Parallelism

Lab 2

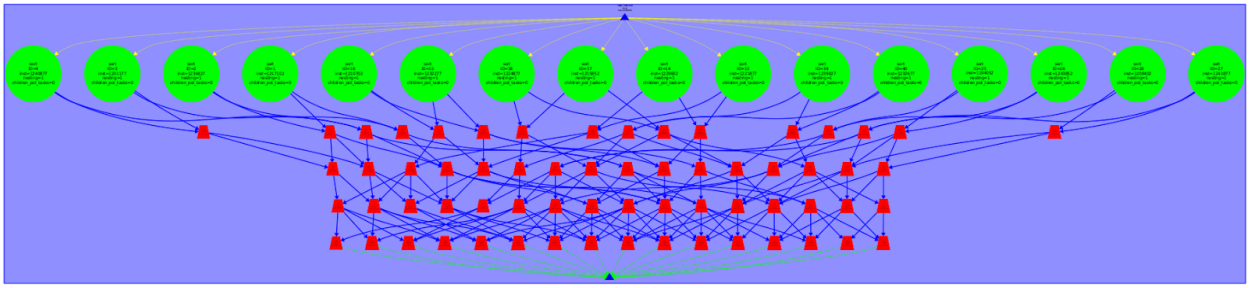
Gerard Bayego
Martín Dans

Grup: par3101
Curs: 2014-15
Fall semester

Question 1.1:

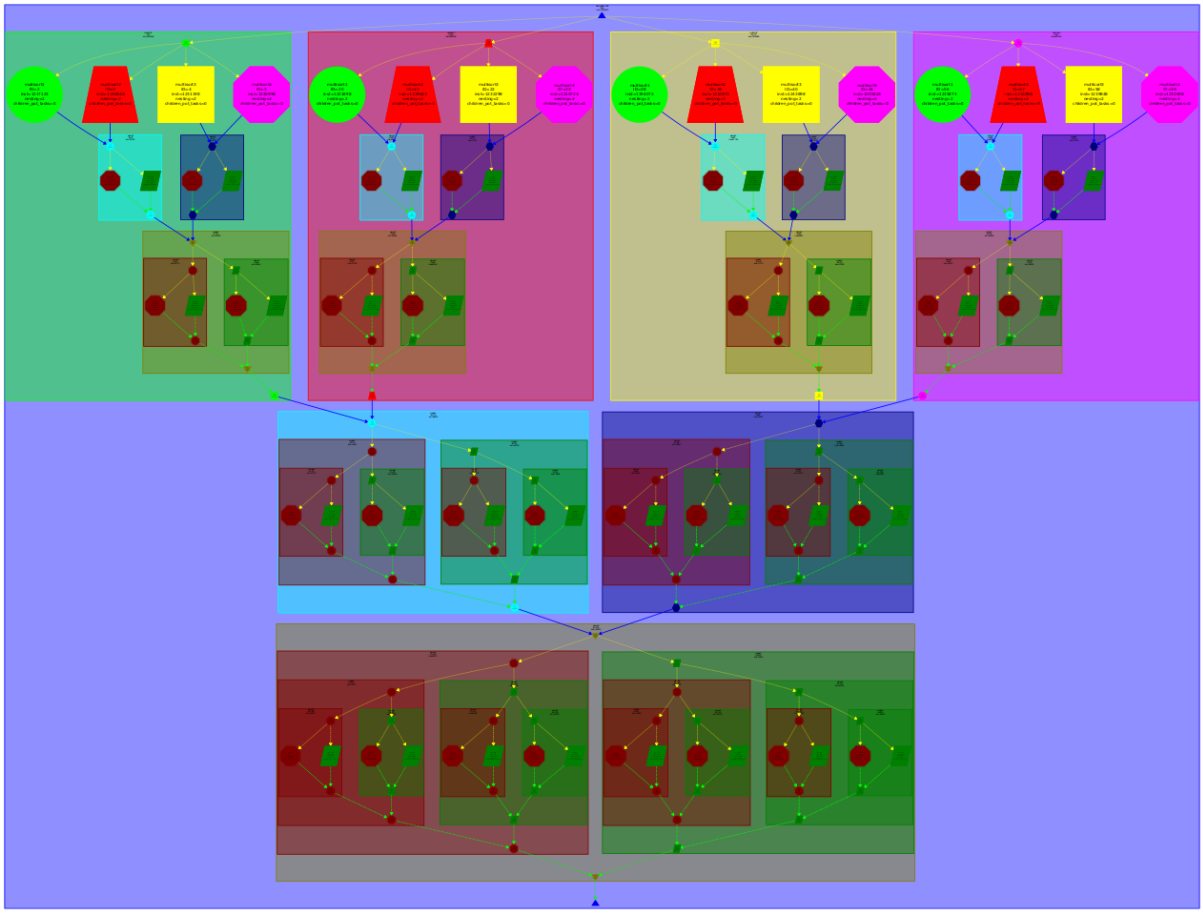
Leaf tareador (multisort_tareador_leaf.c): We have added the following code at the base cases of the recursion to generate the leaf dependencies:

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

tareador_start_task("sort");
basicsort(n, data);
tareador_end_task("sort");
```



Tree tareador (multisort_tareador_tree.c): We have added to all the calls of the functions that of the recursivity to get a clearly view of the tree generated and the dependencies the tasks have. (Example code, this is repeated for every call)

```
tareador_start_task("merge4");
merge(n, left, right, result, start, length/2);
tareador_end_task("merge4");

tareador_start_task("multisort1");
multisort(n/4L, &data[0], &tmp[0]);
tareador_end_task("multisort1");
```



The basicsort function doesn't need any task before. That's because it's basically a calculation inside the initial vector splitted. (An element can only be at one splitted part).

To see how the merge works we have to look the the tree graph. Every merge we call is subdivided in other two merges. To do a merge we need two tasks before we can execute it and this merge is needed for another merge except the last one obviously.

Question 1.2:

	1	2	4	8	16	32	64
Tareador Time	20340	10176	5089	2548	1291	1291	1291
Speedup	-	2	4	7,98	15,76	15,76	15,76

Basically we have 16 big tasks that are the basicsort function. Once this is done with full parallelism (16 processors) the scalability of the program stops as the merge tasks are insignificant compared with the other. Before the 16 it's normal that the times is halved because we are using divisors of 16 number of processors and we are doubling the number.

Question 2.1:

Leaf:
To the leaf version, we have added tasks in every base call, and we group some of the recursion calls with task group to avoid dependence problems:

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

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

- We have added a taskgroup at merge functions to force both of them to finish before continuing the execution since we have basicmerges as a tasks.

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

- Also we have to wait the multisort tasks before we can start merge tasks.

```
#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]);
}
```

- Once again we have to wait the merge functions of the half vector before we can execute the third merge to merge the two parts.

```
#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);
}
merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
```

Tree:

To the tree version, we have added tasks in every recursive call, and we group some of them with task group to avoid dependence problems:

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

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

As we can see in this code we have implemented the tasks at all the calls of multisort and merge functions, and we have packed them with taskgroup:

- Function multisort has 4 multisorts packed:

```
#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]);
}
```

- Function multisort has 2 merges packed:

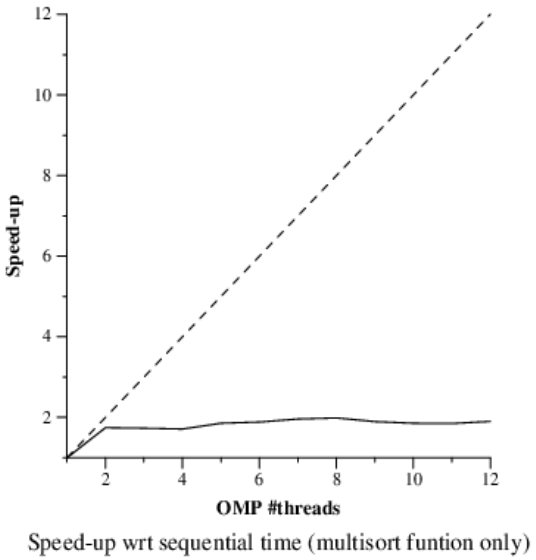
```
#pragma omp taskgroup
{
    #pragma omp task
    merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
    #pragma omp task
     merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0,
n/2L);
}
```

- Function merge has 2 merges packed:

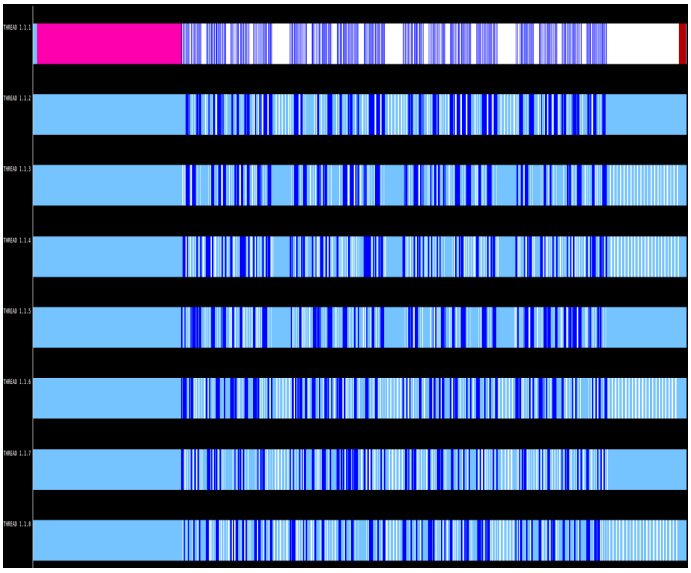
```
#pragma omp taskgroup
{
    #pragma omp task
    merge(n, left, right, result, start, length/2);
    #pragma omp task
    merge(n, left, right, result, start + length/2, length/2);
}
```

Question 2.2:

Leaf:

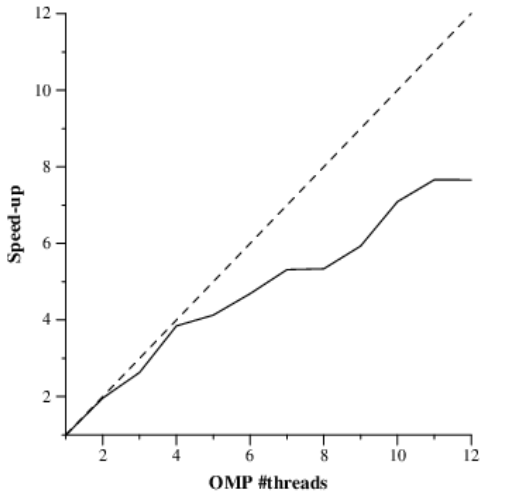


Speed-up wrt sequential time (multisort funtion only)

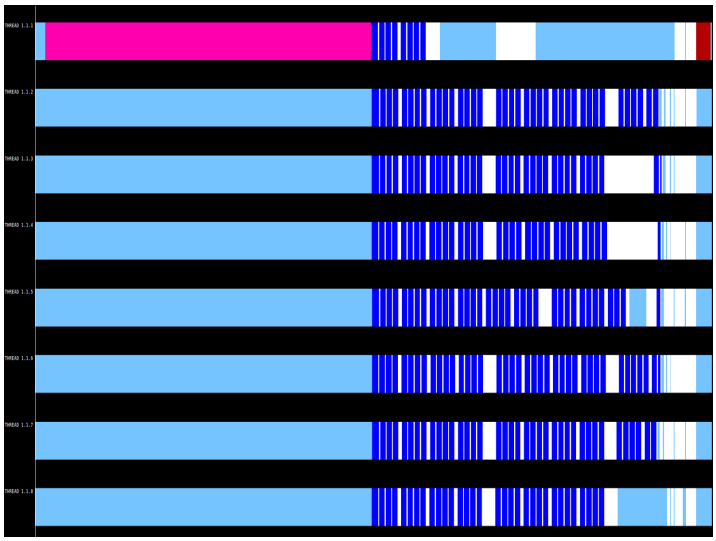


Leaf execution has a big problem. We can observe easily that it does not get a good speed-up even with a lot of processors. In fact, after the second processor it practically doesn't gain speed-up. That's because the recursion has to go to the deeper task, throw all the tasks in that block and wait for all of them, so basically you are executing the lowest level of the recursion i parallel the other ones are sequential. It's impossible to get more than a 4 speed-up.

Tree:

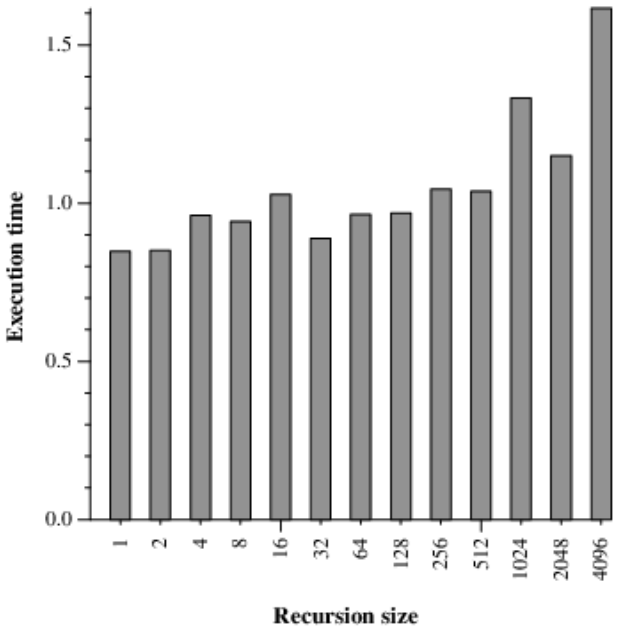


Speed-up wrt sequential time (multisort funtion only)



The tree code doesn't suffer from the problem that leaf has because while we are going down we are generating tasks that can be executed in parallel and with the group task we are forcing them to wait their neighbours. So basically, we are executing all the levels of the recursion in parallel and the speed-up is close to the ideal speed-up. Of course it have overheads that prevent that.

Question 2.3:



Average elapsed execution time (multisort only)

If we assume that we are creating the same leafs tasks and the only difference between executions is the depth level, at every level of the recursion we call more or less subfunctions, we can conclude that the recursion depth in the tree version practically has no impact, as in this version, we are full parallelizing the leaf tasks. We can also see this in the plot generated. The only difference is the overhead that making too many tasks has. We are increasing the number of tasks because we have to add more intermediate tasks to go deeper.
Example of the explication: The parallelism is the same when calling eight subfunctions in one level and calling two subfunctions in three levels. Otherside calling eight subfunctions has less intermediate tasks than calling only two more than once.

Question 3.1:

In this version we have added pragma omp task with dependences, so we can block execution of some tasks if they have dependences not processed yet.

- In multisort function, we have added pragma omp task depend out at all the tasks of recursion calls of multisort.

```
#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]);
```

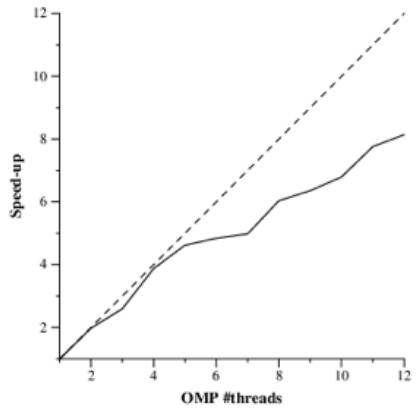
- In multisort function, we have added also pragma omp task depend in and out at all the tasks of recursion calls of merge. And a taskwhait at the end of recursion, because we cannot continue before the sub-problems of recursion have ended.

```
#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
```

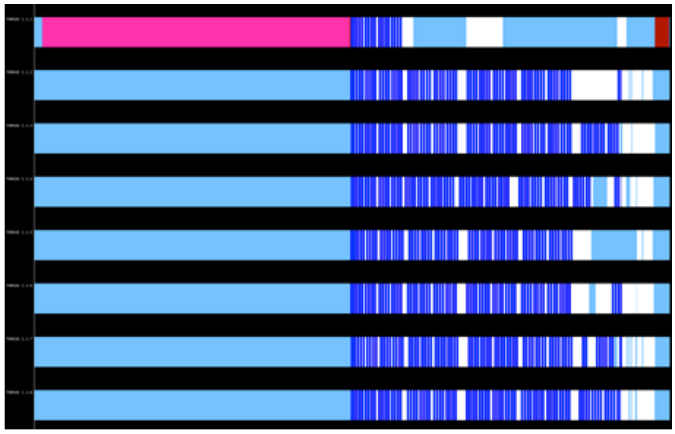
- At the merge function we find one taskgroup with two tasks, the same that the previous version, because we cannot continue execution of parent-calls before this recursion has ended.

```
#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);
}
```

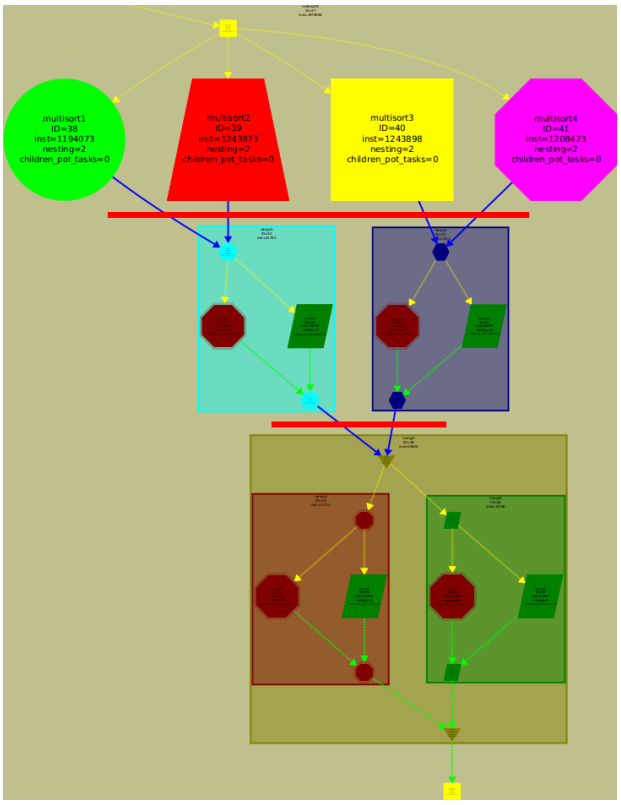
Question 3.2:



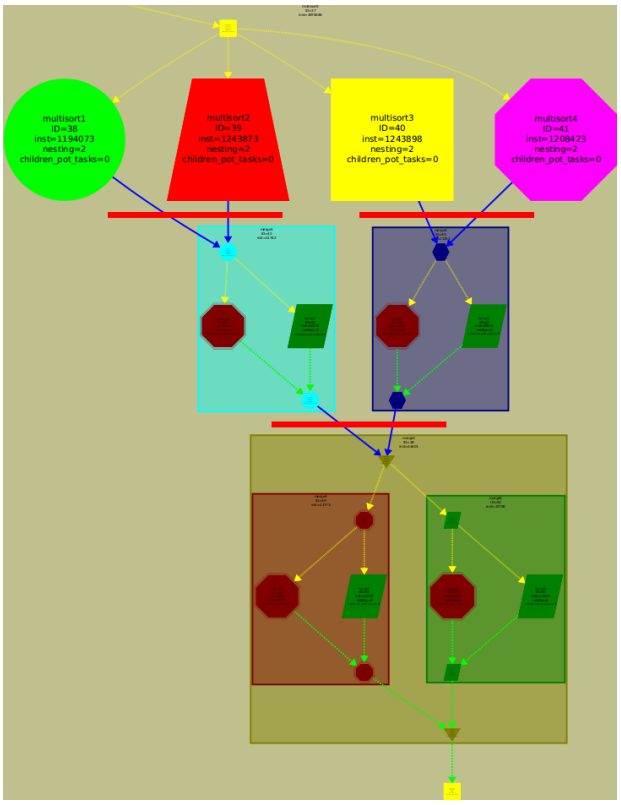
Speed-up wrt sequential time (multisort funtion only)



We can see that the speed-up plots of task tree and task dependence tree, are so similar, near to equals. This is caused because we cannot increase parallelism so much limited by the las merge in each recursion call. The last merge needs the result of the two previous merges, and each of programs implement a different gaph of synchronization barriers:



omp tasks (taskgroup)



omp tasks (dependences)

As we can see the first option waits all the multisort results, after execute first two merges and finally last merge. But we didn't need to wait all multisorts to compute merges, we need to wait multisort 1 and 2 to compute merge 1 and multisort 3 and 4 to compute merge 2. The version with omp task dependences implements this version that if the execution time of multisorts and merges are a bit differents, we can obtain some extra speed-up, but is not the case, and we have speed-up plots of two versions too similars.