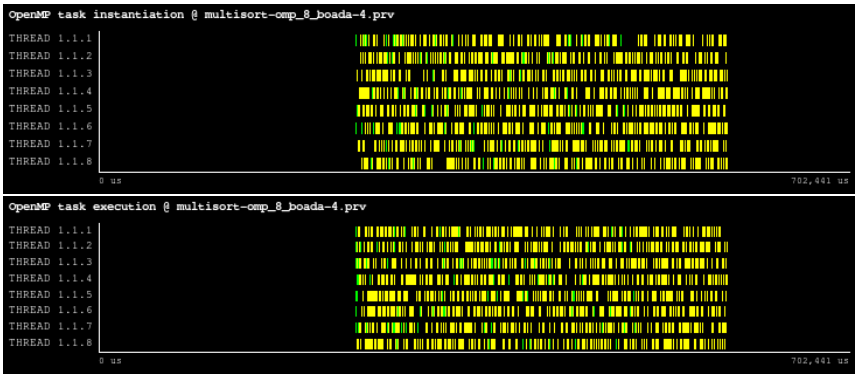

Figure 20: Task instantiation and execution views of the Paraver trace with the initialization parallelized.

4 Parallelization and performance analysis with dependent tasks

In this final section we will change the parallelization of the *tree* version developed throughout previous chapters in order to express dependencies among tasks and therefore avoiding the usage of some *taskwait* clauses.

4.1 Implementation of the *Tree* version with task depencencies

For the purpose of implementing a *tree* version with task dependencies we will use the `task depend (in/out: [dependencies])` clause. Such usage can be seen in *Figure 21* where we specify the values that a particular task will need (with the `in` condition) and the values a task will modify that will be needed by other tasks (with the `out` condition). Perceive how we still need a `taskwait` clause right before the base case.

```
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 21: Body of the *multisort* function from `multisort-omp-deps.c`.

4.1.1 Comparison between different numbers of processors

The developed *tree* strategy with task dependencies management will result in the same amount of threads than those necessary for the original *tree* strategy. As a consequence the task instantiation and execution views in Paraver (*Figure 23*) as well as speed-ups observed in *Figure 22* apparently have the same characteristics than those previously seen in *Figure 10*.

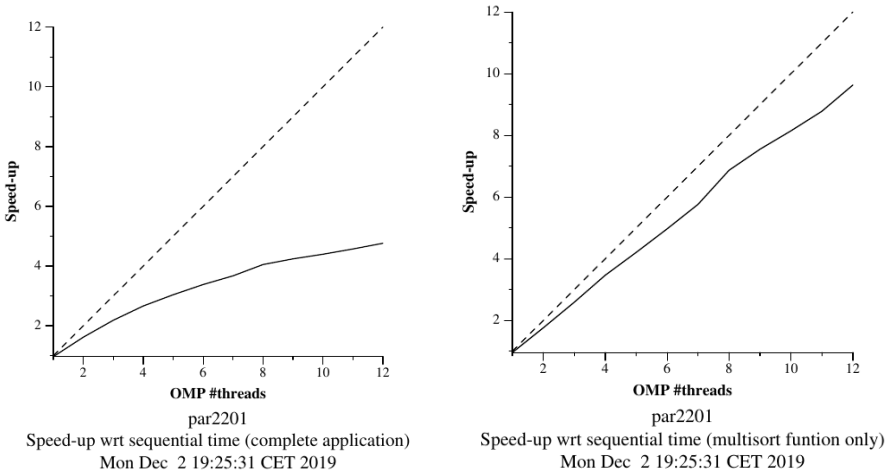

Figure 22: Speed-ups for the *tree* version with task dependencies.

Figure 23: Task instantiation and execution views of the Paraver trace with task dependencies.

5 Conclusions

Throughout this whole project we have studied several possibilities for two parallelization strategies of a multisort algorithm: the *leaf* strategy and the *tree* strategy. The last being the one we have focused the most given that it is normally better than the *leaf* strategy as we have seen.

Also, by improving the *tree* strategy step by step we have been able to understand how important a correct management of task dependencies may be, as well as to comprehend that only parallelizing a small part of the code is not enough to get the best results as possible, we need to try to parallelize as much code as possible.