Laboratory 2

PAR-FIB

Group 22/05

QT 15-16

Albert Suàrez Molgó
Víctor Pérez Martos

INDEX

Analysis with Tareador

1.

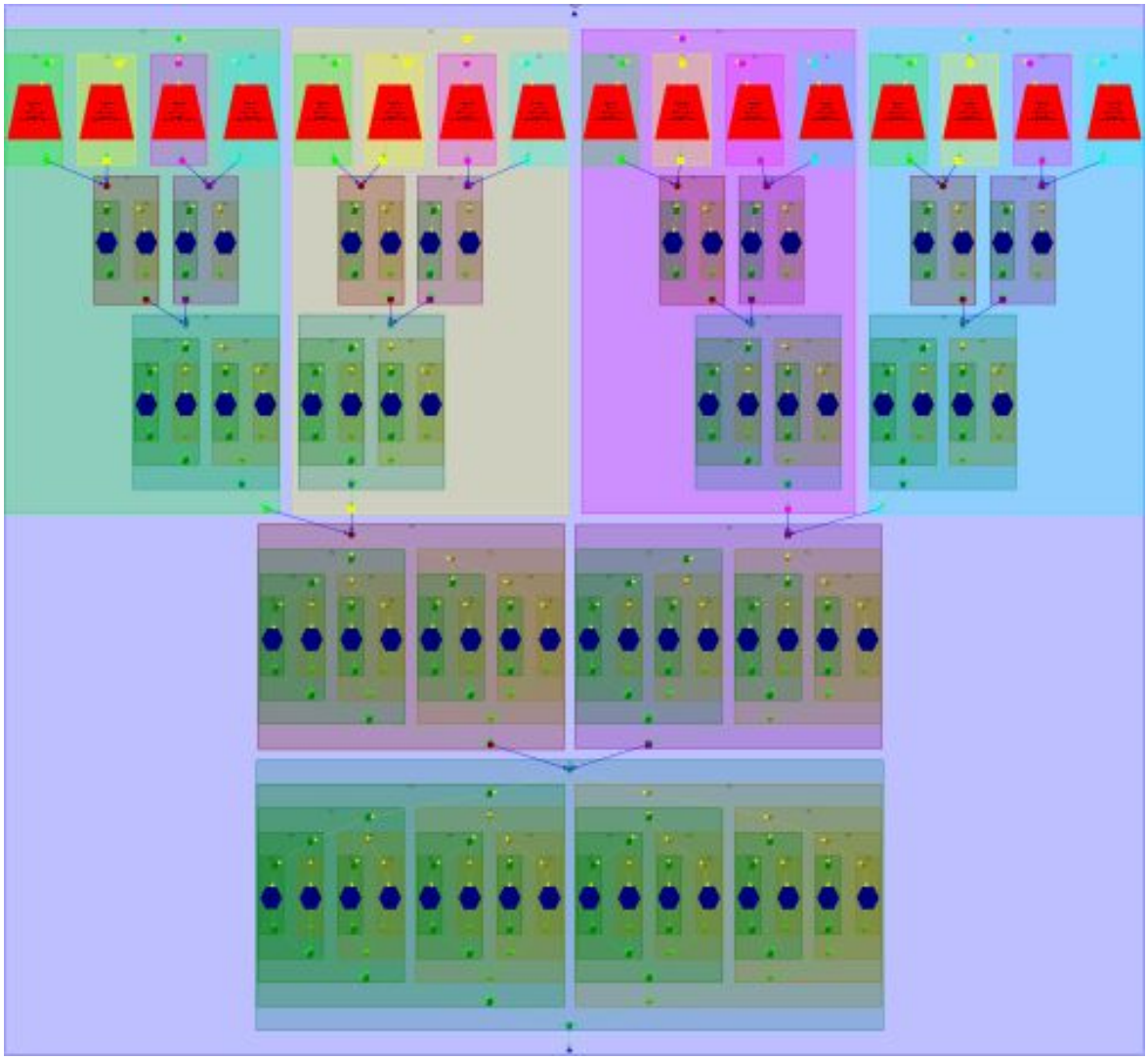


Image 1: Tasks graph generated by Tareador of the *multisort-tareador* execution.

```
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
        tareador_start_task("basicmerge");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("basicmerge");

    } else {
        // Recursive decomposition
        tareador_start_task("merge_4");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("merge_4");

        tareador_start_task("merge_5");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("merge_5");
    }
}

void multisort(long n, T data[n], T tmp[n]) {
    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_3");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("merge_3");
    } else {
        // Base case
        tareador_start_task("basicsort");
        basicsort(n, data);
        tareador_end_task("basicsort");
    }
}
```

Code 1: Tareador's modifications in *multisort-tareador.c*

As shown on Image 1, we have mixed leaf and tree task decomposition and placed a *Tareador* clause for multisort and merge.

The graph obtained, allows us to visualize dependencies between tasks. We can see that first, the initial vector is divided into small segments until it reaches a desired size. Then each segment is ordered and later mixed with other segments. At the end, we'll get the initial vector ordered.

2.

	1	**2**	**4**	**8**	**16**	**32**	**64**
Execution Time (ns)	20,334,421,001	10,173,712,001	5,086,801,001	2,550,377,001	1,289,899,001	1,289,850,001	1,289,850,001
Speed-up	1	1.999	3.997	7.973	15.764	15.765	15.765

Table 1: Execution time and speed-up predicted by *Tareador*.

As we can see on Table 1 the execution time is reduced when we add more threads until we reach the amount of 16 threads, where we get the ideal speed-up and we can't improve the time value. This is because we divide the vector in 16 segments (in this example), so only 16 threads are needed for the parallel executions.

Parallelization and performance analysis with tasks

1.

LEAF VERSION

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

Code 2: OMP pragmas before multisort call on leaf strategy.

```
#pragma omp task
basicmerge(n, left, right, result, start, length);

[...]

#pragma omp task
basicsort(n, data);
```

Code 3: OMP pragmas before basicmerge and basicsort calls on leaf strategy.

As seen on *Code 2* first of all we must indicate that we are going to do a parallel execution, so we add a *#pragma omp parallel* clause before calling the function. We also use a *#pragma omp single* clause to indicate that we only want one only thread to be creating the different tasks.

On *Code 3* we can see that before every call to *basicsort* or *basicmerge* we create a task with *#pragma omp task* to indicate that it can be done by a thread waiting on the single barrier.

```
#pragma omp taskgroup
{
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 taskgroup
{
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);
}
```

Code 4: OMP pragmas before multisort and merge calls on leaf strategy.

Code 4 shows how, to avoid dependency errors between tasks, some recursive callings has been grouped with the *#pragma omp taskgroup clause*.

TREE VERSION

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

Code 5: OMP pragmas before multisort call on tree strategy.

```
#pragma omp task
merge(n, left, right, result, start,
length/2);

[...]

#pragma omp task
multisort(n/4L, &data[0], &tmp[0]);
```

Code 6: OMP pragmas before merge and sort calls on tree strategy.

As seen on *Code 5* first of all we must indicate that we are going to do a parallel execution, so we add a *#pragma omp parallel* clause before calling the function. We also use a *#pragma omp single* clause to indicate that we only want one only thread to be creating the different tasks.

On *Code 6* we can see that before every call to *sort* or *merge* we use the *#pragma omp task* to create a task that can be done by one of the threads waiting on the single barrier.

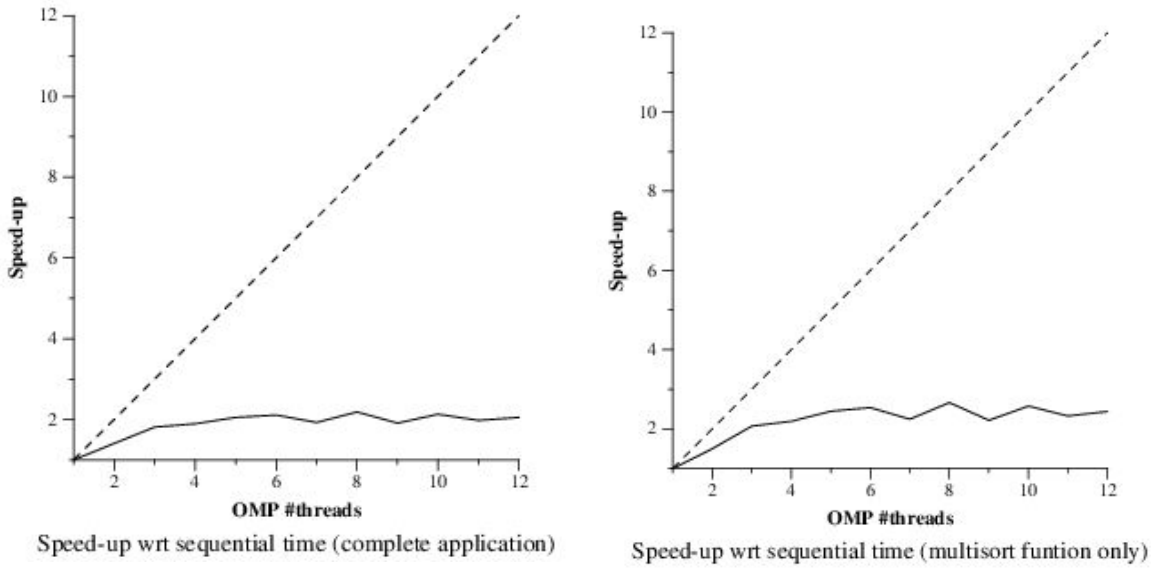
```
#pragma omp taskgroup
{
#pragma omp task
multisort(n/4L, &data[0], &tmp[0]);
#pragma omp task
multisort(n/4L, &data[n/4L], &tmp[n/4L]);
#pragma omp task
multisort(n/4L, &data[n/2L], &tmp[n/2L]);
#pragma omp task
multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
}

#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);
}
```

Code 7: OMP pragmas before multisort and merge calls on tree strategy.

Code 7 shows how, to avoid dependency errors between tasks, some recursive callings has been grouped with the *#pragma omp taskgroup clause*.

2.

LEAF VERSION



Graphic 1: Graphics of the speed-up of the *multisort-omp.c*'s program (leaf version) and its multisort respectively routine.

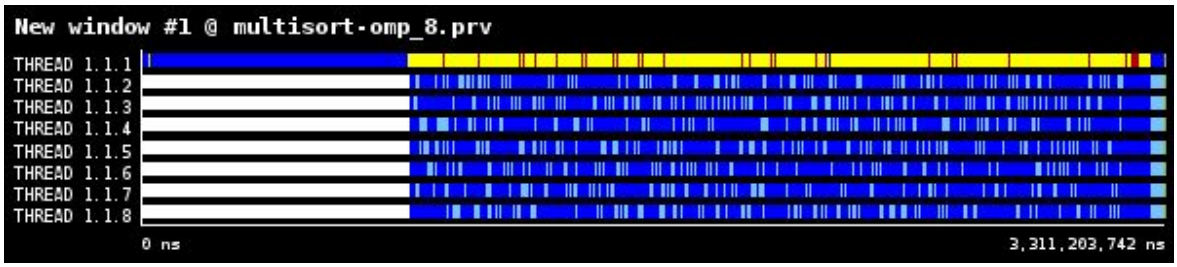
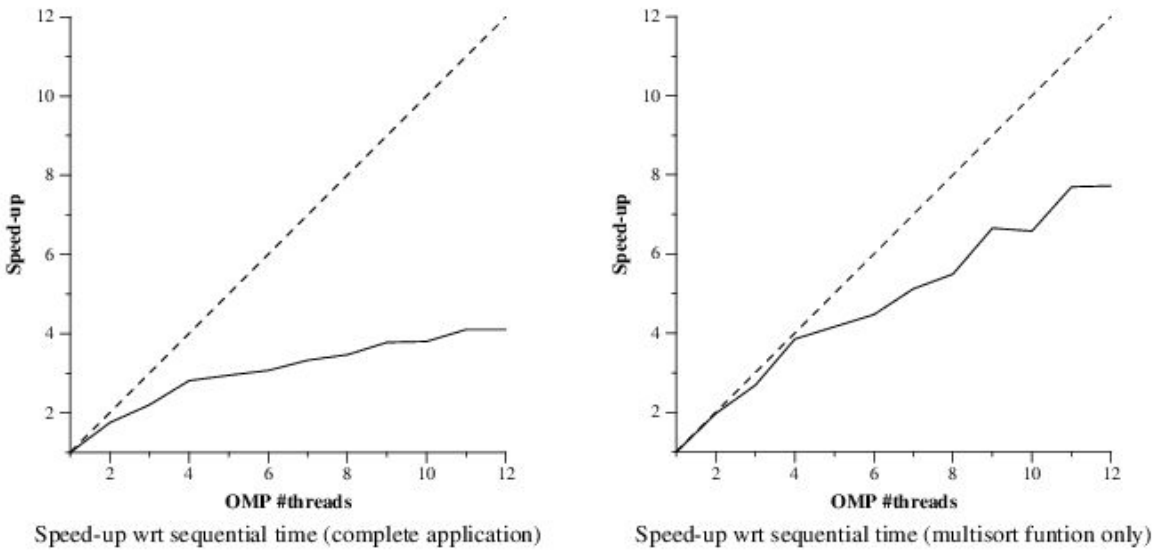


Image 2: Paraver's traces of leaf version execution.

On *Graphic 1*, we can see that the speed-up obtained is not really good when we use more processors. In fact, after the third processor the gain is almost nothing. This is caused because on leaf execution we have to go to the deepest tasks and wait there some time (due to dependences and the *#pragma omp taskgroup*).

TREE VERSION



Graphic 2: Graphics of the speed-up of the *multisort-omp.c*'s program (tree version) and its multisort respectively routine.

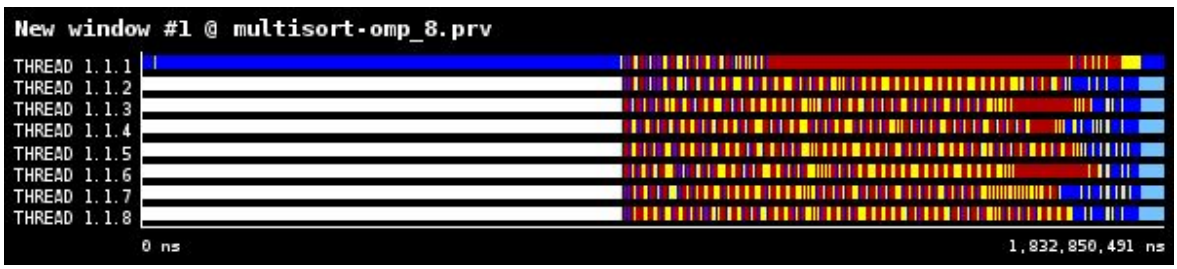
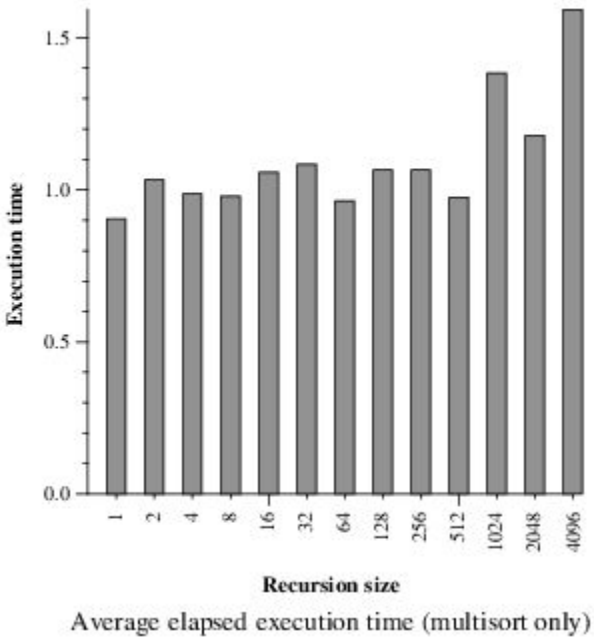


Image 3: Paraver's traces of tree version execution.

On *Graphic 2*, we can see that the speed-up obtained is really good, despite the overhead associated. With this version, we avoid the lack of parallelization caused with the leaf strategy. It is clearly seen, that tree strategy causes better results of time and speed-up (in this case).

3.



Graphic 3: Result plot after executed *multisort-omp* program in tree strategy with differents recursion depths.

Looking at the graph, we can appreciate that the recursion size on recursivity depth doesn't have a particular impact on the execution time. Nevertheless, we can see a meaning difference when we use big sizes. This is because when we use big sizes, we generate less tasks, so we decrement the parallelization part of the program.

Parallelization and performance analysis with dependent tasks

1.

```
#pragma omp taskgroup
{
    #pragma omp task
    merge(n, left, right, result, start, length/2);

    #pragma omp task
    merge(n, left, right, result, start + length/2, length/2);
}
```

Code 8: OMP pragmas before merge calls on tree strategy using task dependencies.

On *Code 8*, we can see that we execute every merge call as a task. Both of them are inside a *taskgroup* clause to ensure that we execute both before returning to the parent function.

```
#pragma omp task depend(out:data[0:n/4L])
multisort(n/4L, &data[0], &tmp[0]);

#pragma omp task depend(out:data[n/4L:n/4L])
multisort(n/4L, &data[n/4L], &tmp[n/4L]);

#pragma omp task depend(out:data[n/2L:n/4L])
multisort(n/4L, &data[n/2L], &tmp[n/2L]);

#pragma omp task depend(out:data[3L*n/4L:n/4L])
multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
```

Code 9: OMP pragmas before multisort calls on tree strategy using task dependencies.

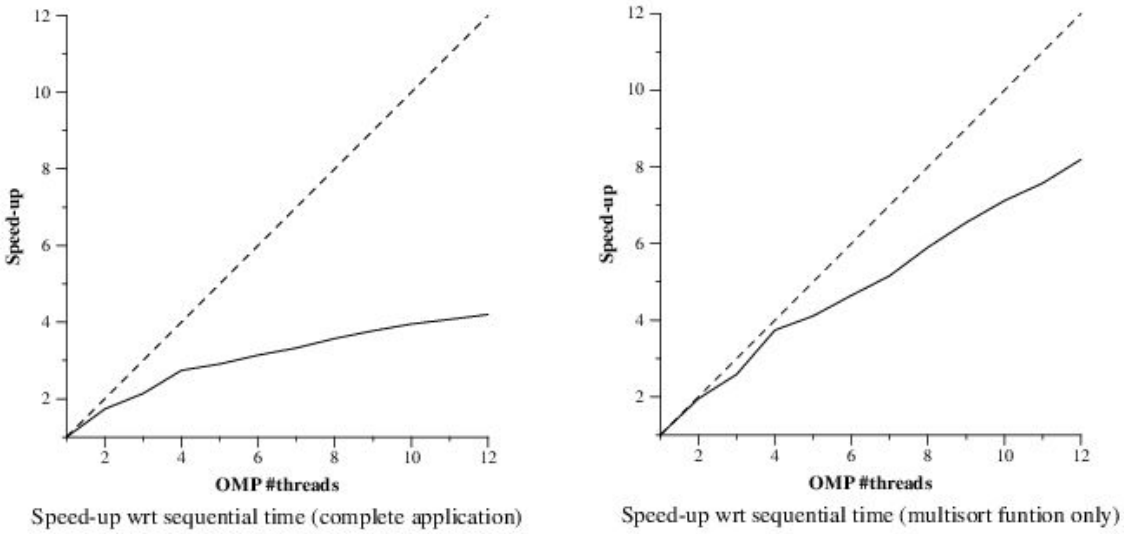
Code 9 shows how every recursive call to multisort, has been divided into tasks. Moreover, we have used the clause *depend* to generate a dependence between them and avoid the use of *taskgroup* clauses.

```
#pragma omp task depend(in:data[0:n/4L], data[n/4L:n/4L]) depend(out:tmp[0:n/2L])
merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);

#pragma omp task depend(in:data[n/2L:n/4L], data[3L*n/4L:n/4L])
depend(out:tmp[n/2L: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:n/2L], tmp[n/2L:n/2L])
merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

#pragma omp taskwait
```

Code 10: OMP pragmas before merge calls on tree strategy using task dependencies.

Code 10 does exactly as *Code 9,* where we execute every recursive call as a task and we avoid the *taskgroup* clause by generating a dependence between tasks. Additionally, we have added a *#pragma omp taskwait* clause at the end to avoid going one level up on the recursion if one thread hasn't already ended.

2.



Graphic 4: Graphics of the speed-up of the *multisort-omp.c*'s program (tree version) and its multisort respectively routine using task dependencies.

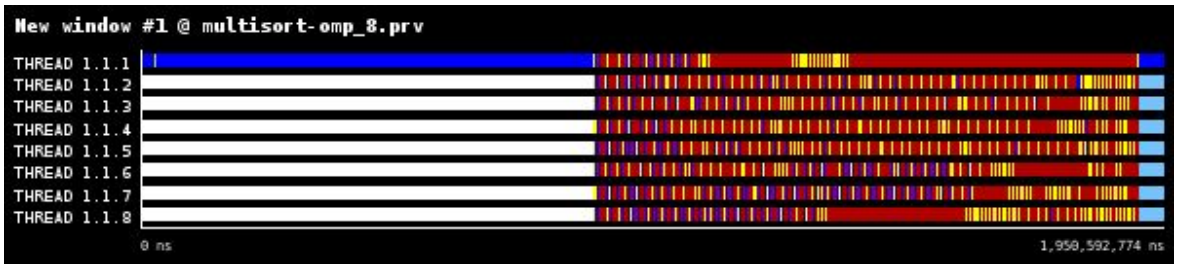


Image 4: Paraver's traces of tree version execution using task dependences.

Looking at *Image 4* and *Graphic 2*, we can see that they both are pretty similar. It can also be seen comparing the timelines with paraver. This is caused because the parallelism that we can get is pretty limited because of the dependence between task (to do a parent's task, we must have done the son's task).

Optional

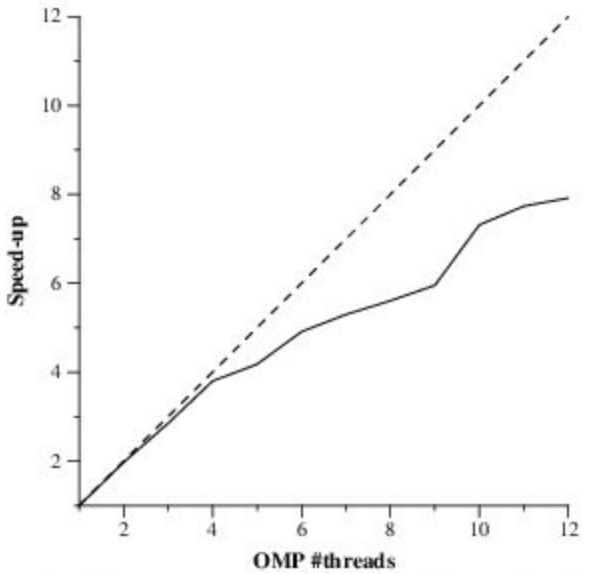
1.

```
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;
        }
    }
```

Code 11: Parallelized code of initialize function.

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

Code 12: Parallelized code of clear function.



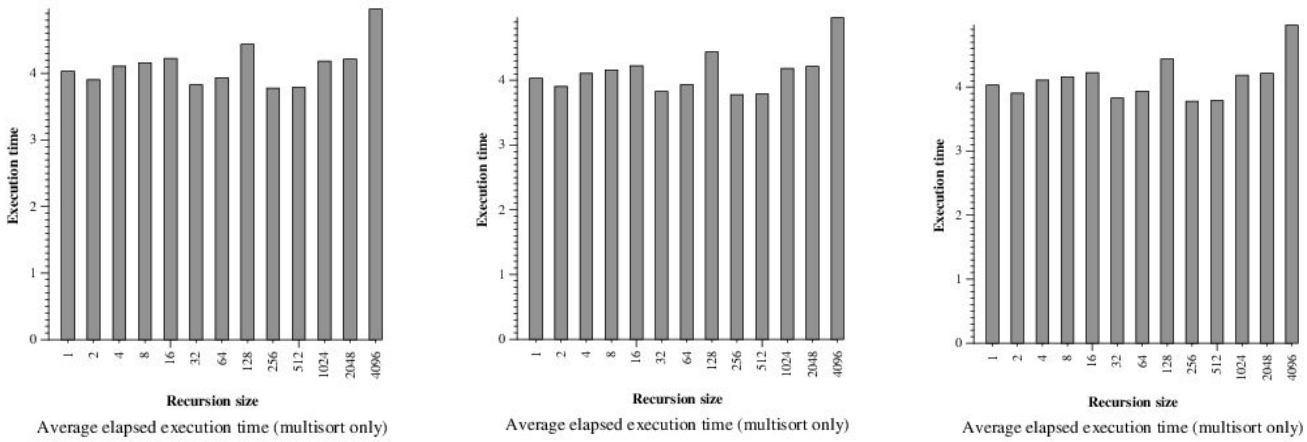
Speed-up wrt sequential time (complete application)

Graphic 5: Result plot of executed *multisort-omp* with initialize and clear functions parallelized.

Looking at *Code 11* and *Code 12*, we can see how we have parallelized the functions to initialize the vector used later on *multisort.* As *graphic 5* shows, the gain obtained with this parallelization is really good.

We can clearly see the improvement by comparing *graphic 5* with *graphic 2* (which shows the speedup with the parallelization indicated before).

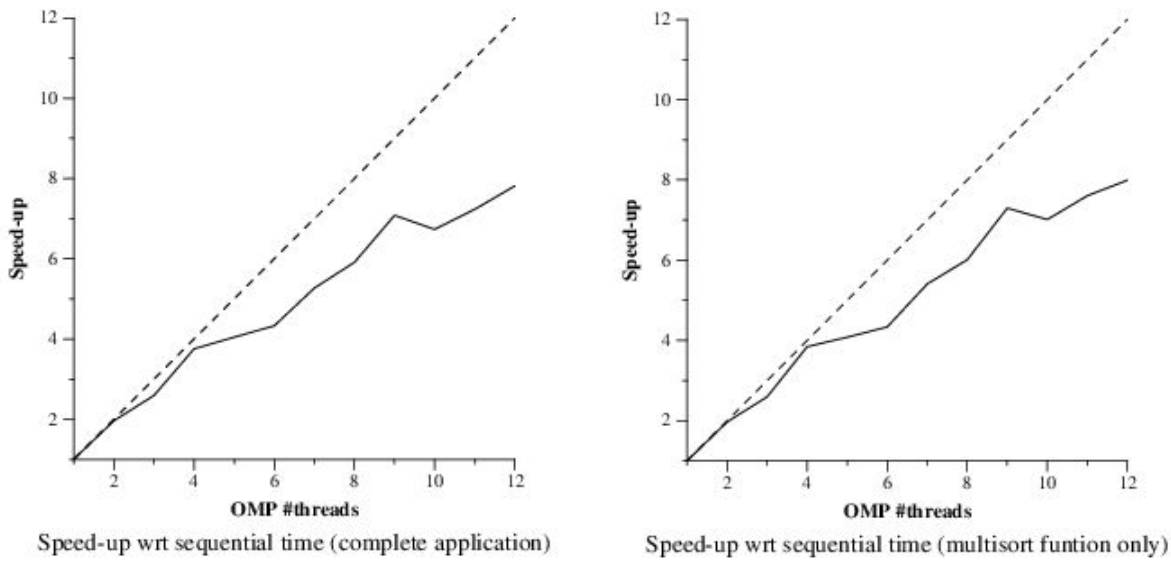
2.



Graphic 6: Graph evolution of the execution of *multisort-omp.c* program for various depths (*merge size)* and *sort size* (from left to right: 128, 256 and 512 Kilo-Elements).

After changing the sizes of the organization (*sort size*) and testing various combinations of deep levels (from *1* to *4096*) we can determinate that the best combinations available are 128 for *sort size* and 256 for *merge size* and vice versa.

In conclusion, we can obtain the graphics with 128 Kilo-elements for *sort size* and 256 Kilo-elements for *merge size*.



Graphic 7: Result plots of executed *multisort-omp* with 128 Kilo-elements for *sort size* and 256 Kilo-elements for *merge size*.