LAB 4: DELIVERABLE

Alfonso Toro Roll
Albert Figuera Pérez
PAR1313
Q1 2017/2018

INDEX

INTRODUCTION

In this session we'll work with a '***divide and conquer***' rule. In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A *divide and conquer* algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.
Specifically, we'll work with ***multisort** and **merge*** algorithms. Multisort is a sorting algorithm which combines a "divide and conquer" mergesort strategy that divides the initial list into multiple sublists recursively, a sequential quicksort then is applied when the size of these sublists is small enough, and then a merge of the sublists back into a single sorted list is done.
We will use two different approaches of this algorithm to exploit parallelization in our program: Leaf and Tree.

· In the ***Leaf version*** we should define a task for the invocations of basicsort and basicmerge once the recursive divide–and–conquer decomposition stops.

· In the ***Tree*** version we should define tasks during the recursive decomposition, i.e. when invoking multisort and merge.

Analysis with Tareador

1. Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed.

The calls from the Tareador API used in our code are tareador_start_task() and tareador_end_task() each used four times, once for each part of the code (multisort's recursive decomposition/base case and merge's recursive decomposition/base case).

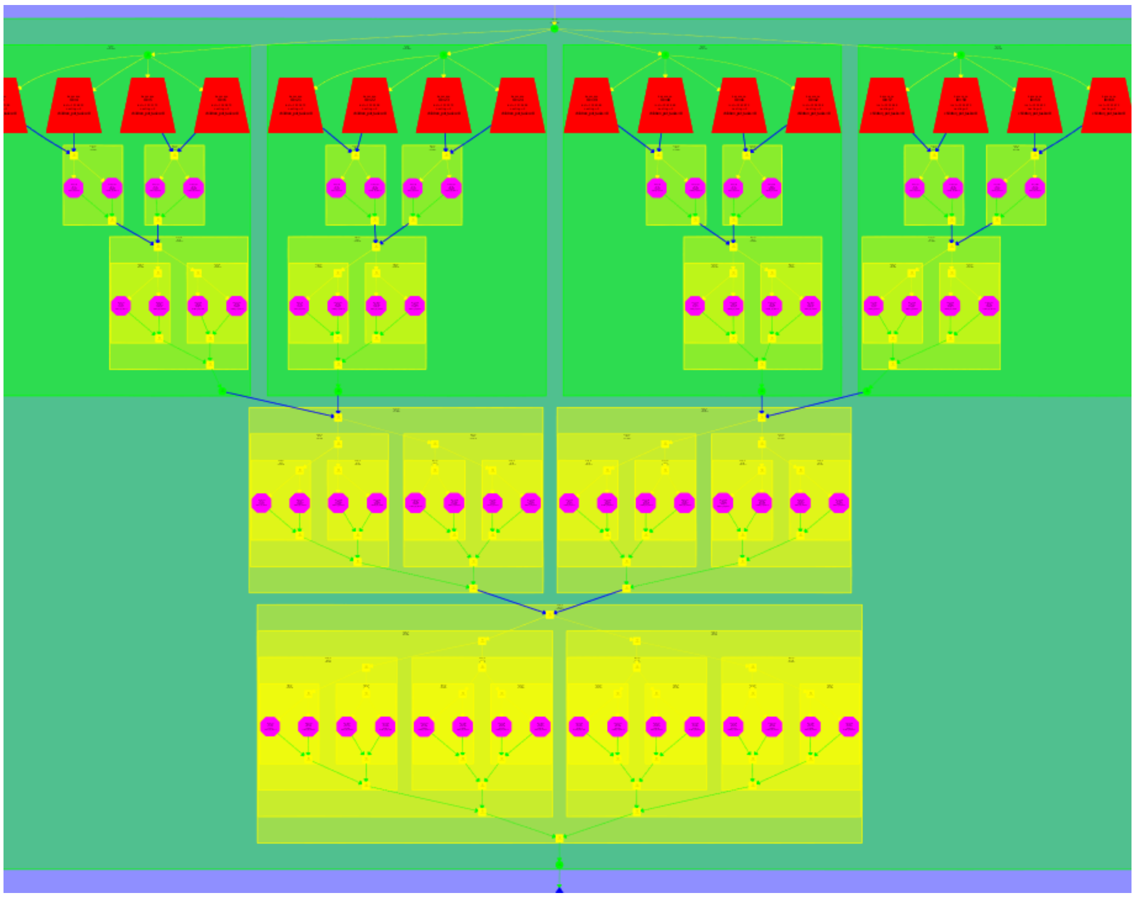


Image 1: tasks graph generated by Tareador of the multisort-tareador execution.

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

We can see the dependencies between tasks. Also we can see 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. After this we'll get the initial vector ordered.

2. Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

#P	Execution time	Speed-up
1	20.334.421.001ns	1
2	10.173.679.001ns	1.998
4	5.087.963.001ns	3.997
8	2.548.394.001ns	7.979
16	1.289.949.001ns	15.764
32	1.289.949.001ns	15.764
64	1.289.949.001ns	15.764

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

The minimum number of processors we need to reach maximum parallelization is 16, until then, doubling the number of processors reduces in half the execution time. Once we have reached 16 processors each segment of the program is assigned a thread, adding more processors doesn't change the execution time because all segments are already assigned.

Parallelization and performance analysis with tasks

1. Include the relevant portion of the codes that implement the two versions (Leaf and Tree), commenting whatever necessary.

Tree version:

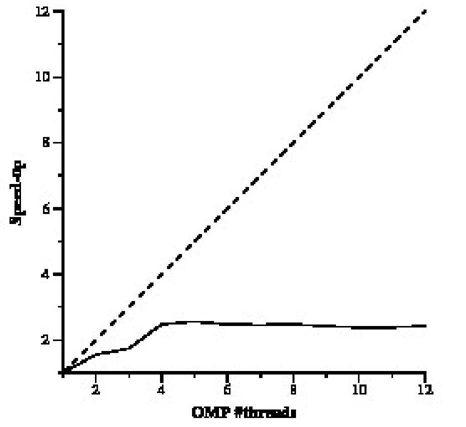
We must create the parallel section with *#pragma omp parallel* before calling the multisort function, we also need *#pragma omp single* in order to make the call just once.
A task is then assigned to each *multisort* and merge call in the multisort function, with some integrity control offered by *#pragma omp taskwait.*
A task is also assigned to each merge call in the merge function.

Leaf version:

First of all, like tree version, we must indicate that we are going to do a parallel execution with *#pragma omp parallel* before calling the multisort function. We only want one thread to be creating different task so we use *#pragma omp single* clause.
In this version we only create tasks when calling *basicsort* and *basicmerge* functions.

2. For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

Leaf version:



Speed-up wrt sequential time (multisort funtion only)

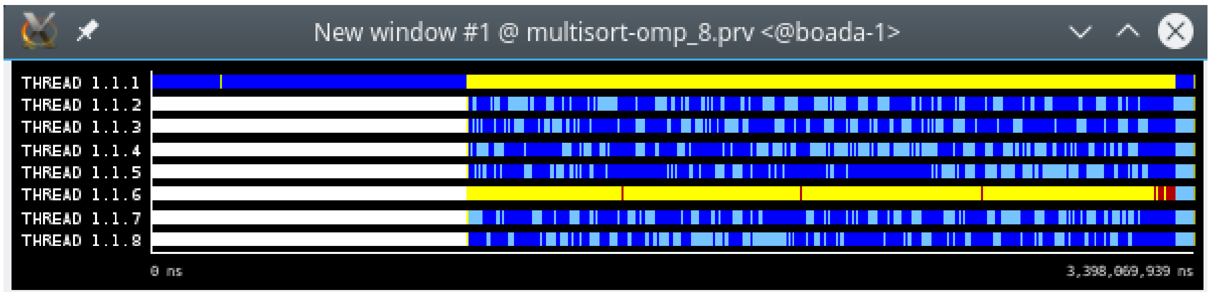
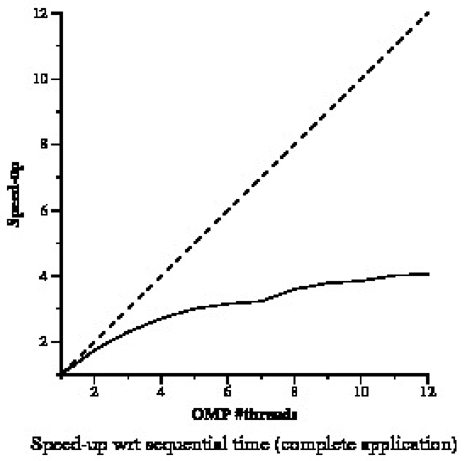
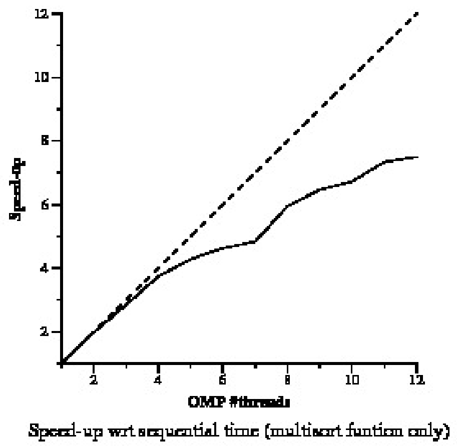


Image 2: Paraver's traces of leaf version execution

As we can see, on the speed up plots, the speed-up obtained is not very good when we increment the number of processors. We can see when we have 4 processors we obtained the higher speed. The reason of this is caused because on leaf execution we have to go to the deepest tasks and wait there some time.

Tree version:





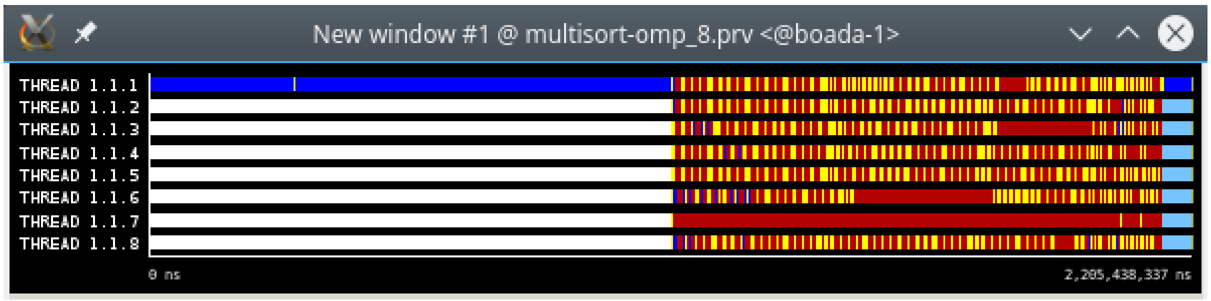
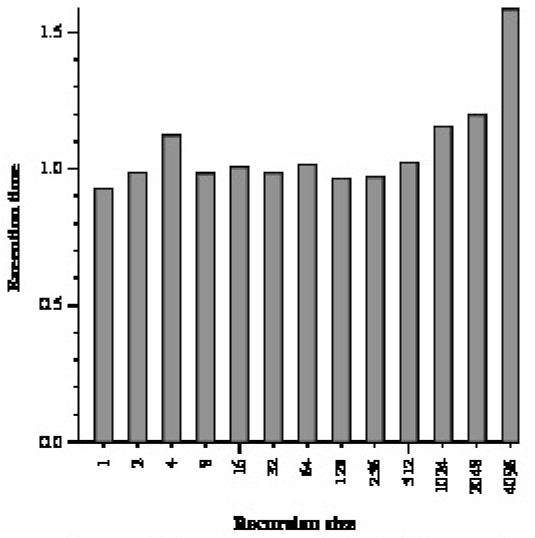


Image 3: Paraver's traces of tree version execution

This version is clearly better than the leaf version despite the overhead associated, each recursive call before reaching the leaves in multisort receives a new task, exploiting parallelization in a much higher level.

3. Analyze the influence of the recursivity depth in the Tree version, including the execution time plot, when changing the recursion depth and using 8 threads. Reason about the behavior observed. Is there an optimal value?



Average elapsed execution time (multisort only)

We consider that the optimal value would be 1, since increasing the recursion size also increases the number of “child” tasks each task has, which it will have to wait. Therefore, we want the smallest recursion size possible in our program.

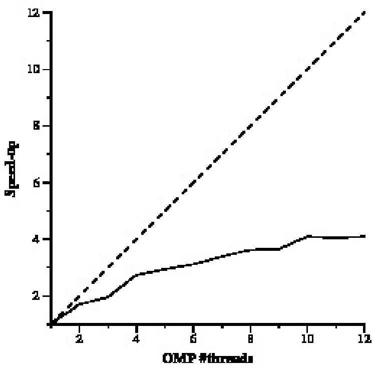
Parallelization and performance analysis with dependent tasks

1. Include the relevant portion of the code that implements the Tree version with task dependencies, commenting whatever necessary.

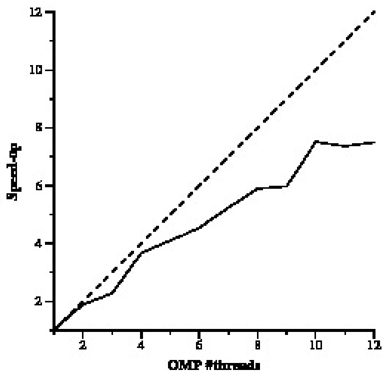
```
...
#pragma omp task depend(out:data[0:n/4L])
multisort(n/4L, &data[0], &tmp[0]);
#pragma omp task depend(out:data[n/4L:n/4L])
multisort(n/4L, &data[n/4L], &tmp[n/4L]);
#pragma omp task depend(out:data[n/2L:n/4L])
multisort(n/4L, &data[n/2L], &tmp[n/2L]);
#pragma omp task depend(out:data[3L*n/4L:n/4L])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


#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);
...
```

The specified depend clauses create dependency between task using (in:X) and tasks using (out:X). Once a task with an out clause has finished, the depending task with an in clause can start its own execution.

2. Reason about the performance that is observed, including the speed–up plots that have been obtained different numbers of processors and with captures of Paraver windows to justify your reasoning.



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

We can see the graph that we've obtained in tree version execution (Image 3) is very similar to this plot. We expected better results because we didn't think that the *taskwait* version would be so close to the dependency version in execution time results. This might be caused by the small recursion size of the program, if it were deeper, the dependency version would be much better in comparison to the *taskwait* version.

Optional

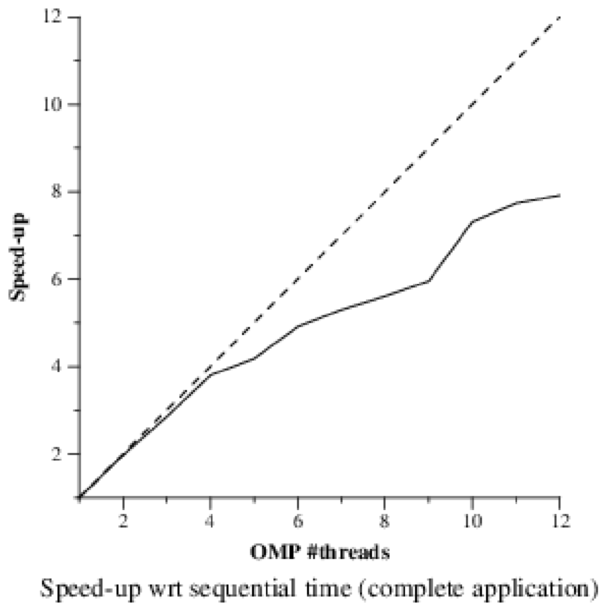
Optional 1: Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors1 . Analyze the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines.

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

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;
        }
}
```

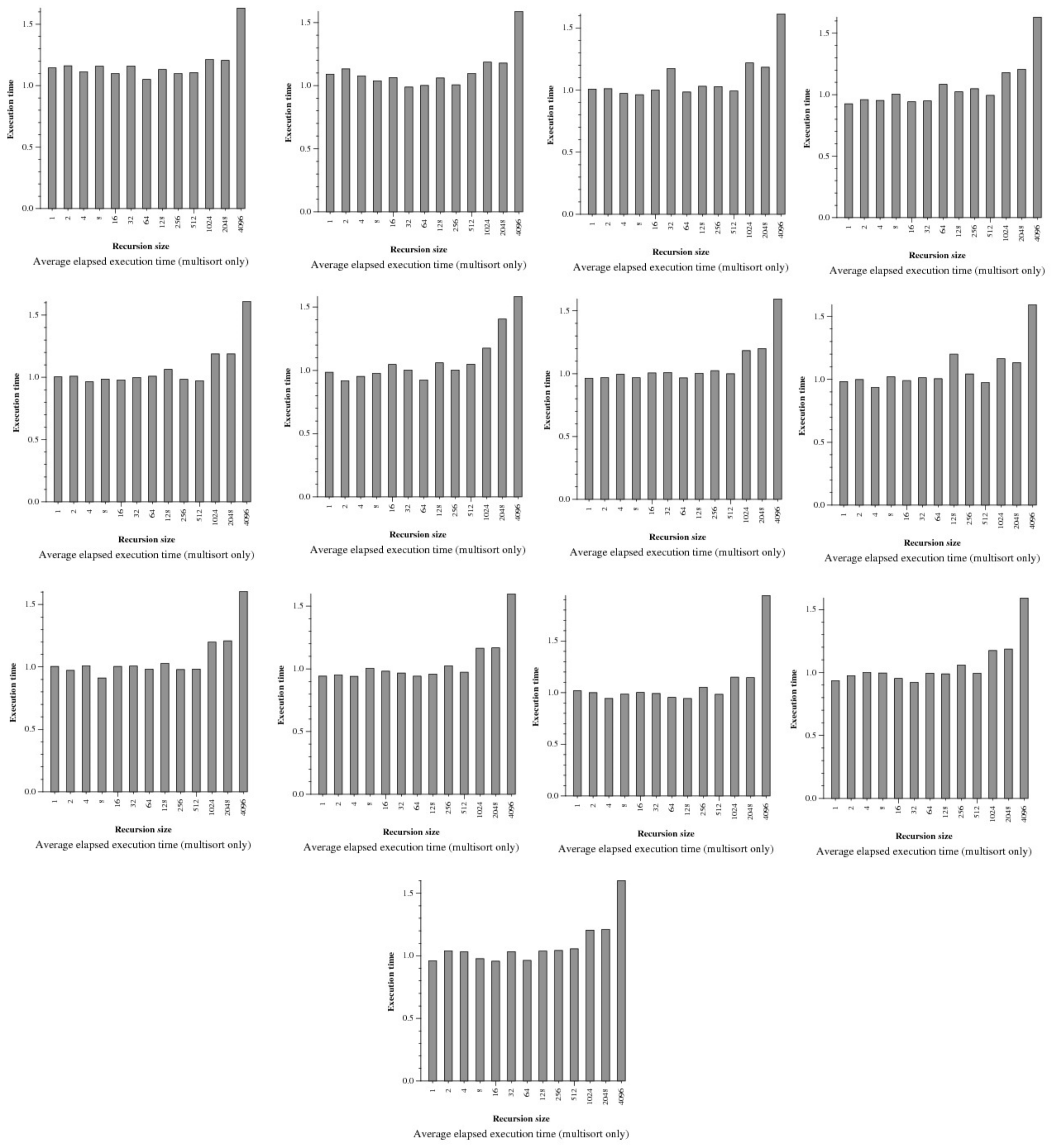
Code 2: Parallelized code of clear function.



Graphic1: Result plot of executed multisort-omp with initialize and clear functions parallelized.

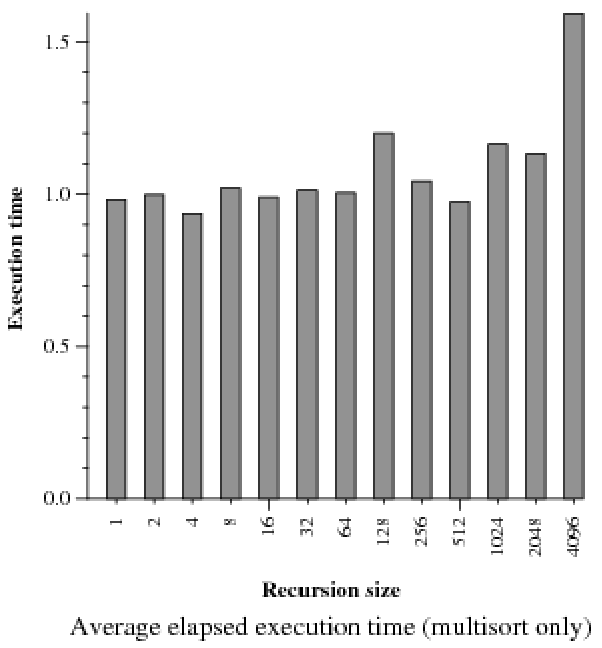
Looking at Code 1 and Code 2, we can appreciate how we parallelized the functions to initialize the vector used later on multisort. The graphic shows the benefits of this parallelization. We can see that the gain obtained is really good.
In conclusion, the improvement over the graphic of image 3, which shows the speedup with the parallelization indicated before, is notable and better than we anticipated.

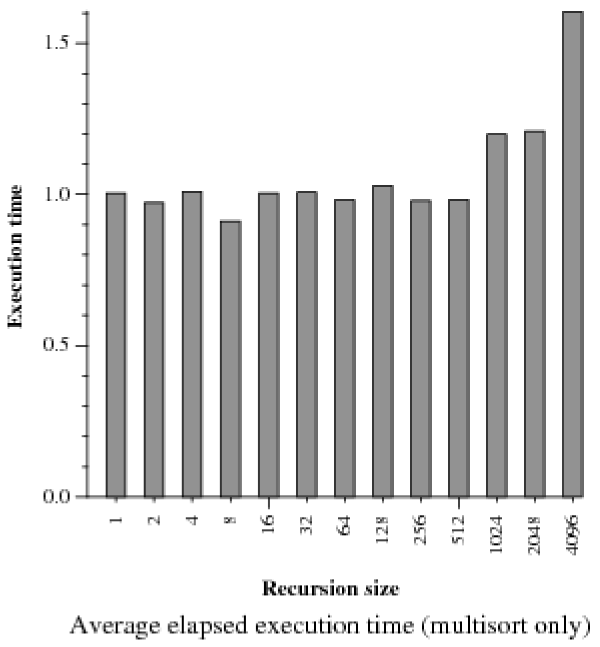
Optional 2: Explore the best possible values for the sort size and merge size arguments used in the execution of the program. For that you can use the submit-depth-omp.sh script, modified to first explore the influence of one of the two arguments, select the best value for it, and then explore the other argument. Once you have these two values, modify the submit-strong-omp.sh script to obtain the new scalability plots.



Graphics 2: Graph evolution of the execution of multisort-omp.c program for various depths (merge size) and sort size (from left to right: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 Kilo-Elements).

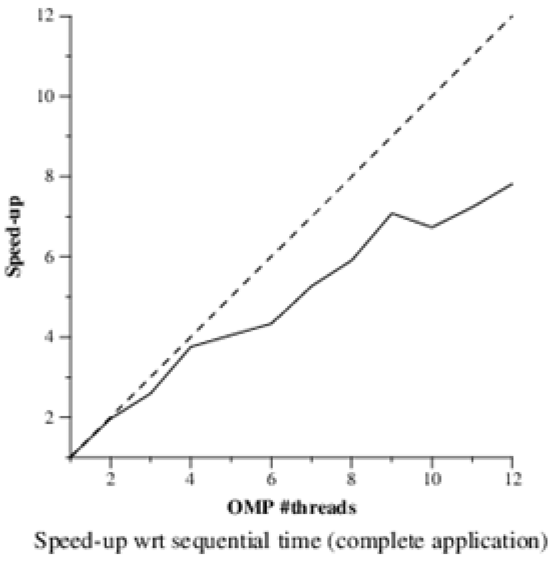
After changing the sizes of the organization (*sort size*) and testing various combinations of deep levels (from *1* to *4096*) we can determinate that the best combinations available are 128 for *sort size* and 256 for *merge size* and vice versa.

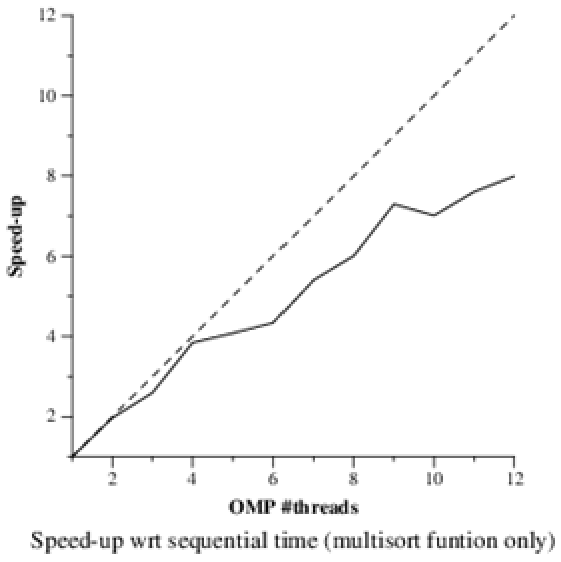




graphic 3: 128 kilo-Elements

graphic 4: 256 kilo-Elements





Graphic 5: Result plots of executed multisort-omp with 128 Kilo-elements for sort size and 256 Kilo-elements for merge size.

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

Conclusions

As we have seen during this and previous deliverables, understanding and exploiting the power of parallelization is beneficial to many of the programs we work with. Previously we had worked with more complex algorithms such as *Mendelbrot* but these last weeks we've experimented with a very simple one.
It's very important to be aware of the ways our code can be executed in parallel, with divide-and-conquer type of algorithms we've found two very easy ways to approach parallelization: *Leaf* and *Tree*. Exploring these solutions with the tools we have at hand will help us determine in which cases we want to use each of these approaches.
In our tests we've found the best results by using a tree parallelization strategy and setting recursivity as shallow as possible (using as many cores as necessary) and then defining dependencies, if the size of the problem is big enough we will see great improvements over a more simple approach.